Граф М.С., Кузьменко О.В.

Архітектура, проєктування та безпека веб-орієнтованих інформаційних та комп'ютерних систем

Навчальний посібник

Передмова

Веб-орієнтовані системи та технології охоплюють широкий спектр напрямів, які пов'язані з проектуванням, розробкою веб-застосунків та веб-сервісів (далі веб-систем), їх технічним супроводом та керуванням процесами їх неперервної інтеграції та неперервного розгортання. Кожен із цих напрямів можна вважати етапом життєвого циклу веб-системи з власним інструментарієм, підходами, принципами, практичним досвідом, які в комплексі дають можливість правильно організувати інфраструктуру та внутрішній процес, його ефективну взаємодію з іншими етапами для забезпечення неперервності, якості та швидкості розробки, доставки та роботи веб-системи.

Прикладами таких напрямів веб-технологій є UI/UX дизайн, адаптивний дизайн, розробка API та веб-сервісів, мікросервісна архітектура, управління та зберігання даних, безсерверні обчислення (FaaS), хмарні сервіси та їх типи, мови розробки, бібліотеки, фреймворки, контейнеризація та оркестрація веб-систем, керування версіями, автоматизація збірки та розгортання. Критично важливими факторами, що визначають успішність веб-системи, є її доступність, безпека, оптимізація продуктивності, масштабованість, оптимізація для пошукових систем.

Крім вище зазначених напрямів, до сучасних трендів сфери веб-орієнтованих систем і технологій можна навести прогресивні веб-застосунки (PWA), Інтернет речей (IoT), веб-аналітику та моніторинг продуктивності, технологію блокчейну.

Суттєвою передумовою для успішного оволодіння курсом "Веб-орієнтовані системи і технології" є знання з фундаментальних основ програмування та об'єктно-орієнтованого програмування, а також з основ веб-дизайну та

веб-розробки. В даному посібнику автори не роблять акцент на безпосередньо розробці веб-застосунків. Переважно, увага відводиться розгляду методології DevOps, ключовими процесами якої є CI/CD – неперервна інтеграція та неперервна доставка (розгортання).

Даний навчальний посібник призначений для всіх, хто хоче навчитися проектуванню, розробці, розгортанню та управлінню захищеними веб-застосунками та веб-сервісами. Він представляє деталізований розгляд таких напрямів веб-розробки, які можуть охопити як процес безпосереднього створення програмного забезпечення, так і технології контейнеризації та автоматизації процесу збірки, налаштування і розгортання веб-застосунків. Посібник поєднує теоретичні концепції з практичними прикладами, що робить його корисним ресурсом як для початківців, так і для досвідчених професіоналів у сфері веб-розробки та технічного супроводу веб-систем.

Розділ 1. Вступ до веб-орієнтованих систем і технологій

1.1 Поняття та ключові характеристики веб-орієнтованих інформаційних систем

Використання сучасних технологій пов’язаних з інформаційними процесами надає ряд можливостей для швидкого доступу до інформації, отримання і передавання її в будь-якій точці світу, роботою з даними онлайн, оптимізації керування процесами, розподілу задач між співробітниками, автоматизації рутинної роботи, здійснення аналітики та ведення звітності тощо.

Інформаційною системою (англ. information system) назвемо сукупність організаційних і технічних засобів для збереження та обробки інформації з метою забезпечення інформаційних потреб користувачів.

Веб-орієнтована інформаційна система або веб-інформаційна система, є типом інформаційної системи, яка в основному працює через Інтернет засобами веб-технологій. Вона призначена для керування, обробки, зберігання та передачі інформації зручним і ефективним способом з використанням можливостей Інтернету та Всесвітньої павутини і надає користувачам доступ до даних і функцій практично з будь-якого місця та в будь-який час.

Приклади веб-орієнтованих інформаційних систем включають:

Системи онлайн-банкінгу. Такі системи дозволяють клієнтам отримувати доступ до своїх банківських рахунків, переглядати історію транзакцій, переказувати кошти та виконувати інші банківські операції через веб-інтерфейс.

Системи електронної комерції – платформи, що дозволяють клієнтам переглядати товари, додавати до своїх кошиків і робити покупки онлайн.

Системи керування вмістом (CMS) дозволяють користувачам створювати, редагувати та керувати цифровим вмістом, таким як веб-сайти, блоги та статті, через веб-інтерфейс.

Системи управління взаємовідносинами з клієнтами (CRM). Дані системи допомагають компаніям керувати взаємодією з клієнтами, відстежувати потенційних клієнтів і аналізувати дані клієнтів.

Навчальні платформи. Навчальні заклади використовують ці системи для надання курсів і освітнього контенту студентам через Інтернет.

Соціальні мережі. Платформи Facebook, Twitter і LinkedIn є прикладами веб-орієнтованих інформаційних систем, які сприяють соціальній взаємодії та обміну інформацією.

Серед інших прикладів таких систем: система продажу квитків, бронювання місць в готелях; система керування польотами; фітнес-трекер; книжкова онлайн-бібліотека; організаційно-навчальна платформа навчального закладу тощо.

Основними характеристиками веб-орієнтованих інформаційних систем є:

- *Доступність.* Користувачі можуть отримати доступ до системи через веб-браузери на різних пристроях, таких як комп'ютери, планшети та смартфони, якщо вони мають підключення до Інтернету.
- *Веб-інтерфейс*. Веб-орієнтовані інформаційні системи зазвичай мають графічний інтерфейс користувача (GUI), який робить взаємодію інтуїтивно зрозумілою та зручною. Користувачі можуть взаємодіяти з системою, натискаючи кнопки, заповнюючи форми та переміщаючись по веб-сторінках.

- *Централізоване зберігання даних*. система зберігає дані на центральному сервері або в базі даних, що полегшує керування та оновлення інформації з єдиного місця.
- *Масштабованість* – можливість обслуговувати велику кількість користувачів одночасно, що має вирішальне значення для програм, які потребують широкого доступу, наприклад платформ електронної комерції або сайтів соціальних мереж.
- *Сумісність між платформами*. Оскільки доступ до веб-орієнтованих інформаційних систем здійснюється через веб-браузери, вони зазвичай сумісні з різними операційними системами та типами пристроїв, не вимагаючи інсталяції спеціального програмного забезпечення.
- *Віддалений доступ*. Користувачі можуть отримувати доступ до системи з різних місць, що забезпечує віддалену роботу та співпрацю.
- *Безпека*. Впровадження належних заходів безпеки має важливе значення для захисту конфіденційної інформації та забезпечення конфіденційності даних, особливо коли йдеться про аутентифікацію користувачів, транзакції та особисті дані.

Отже, веб-орієнтована інформаційна система використовує веб-технології для надання користувачам доступу до інформації та функціональних можливостей через Інтернет, пропонуючи зручність, доступність і масштабованість для різних програм.

1.2 Веб-сайти, веб-сервіси та веб-застосунки

Основними формами організації взаємодії веб-систем та користувачів є *веб-сайти*, *веб-сервіси* та *веб-застосунки*.

Веб-сайт – сукупність взаємопов'язаних веб-сторінок і мультимедійного вмісту, які об'єднані за змістом і навігацією під єдиним доменним ім'ям та доступні у мережі Інтернет через веб-браузер. Веб-сайти можуть бути статичними (з фіксованим вмістом) або динамічними (з вмістом, створеним на основі взаємодії користувача або обробки на стороні сервера). Вони в основному призначені для представлення інформації, надання ресурсів і залучення користувачів. Веб-сайти можна віднести до різних категорій, таких як інформаційні, електронна комерція, блоги, соціальні мережі тощо.

Веб-сервіси – програмні засоби, що забезпечують обмін та обробку інформації між різними системами: веб-сайтами, іншими сервісами чи застосунками, базами даних. Вони часто використовуються для надання певних функцій системи іншим системам чи програмам стандартизованим способом. Серед типових функцій веб-сервісів – отримання даних, виконання обчислень, керування складними бізнес-процесами. Веб-сервіс може працювати як окремий додаток, або бути частиною веб-сайту чи складної веб-системи. Приклади: картографічні сервіси, сервіси для отримання даних про погоду, відслідковування трендів та аналізатори пошукових запитів, сервіси для обміну миттєвими повідомленнями, отриманні чи наданні інформації про вільні місця в готелях тощо.

Веб-застосунки – програми, інструменти, доступні в Веб, орієнтовані на безпосередню взаємодію з користувачем для виконання певних завдань, обробки даних певного типу та

досягнення конкретних цілей безпосередньо через інтерфейс браузера. Приклади веб-застосунків можуть включати текстовий або табличний онлайн-редактор, системи аналізу даних, електронні системи онлайн-банкінгу, платформи електронної комерції, клієнти електронної пошти, інструменти управління проектами.

Веб-сайти, веб-сервіси та веб-застосунки багато в чому функціонально споріднені, проте мають ключові відмінності:

По функціональності:

- Веб-сайти насамперед надають користувачам інформацію, ресурси або вміст.
- Веб-сервіси полегшують спілкування та обмін даними між різними програмними додатками.
- Веб-додатки пропонують інтерактивні функції, що дозволяє користувачам виконувати певні завдання та операції через інтерфейс на основі браузера.

Щодо взаємодії з кінцевим користувачем:

- Веб-сайти зосереджені на представленні вмісту користувачам у зручному форматі.
- Кінцеві користувачі зазвичай не мають прямого доступу до веб-служб; вони використовуються для спілкування між програмами.
- Веб-застосунки надають веб-інтерфейс, який забезпечує пряму взаємодію з користувачами, дозволяючи їм вводити дані, здійснювати вибір і отримувати результати.

Щодо використання комунікаційного протоколу:

- Веб-сайти та веб-застосунки часто використовують HTTP/HTTPS для зв’язку між браузером користувача та веб-сервером.
- Веб-сервіси можуть використовувати різні протоколи, такі як SOAP, REST тощо, залежно від їх архітектури та призначення.

Таким чином, веб-сайти зосереджуються на представленні вмісту, веб-сервіси забезпечують зв’язок між системами, веб-застосунки забезпечують інтерактивні функції для користувачів для виконання певних завдань. Сучасні інформаційні системи – великі розподілені системи, що поєднують декілька форм організації інформаційних процесів, розглянутих вище. Приклади комплексних веб-систем є хмарні системи (розподілені обчислення, прикладні додатки, веб-хостинг, дискові сховища, сервіси), соціальні мережі, відео-хостинги, глобальні торговельні площадки тощо.

Розділ 2. Операційні системи. Вступ до операційної системи Linux

2.1 Операційні системи, їх структура та особливості

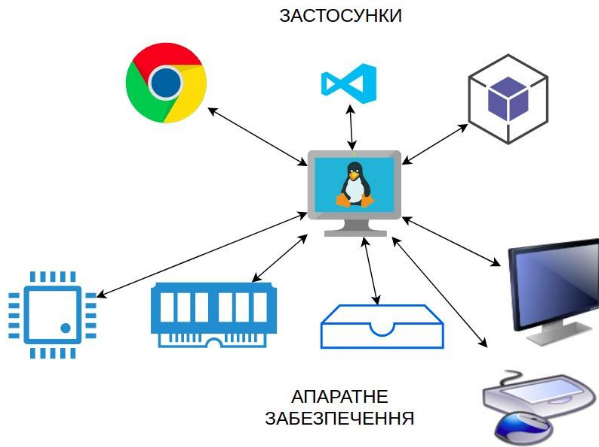
Визначимо поняття операційних систем, їх призначення, структуру та особливості. Зауважимо, що при розробці та користуванні прикладним програмним забезпеченням необхідно враховувати, що

- застосунки не повинні безпосередньо взаємодіяти з апаратними пристроями;
- застосунки не можуть бути встановлені прямо на апаратне забезпечення;

Таким чином існує посередник між апаратним забезпеченням та застосунками: *операційна система*.

Наведемо деякі функції операційної системи:

- Операційна система є транслятором між застосунком і апаратним забезпеченням (виділяє пам'ять, обробляє чи передає команди введення/виведення тощо).
- Керує та розподіляє ресурси між застосунками (вирішує який застосунок та скільки повинен використовувати ресурсів)
- Ізолює контент різних застосунків



Операційні системи використовуються для обслуговування веб-серверів. Особливостями таких операційних систем є:

- В більшості використовується Linux
- Більш легкі та продуктивні
- Не мають GUI чи деякихприкладних користувацьких застосунків

Задачі операційної системи

До задач операційно системи віднесемо:

- Виділення та керування процесами (CPU)
- Керування оперативною пам’яттю
- Керування постійною пам’яттю
- Керування файловою системою
- Керування пристроями вводу та виводу
- Керування безпекою та мережею

Керування процесами (CPU).

Процес - це одиниця виконання на комп’ютері певної задачі (запуск та робота застосунку). Кожен процес має власний ізольований простір (що не призводить до збоїв при відкритті наприклад декількох застосунків). 1 процесор виконує 1-ну задачу одночасно. Проте, він перемикається так швидко, що у нас складається враження багатозадачності.

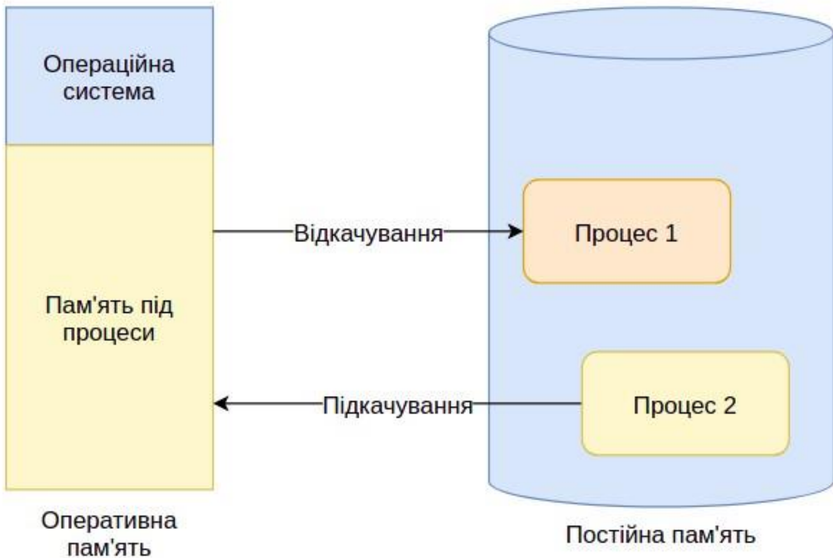
На сьогодні, типовий комп’ютер має декілька процесорів (Dual-Core, Quad-Core). Це дає змогу виконувати процеси паралельно.

Керування пам’яттю.

RAM (Rapid Access Memory) (обмежена величина) - швидкодійна комп'ютерна пам'ять, призначена для поточної обробки даних – запиту, читання та зберігання інформації у процесі її обробки. Серед основних функцій ОС по керуванню пам’яттю можна виділити наступні:

- виділення робочої пам’яті для запуску і роботи застосунку;
- підкачка (обмін) пам’яті (memory swapping).

Підкачка — це схема керування пам’яттю, за якої будь-який процес можна тимчасово перемістити з основної пам’яті на додаткову, щоб основна пам’ять стала доступною для інших процесів. Він використовується для покращення використання основної пам'яті. У вторинній пам'яті місце, де зберігається викачаний процес, називається простором підкачки.



Керування постійною пам’яттю (storage):

- Дані для постійного зберігання: файли, програмний код, системні файли інтегрованого середовища розробки, конфігурації та модулі;
- Керування підкачкою: під час користування, пам’ять для даних переноситься в оперативну з постійної і навпаки. *Керування файловою системою*
- Структурованість;
- В Unix-системах: деревовидна структура;
- В системах Windows: багатодискова деревовидна структура

Керування пристроями вводу/виводу

- Обробка та передача повідомлень і команд між застосунками та пристроями.

Керування безпекою

- Користувачі та дозволи;
- Кожен користувач має свій власний простір: робочий стіл, файли, налаштування;
- Кожен користувач має ролі та дозволи: адміністратори, користувачі, гості.

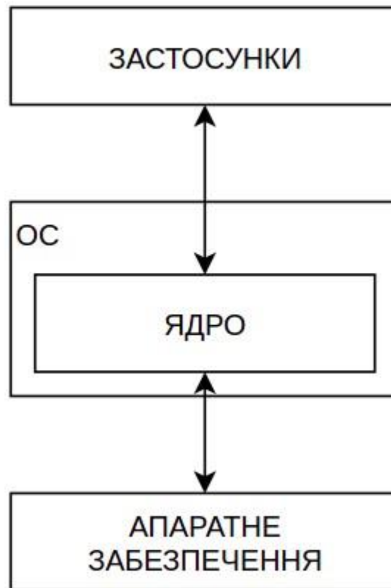
Керування мережею

- Порти, IP-адреси
- Доступ до мережевих ресурсів
- Мережевий екран

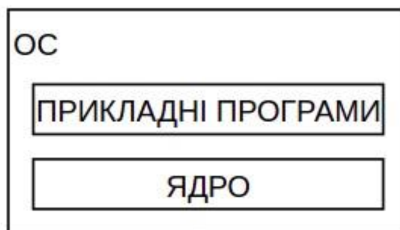
Компоненти операційної системи

Ядро операційної системи (kernel) – програма низького рівня, що є основою будь-якої ОС, містить драйвери, диспетчер, планувальник, файлову систему тощо. Ядро завантажується в першу чергу та володіє такими функціями і особливостями:

- Керує апаратними компонентами системи;
- Містить драйвери для керування пристроями вводу та виводу;
- Запускає процес для застосунку
- Виділяє ресурси при роботі;
- Звільняє ресурси при завершенні роботи застосунку;
- Різні ОС можуть мати різне ядро;
- Взаємодія з ядром може здійснюватись через графічний інтерфейс користувача або інтерфейс командного рядка



Шар прикладних програм (різні дистрибутиви). Дистрибутиви Linux: Ubuntu, Mint, Debian, CentOS мають спільне ядро Linux але різні шари прикладного ПЗ: різна кольорова схема, іконки, GUI, різні застосунки упаковані в дистрибутив, файл-менеджер, менеджер застосунків, мови, CLI, компілятори тощо.



Як окремий приклад дистрибутив на основі ядра Linux можна навести ОС Android. Крім того що даний дистрибутив має

повністю інший прикладний шар, він встановлюється на мобільних пристроях. Наведемо структуру ОС Android:

Applications
Home
Contacts
Phone
Browser
Camera

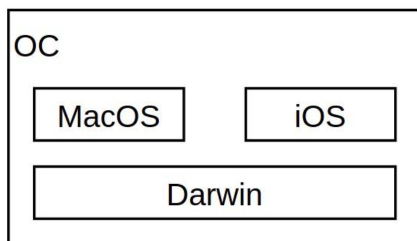
Application Framework
Activity Manager
Window Manager
Content Providers
View System
Package Manager
Package Manager
Telephony Manager
Resource Manager
Location Manager

Libraries
Surface Manager
Media Framework
SQLite
OpenGL|ES
FreeType
Webkit
SGL
SSL
Libe

Android Runtime
Libraries Core
Dalvik Virtual Machine

Linux Kernel
Display Driver
Camera Driver
Flash Memory Driver
Binder (IPC) Driver
Keypad Driver
WiFI Driver
Audio Driver
Power Management

Операційні системи MacOS та iOS також використовують спільне ядро – Darwin.



Існує три найбільш популярні операційні системи: Linux, MS Windows та Mac OS. Кожна з них має різні версії. Ядра цих систем як правило оновлюються рідко, в той час як прикладний рівень – достатньо часто. Linux та Windows мають клієнтські та серверні дистрибутиви. Командний рядок та файлова структура MacOS подібні до Linux, в той час Windows в цьому повністю відрізняється. Список версій MacOS доступний за адресою: https://support.apple.com/en-us/HT201260

Основою для багатьох ОС є операційна система *UNIX,* рік виходу 1-ї версії якої 1970. Наприклад, ядро Darwin операційної системи MacOS побудоване поверх Unix. Linux створена без використання вихідного коду Unix. Проте Linux використовує ту ж концепцію, що і Unix. Таким чином Linux не є версією Unix, проте її часто називають Unix-подібною.

Для сумісності систем, побудованих на основі Unix, було запропоновано деякі стандарти для застосунків, що працюють на

таких системах з метою їх роботи на всіх Unix-подібних системах. Одним із таких стандартів є POSIX (Portable Operating System Interface). Т.ч. Linux та MacOS відповідають стандартам POSIX, тому в них є багато чого спільного (CLI, деякі застосунки тощо).

В даному розділі буде здійснено огляд операційної системи Linux. Наступні розділи даного підручника розглядають переважно технології, що використовуються в процесі розробки, побудови, розгортання та технічного супроводу веб-застосунків. Linux встановлена на більшості сучасних веб-серверах, а також більшість сучасних технологій є Linux-нативними (Docker, Kubernetes, Ansible тощо). Тому знання даної ОС є важливим і необхідним не лише для системних адміністраторів, але й для розробників, DevOps-інженерів, QA-інженерів, IT-менеджерів тощо.

2.2 Віртуалізація та контейнеризація

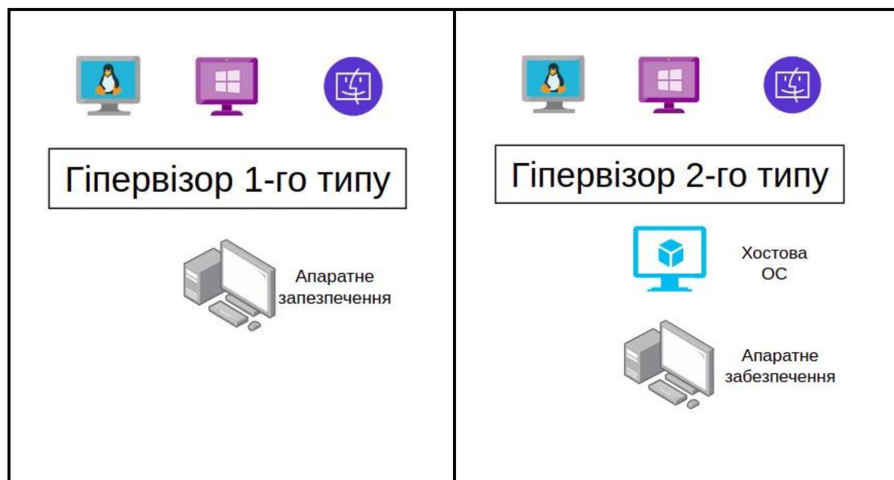
Первинно різні ОС встановлюються на різні комп’ютери. Віртуалізація дає можливість обійтися без окремого комп’ютера для використання потрібної ОС.

Таким чином, можна встановити будь-яку операційну систему на будь-яку іншу, використовуючи віртуалізацію. Основним засобом для віртуалізації є *гіпервізор*.

Гіпервізор або *монітор віртуальних машин* — комп'ютерна програма або обладнання процесора, що забезпечує одночасне і паралельне виконання декількох віртуальних машин, на кожній з яких виконується власна операційна система, на одному фізичному комп'ютері, що називається хост-машиною. Гіпервізор забезпечує взаємну ізоляцію операційних систем, що виконуються на віртуальних машинах, шляхом розділення фізичних та логічних пристроїв між декількома віртуальними машинами.

Існує 2 типи гіпервізорів. Гіпервізори 1-го типу керують розподілом ресурсів одразу з фізичного рівня, тобто без хостової ОС. Такі гіпервізори часто називають bare-metal гіпервізорами.

Типове використання: сервери, сервісні хмарні платформи Популярними гіпервізорами 1 типу є HyperKit для macOS, Hyper-V для Windows та KVM для Linux.



Гіпервізори 2-го типу зазвичай встановлюються на наявну операційну систему (основна ОС). Такі гіпервізори працюють в одному кільці з ядром основної (хостової) ОС. Керування зверненнями до ЦП, розподіл пам'яті, сховища та мережевих ресурсів може здійснюватись як основною ОС так і гіпервізором автономно, проте доступ до пристроїв вводу-виводу комп'ютера з гостьової ОС здійснюється через звичайний процес основної ОС — монітор рівня користувача. Типове використання: персональні комп'ютери. Популярними гіпервізорами 2 типу є VirtualBox і VMWare.

VirtualBox є одним із найбільш поширених гіпервізорів. Це open-source програма віртуалізації для операційних систем від Oracle Corporation. Вона встановлюється на наявну операційну систему, яка називається хостовою, усередину цієї програми встановлюється інша операційна система, яку називають гостьовою операційною системою. Підтримується практично

всіма основними операційними системами Linux, FreeBSD, Mac OS X, Microsoft Windows, які підтримують роботу гостьових операційних систем FreeBSD, Linux, OpenBSD, OS/2 Warp, Windows і Solaris.

Ключовими можливостями VirtualBox є:

- Отримання ресурсів апаратного забезпечення від хостової ОС;
- Створення віртуального процесора, віртуальної операційної пам'яті, віртуального диску для кожної віртуальної машини;
- Обмеженість віртуальних ресурсів фактично наявними фізичними ;
- Повна ізольованість віртуальних машин.

Віртуальні машини для звичайних користувачів можуть бути корисними для

- встановлення інших операційних систем поверх основної (не потрібно купувати окремий комп'ютер);
- захисту основної ОС;
- тестування роботи застосунку на різних ОС;
- навчання або прикладне використання різних ОС.

В хмарних платформах користувачі створюють власні ізольовані віртуальні середовища, при цьому вони можуть вибирати різні варіанти ресурсів.

Сучасні компанії в більшості вибирають віртуальні машини для хостингу власних інформаційних систем. Переваги віртуалізації наведемо в порівняльній таблиці:

Без віртуалізації	*З віртуалізацією*

ОС тісно прив’язана до апаратного забезпечення	ОС є фізично файлом (образом віртуальної машини), що включає в себе: саму ОС, всі застосунки, конфігурацію
При виході з ладу пристрою, є висока ймовірність втрати даних та збою роботи сервісів	Гнучкість у створенні та керування резервними копіями
	Високий рівень та простота керування захистом
	Портативність
	Незалежність від фізичного серверу

Зважаючи на переваги віртуалізації, з її основами повинні бути знайомі всі члени проектної команди: інженери хмарних сервісів, системні адміністратори, розробники, DevOps-інженери тощо.

Практична робота 1.1. Установка операційної системи Linux на віртуальну машину

Мета роботи: підготувати середовище віртуальної машини, встановити операційну систему Linux на віртуальну машину, налаштувати обмін даними між гостьовою та основною операційними системами.

Передумови

Для роботи віртуальної машини необхідно мати щонайменше 4 Gb пам’яті

Порядок роботи

1. Завантажимо і встановимо гіпервізор VirtualBox.
2. Підготуємо середовище віртуальної машини:
    - Задамо ім’я, папку тип та версію віртуальної машини
    - Виділимо оперативну пам’ять (RAM) (наприклад, 2048 Mb)
    - Створимо віртуальний жорсткий диск (наприклад, 10 Gb)
    - Задамо тип жорсткого диску VDI (Virtual Disk Image)
    - Задамо тип сховища (Dynamically allocated або Fixed size)
    - Задамо розташування файлу і розмір віртуального диску
3. Здійснимо установку операційної системи Linux з образу:
    - Завантажимо образ системи (https://ubuntu.com/download) (LTS-версія)
    - Запустимо створену віртуальну машину (ВМ). При першому старті потрібно вибрати iso-образ Linux і запустити установку
    - В установщику потрібно вибрати мову, часову зону тощо

- При установці виникне повідомлення “Erase disk and install Ubuntu). Не варто турбуватись: це віртуальний диск, тому файли нашої основної ОС не постраждають
- Створимо Linux-користувача
- Установимо мовні пакети, застосунки
- В середовищі ВМ перезавантажимо Linux, здійснимо деякі кроки налаштувань і система готова до використання

4. Налаштуємо обмін даними між гостьовою ОС (Linux) і основно ОС:
   - Для керування буфером обміну (операції копіювання та вставки) та операціями переміщення методом Drag’n’Drop. Це зручно для копіювання команд, посилань, снапшотів. Для цього потрібно:
   - Зробити відповідне налаштвання ВМ: General - Advanced – Shared Clipboard
   - Установити VM VirtualBox Extension Pack (All supported platforms).
   - Встановити гостьові доповнення: в вікні VM вибрати Devices – Insert Guest Additions CD image…
   - При завершенні роботи можна зберегти поточний стан системи або повністю вимкнути операційну систему
5. Зробіть висновок до практичної роботи.

2.3 Файлова система Linux

Операційні системи зберігають дані на диску за допомогою файлових систем. Класична файлова система представляє дані як вкладені каталоги (папки), у яких містяться файли. Існує каталог, що є «вершиною» файлової системи – кореневий каталог. У ньому містяться решту каталогів і файлів.

Файлова система Linux - деревоподібна (tree-shared) складна структура, що починається з кореня. Вона складається з каталогів, підкаталогів. Кожен файл та файлова система взаємопов'язані між собою.

Для порівняння, у файловій системі Windows жорсткий диск може бути розбитий на розділи, то кожному розділі організується окрема файлова система з власним коренем і структурою каталогів (адже розділи повністю ізольовані друг від друга).

Файлова система Linux	Файлова система Windows
Деревовидна ієрархічна структура 1 кореневий каталог	Має багато кореневих каталогів (диски, драйви) A, B – зйомні диски C – локальний системний диск D – диск з даними

Важливо відмітити, що *в Linux всі ресурси є файлами*:

- текстові документи, мультимедійні дані, бази даних, бінарні файли, команди, бібліотеки
- папки
- пристрої комп'ютера представляються також у формі файлів: драйвери, файлові системи, введення/виведення

Основні папки операційної системи Linux:

/home – домашня папка для користувачів. В ній для кожного користувача створюється своя папка, що використовується для персональних файлів. Оскільки ОС може мати багато користувачів, тому кожен користувач має свій персональний

каталог та власну конфігурацію. Домашня папка користувача відкривається в провіднику по замовчуванню.

/root – домашня папка для супер-користувача (root).

/bin – містить бінарні застосунки для виконання більшості базових команд (cat, cp, echo, …)

/sbin – системні бінарні команди (adduser, chgpassword, iptables тощо). Для виконання вимагають дозвіл супер-користувача. Можуть виконуватись як адміністратором так і самою системою.

/lib – містить бібліотеки для бінарних застосунків. Застосунки в ОС Linux можуть бути розділені на декілька папок: */bin*, */lib*.

/usr – рівень, що містить ті ж папки */bin*, */sbin*, */lib* тощо. Причина дублювання полягає в необхідності такого розділення в минулому, коли потрібно було враховувати обмеження в дисковому сховищі. Як правило в цій папці більше команд, ніж в кореневій.

/usr/local – застосунки, встановлені самим користувачем (third party applications): *docker, docker compose, minikube, java, python* тощо. Дані застосунки доступні всім користувачам. Для суто власних потреб, застосунки встановлюються в */home*.

/opt – (додаткова папка, optional folder). Сюди встановлюються застосунки, яким не властиве розділення на */bin* та на */lib*. Наприклад, *lampp, chrome, viber, zoom, teamviewer*.

/boot – містить файли, необхідні для завантаження.

/etc (et cetera) – конфігурація системи: мережева, конфігурація користувачів, груп, паролі, хости, бази даних.

/dev (devices) – папка для підключених пристроїв (драйверів), системних файлів для взаємодії системи та пристроїв.

/var – папка для логування процесів, кешування даних.

/tmp – тимчасові ресурси, необхідні для роботи процесів. При завершенні роботи процесу, видаляються.
/media – зовнішні пристрої (зовнішній жорсткий диск, USB flash-диск тощо).

/mnt – папка, що переважно використовується адміністраторами для монтуванні додаткових файлових систем.

Взаємодія користувача з папками ОС Linux

Зазвичай користувач не має справу безпосередньо з папками. Установка застосунків здійснюється з використанням менеджеру пакунків, якому делегується взаємодія з папками ОС Linux. Менеджер пакунків розподіляє ресурси застосунку по папках /bin, /lib тощо. Конфігурацію змінювати бажано на рівні застосунку, вона автоматично оновлюється в папці */etc*. Підключення зовнішнього медіа-пристрою автоматично створює папку в */media.*

Проте для відлагодження та виявлення проблем, потрібно бути знайомим з папками та їх призначенням.

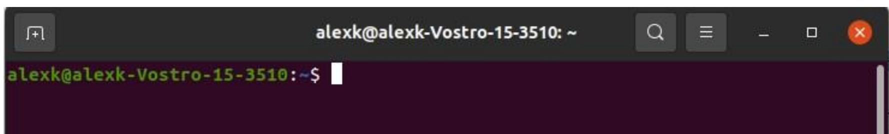
Приховані файли

Приховані файли переважно використовуються для попередження випадкового видалення важливих даних, а також зберігання конфігурації, логів, скриптів виконання тощо. В більшості автоматично генеруються застосунками чи ОС, проте користувач може створювати приховані файли власноруч. Імена прихованих файлів починаються з крапки (dot). Такі файли називаються dotfiles. В домашній папці користувача містяться системні приховані папки та файли: */.cache, /.local, /.config, .bash_history, .profile* тощо.

Для увімкнення перегляду прихованих файлів в даній папці виконаємо *Options – Show Hidden Files*.

2.4 Базові команди Linux

Команда Linux — це програма або утиліта, яка запускається в командному рядку. Командний рядок — це інтерфейс, який приймає рядки тексту та обробляє їх у інструкції для комп'ютера. *В Linux* застосунок командного рядка зветься терміналом. *Термінал* – візуальне вікно для *введення* команд та *виведення* результатів їх виконання. Для вікдриття терміналу, в ОС Ubuntu використовується сполучення *ctrl+alt+T.*



alexk - ім'я поточного користувача *alexk-Vostro-15-3510* - ім'я комп'ютера. У випадку сервера – це може бути ім'я хосту.

: (двокрапка) - роздільник

~ (тильда) – домашня директорія поточного користувача (скорочено)

$ (долар) – представляє звичайного користувача. Супер-користувач позначається символом решітки *#*.

Наведемо варіанти та можливості використання GUI та CLI:

GUI (графічний інтерфейс користувача):

- GUI – можливість взаємодіяти з системою через візуальні графічні компоненти: кнопки, вікна, поля тощо;
- GUI є лише абстракцією інтерфейсу командного рядка. Наприклад, при закритті вікна (клацання на кнопку *X*), за цією дією виконується команда;
- GUI зручно використовувати в прикладних задачах: обробка мультимедійних даних (відео, аудіо), електронні таблиці, текстові редактори, браузери.

CLI (інтерфейс командного рядка):

- CLI – введення команд в спеціальному терміналі та вивід результатів виконання команд;

- На серверах зазвичай є можливість використовувати лише CLI;
- На персональних комп’ютерах корисно також володіти CLI;
- Якщо звикнути працювати через CLI, робота буде більш ефективною: не потрібно переходити по файловій системі, працювати з багатьма вікнами;
- Через CLI зручніше обробляти масові операції з файлами та папками;
- CLI має більші можливості ніж GUI.

Існують найбільш типові команди, що стосуються навігації та роботи з файловою системою: створення папок та файлів, перейменування, переміщення, копіювання чи видалення, перегляд вмісту, робота з прихованими файлами тощо. Інші команди можуть стосуватись отримання інформації про стан та дані стосовно операційної системи, установки прикладних застосунків, конфігурування роботи сервісів. **Операції з папками та файлами *pwd*** – вивести поточну директорію (print working directory) ***ls*** – вивести список файлів та папок в поточній директорії (list)

- `ls /etc/network` – вивести вміст заданої директорії
- `ls -a` – вивести вміст директорії включно з прихованими файлами

cd – змінити робочу папку(change directory)

- `cd myProject` – відносний шлях (без слешу на початку)
- `cd ..` – перейти вище на один рівень по дереву папок
- `cd /` – перейти в кореневий каталогу (абсолютний шлях)
- `cd /opt` – перейти в папку /opt (абсолютний шлях) ● `cd /opt/lampp` – приклад папки із повним шляхом
- `cd Do (tab)` – виведення папок, що починаються з *Do*

- (можливість автодоповнення) ***mkdir*** –
створити директорію (make directory) `mkdir myProject` – створити папку myProject в поточній
(робочій) папці
- `mkdir Documents/myProject` – створити папку myProject за заданим шляхом

touch – створити файл

- `touch myFile.txt` – створити файл з іменем myFile.txt ***rm*** – видалити файл або папку
- `rm myFile.txt` – видалити файл myFile.txt
- `rm -r myProject` – видалити папку myProject ***mv*** – перейменувати файл або папку
- `mv myProject myNodeJSProject` – перейменувати папку myProject на myNodeJSProject

cp – зробити копію файлу або папки

- `cp myFile.txt myFileCopy.txt` – зробити копію файлу myFile.txt
- `cp -r myNodeJSProject myNodeJSProjectCopy` – зробити копію папки myNodeJSProject

cat – показати вміст файлу

- `cat myFile.txt` – показати вміст файлу myFile.txt **Корисні команди та можливості**
- `clear` – очистити термінал
- `ls -R Documents` – переглянути вміст папки Documents, а також всіх інших вкладених папок

-
- З допомогою клавіш “Вверх” та “Вниз” можна здійснювати навігацію по раніше виконаних командах
- history – вивести всі або задану кількість виконаних команд поточної сесії
- ctrl+r – (search history) пошук команди в історії
- ctrl+c – зупити поточний процес, що працює в терміналі
- ctrl+shift+v (або shift+ins) – вставити команду з буферу обміну

  man <command_name> – вивід документації по команді з описом її параметрів, наприклад man ls

```
alexk@alexk-Vostro-15-3510: ~
LS(1)                        User Commands                        LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
       fied.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
 Manual page ls(1) line 1/241 8% (press h for help or q to quit)
```

Розглянуті команди є базовими. В Linux є сотні команд, які не потрібно пам’ятати. Для виконання специфічних задач можна використовувати довідники або пошуковик Google.

-

Вивід інформації про ОС

- `uname -a` – вивести інформацію про систему: ядро, дистрибутив, тип системи тощо
- `cat /etc/os-release` – вивести інформацію про систему з файлу
- `lscpu` – розширена інформація про процесор
- `lsmem` – інформація про пам’ять

Виконання команд супер-користувача

- `ls /sbin` – команди супер-користувача (*adduser, addgroup, delgroup, deluser* та інші)
- для виконання таких команд в режимі регулярного

користувача використовується на початку слово *sudo*: `sudo adduser [username]`

- для першого виконання з sudo, потрібно ввести пароль.
- `su - [username]` – переключення між користувачами
(потрібно вводити пароль)

2.5 Установка застосунків

Застосунками зазвичай називають програмне забезпечення, яке має графічний інтерфейс користувача (GUI). Операційна система Linux, зокрема її дистрибутив Ubuntu, пропонує певний набір застосунків “з коробки” для вирішення повсякденних завдань. Користувач звичайно має можливість встановити більше застосунків, щоб зробити операційну систему більш корисною.

Застосунки постачаються у спеціальному архіві, що містить всі файлу застосунку, – програмному пакунку. Пакунки доступні в двох форматах: snap-пакунки та Debian-пакунки. Інколи застосунки можуть мати два формати пакунків одночасно і в такому разі, Ubuntu Software виводить у списку спочатку span-пакунок.

При встановленні застосунків не рекомендується використовувати інсталятори за типом “майстер установки”. Застосунки зазвичай мають залежності, які також потрібно встановити. Врахуємо, що файли застосунку пожуть розділятись на папки (/bin, /lib тощо).

Тому в більшості для встановлення застосунків використовується менеджер пакунків (package manager). Можливостями менеджера пакунків є:

- завантаження, встановлення чи оновлення наявного програмного засобу з репозиторію;
- забезпечення цілісності та автентичності пакунку;
- керування та вирішення потрібних залежностей;
- знання куди покласти файли застосунку; ● здійснення оновлення.

Менеджер пакунків включений в кожен дистрибутив Linux. В Ubuntu є доступний APT Package Manager (Advanced Package

Tool). APT володиє командами для установки, видалення або оновлення застосунку.

Введемо команду apt і отримаємо довідку по даному менеджеру:

```
Most used commands:
  list - list packages based on package names
  search - search in package descriptions
  show - show package details
  install - install packages
  reinstall - reinstall packages
  remove - remove packages
  autoremove - Remove automatically all unused packages
  update - update list of available packages
  upgrade - upgrade the system by installing/upgrading packages
  full-upgrade - upgrade the system by removing/installing/upgrading packages
  edit-sources - edit the source information file
  satisfy - satisfy dependency strings
```

Приклади команд

- `sudo apt search [назва_пакунку]` – пошук доступних застосунків
- `sudo apt install [назва_пакунку]` – установка застосунку
- `sudo apt remove [назва_пакунку]` – видалення застосунку

Деякі застосунки можуть мати власний інтерфейс командного рядка. Якщо ввести команду деякого застосунку (наприклад, `java`), проте даний застосунок ще не встановлено, отримаємо рекомендації від Linux по встановленню даного застосунку:

```
alexk@alexk-Vostro-15-3510:~$ java

Command 'java' not found, but can be installed with:

sudo apt install openjdk-11-jre-headless  # version 11.0.18+10-0ubuntu1~20.04.1, or
sudo apt install default-jre              # version 2:1.11-72
sudo apt install openjdk-16-jre-headless  # version 16.0.1+9-1~20.04
sudo apt install openjdk-17-jre-headless  # version 17.0.6+10-0ubuntu1~20.04.1
sudo apt install openjdk-8-jre-headless   # version 8u362-ga-0ubuntu1~20.04.1
sudo apt install openjdk-13-jre-headless  # version 13.0.7+5-0ubuntu1~20.04
```

Так наприклад можна встановити *docker, docker compose, npm, node* та інші.

В якості альтернативи apt, проте рідше, використовується менеджер пакунків apt-get. Відмінності apt і apt-get представимо в таблиці:

Можливо	APT-GET	APT
сті		
Інтерфейс користувача	Можна виконувати ті ж функцій, але з більшою кількістю опцій	Має меншу кількість опцій, але зручніше організований інтерфейс
	Немає прогрес-бару, об’ємний лог при установці	Більш “дружелюбний”. Смужка прогресу наявна
	Команда *search* не доступна	Інший набір команд. Має можливість пошуку з командою-параметром *search*
Рівень контролю	APT-GET – інструмент більш низького рівня, проте дає більше контролю керування пакунками	APT є більш автоматизованим і обробляє залежності пакетів самостійно
Сумісність	APT-GET може бути більш сумісний із старими версіями цих дистрибутивів	APT більш сумісний із новими версіями дистрибутивів на основі Debian

Виходячи з даної таблиці віддається перевага менеджеру APT, що і рекомендується сучасними дистрибутивами Linux для використання.

Менеджер пакунків завантажують файли для установки з репозиторіїв. *Репозиторії пакунків* – сховища, що містять файли застосунків. Перед оновленням або установкою застосунку завжди потрібно оновлювати *індекс пакунків* (package index) в ОС, який використовується менеджером APT для пошуку потрібного застосунку.

Індекс пакунків – це, по суті, база даних доступних пакунків з репозиторіїв, визначених у файлі */etc/apt/sources.list* і в каталозі */etc/apt/sources.list.d*. Для оновлення локального індексу пакунків останніми змінами, присутніми на репозиторіях, потрібно виконати команду: `sudo apt update`

Дана команда підтягує актуальні зміни з репозиторіїв APT і оновлює записи в індексі про доступні пакунки на репозиторіях.

Ознайомитись з індексом пакунків можна, вікривши для перегляду файл sources.list:

```
alexk@alexk-Vostro-15-3510:/etc/apt$ cat sources.list

# See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade to
# newer versions of the distribution.
deb http://archive.ubuntu.com/ubuntu/ focal main restricted
# deb-src http://archive.ubuntu.com/ubuntu/ focal main restricted

## Major bug fix updates produced after the final release of the
## distribution.
deb http://archive.ubuntu.com/ubuntu/ focal-updates main restricted
# deb-src http://archive.ubuntu.com/ubuntu/ focal-updates main restricted

## N.B. software from this repository is ENTIRELY UNSUPPORTED by the Ubuntu
## team. Also, please note that software in universe WILL NOT receive any
## review or updates from the Ubuntu security team.
deb http://archive.ubuntu.com/ubuntu/ focal universe
# deb-src http://archive.ubuntu.com/ubuntu/ focal universe
deb http://archive.ubuntu.com/ubuntu/ focal-updates universe
# deb-src http://archive.ubuntu.com/ubuntu/ focal-updates universe
```

Звісно, існують застосунки або нові версії застосунків, що не належить до офіційних репозиторіїв: деякі браузери, середовища розробки (IJ), офісні пакети. Для їх встановлення використовуються альтернативи APT і APT-GET:

- Ubuntu Software Center – програмний засіб для керування застосунками через GUI, встановлений по замовчуванню на Ubuntu. В Ubuntu Software Center можна переглядати ресурс та версію встановлених застосунків, а також встановлювати при потребі інші версії.
- Snap Package Manager – менеджер пакунків, що використовується для ОС з ядром Linux Kernel. Ключовим при роботі є те, що snap-пакунок повністю містить упакований застосунок та всі залежності. Для завантажування пакунків та для перегляду та установки ПЗ використовується SnapStore. Для побудови та публікації snap-пакунків використовується CLI та фреймворк SnapCraft.

APT	Snap
Бінарні файли та залежності розділені по папках /bin, /lib, … (Linux рекомендує саме такий спосіб, якщо можна вибирати)	Пакунок та його залежності упаковані в один файл
Підтримує універсальні пакунки Linux (тип .snap)	Тільки для визначеного типу дистрибутивів Linux (тип .deb)
Автоматичне оновлення	Ручне оновлення

Більший розмір інсталяції	Менший розмір інсталяції: залежності розподілені і надають спільний доступ різним застосункам

Деякі застосунки можна встановлювати одночасно або з apt, або також зі snap.

- Додавання адреси репозиторію до офіційного списку репозиторіїв (add-apt-repository). Адреса додається у файл /etc/apt/sources.list. Далі можна встановити застосунок як звичайно: `apt install [package_name]`
- Personal Package Archive. Розробники застосунків можуть розповсюджувати програмне забезпечення, використовуючи персональний архів пакетів (PPA). Це означає, що користувачі Ubuntu можуть встановлювати та оновлювати пакунки сторонніх виробників так само, як вони встановлюють стандартні пакунки Ubuntu. Досить часто PPA використовуються розробниками для швидких оновлень в порівнянні з офіційними репозиторіями Ubuntu.

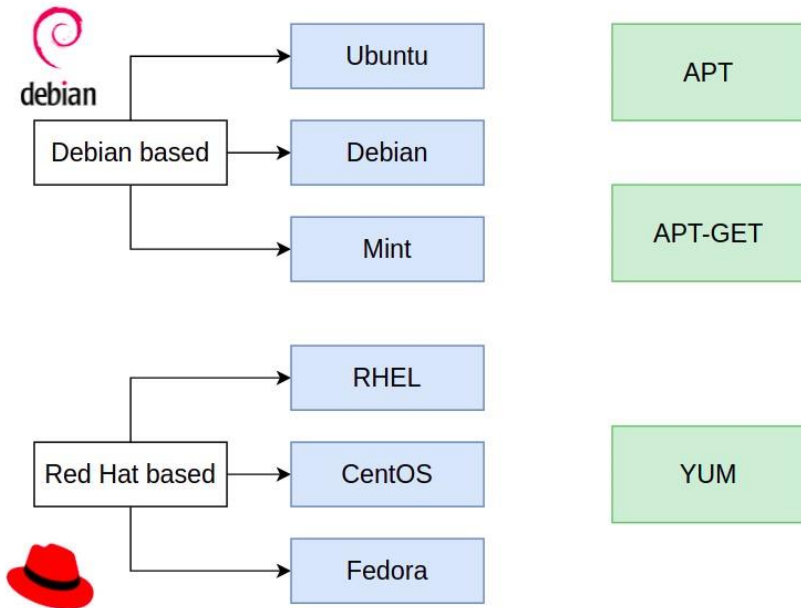
Таким чином, ми розглянули поширені способи установки застосунків: Apt Package Manager (віддаємо перевагу), Snap Package Manager, додавання персонального репозиторію, Ubuntu Software Center.

В випадку інших дистрибутивів Linux, установка і оновлення застосунків має схожі принципи. Пакункові менеджери:

- використовують офіційні репозиторії;
- завантажують пакунки, встановлюють залежності тощо;

- вирішують можливі різниці в версіях або в залежностях пакунків.

Менеджер пакунків часто стає ключовим при вирішенні, який дистрибутив Linux виберуть системні адміністратори для використання.



Таким чином, ми розглянули поняття менеджера пакунків, особливості установки застосунків з використання менеджера пакунків, порівняння менеджерів apt, apt-get, snap, приклади установки застосунків на Ubuntu, альтернативи для інших дистрибутивів Linux.

2.6 Текстовий редактор VIM

Для редагування файлу Linux має вбудований в термінал редактор Vi або його розширену сучасну версію Vim. Vim найбільш розповсюджений та вживаний редактор в Linux. В залежності від дистрибутиву Vi може і не бути встановленим.

Vim підтримує багато форматів та використовується для:

- створення невеликих текстових файлів;
- швидких редагувань, особливо під час роботи в терміналі (конфігурація, змінні середовища, логування);
- при роботі з віддаленим сервером (Git CLI при написанні повідомлення для коміту, перегляд конфігураційного файлу Kubernetes).

Приклад роботи з Vim

Крок 1. Якщо Vim не встановлений, виконуємо установку:

```
sudo apt install vim
```

Крок 2. Відкрити або створити файл:

```
vim [filename]
```

По замовчуванню редактор відкривається в режимі Command Mode:

- не можна редагувати текст
- введений текст інтерпретується як команда
- доступна навігація по файлу, пошук, видалення, відміна

Крок 3. Для переходу в пежим Insert Mode (можливість введення/редагування тексту), натиснемо клавішу *i*.

Для переключення між режимами, збереження і вихід використовуємо такі команди:

- `i` - перехід в режим Insert Mode

- *Esc* - переключення в Command Mode
- `:wq` – write, quit – зберегти та вийти
- `:q!` – вийти без збереження змін

Для користуванням редактором в режимі Command Mode наведемо деякі корисні команди Vim

Видалення рядків:

- `dd` – видалення поточного рядка
- `d10d` – видалення заданої кількості рядків з поточного *Відміна операцій:*
- `u` - скасування останньої операції в історії змін *Навігація:*
- `A` - перейти в кінець файлу з переключенням в режим Insert Mode

- `0` - перейти на початок поточного рядка
- `$` - перейти на кінець рядка без переключення в Insert Mode
- `12G` - перейти на рядок з заданим номером *Пошук та заміна:*
- `/pattern` – пошук заданого рядка в файлі (команда `n` – наступне входження, `N` - попереднє входження;
- `:%s/nginx/web-app` – пошук з заміною

2.7 Користувачі та дозволи

Важливим компонентом безпеки операційної системи та серверу є керування користувачами. Linux володіє простими та ефективними засобами і методами керування обліковими записами, які потрібно розуміти та використовувати.

Облікові записи в Linux є трьох типів: супер-користувач (root), стандартні користувачі, користувачі сервісів.

Супер-користувач (root) – єдиний в системі обліковий запис, що має необмежені дозволи. Потребує логін як *root* або виконання команд як *root* з ключовим словом *sudo*.

Розробниками було прийнято рішення вимкнути обліковий запис *root* за замовчуванням у всіх інсталяціях Ubuntu. Це не означає, що кореневий обліковий запис було видалено або що до нього немає доступу. Йому надано хеш пароля, який не відповідає жодному можливому значенню, тому по користувач *root* не може по замовчуванню ввійти в систему.

Для виконання команд під супер-користувачем пропонується скористатися інструментом *sudo* (super user do). Sudo дозволяє авторизованому користувачеві для даної команди підвищити свої привілеї. Ця методологія забезпечує звітність за всі дії користувача та надає адміністратору детальний контроль над

тим, які дії користувач може виконувати із зазначеними привілеями.(sudo -i, sudo bash)

Стандартні користувачі. Такі облікові записи можна створювати для подальшого користування системою.

- *Користувачі сервісів*. Відносяться до серверних служб Linux: сервер баз даних, веб-сервер. Кожен сервіс має свого користувача: наприклад, користувач MySQL або Apache. Таким чином, ми ізолюємо кожен застосунок один від одного, забезпечуючи належний рівень захисту. Зауважимо, що не можна запускати сервіс під root-користувачем: сервіс може здійснити непередбачувані зміни та пошкодити систему і дані.

Існують певні відмінності при роботі з обліковими записами в операційних системах Windows та Linux:

Багато компаній та університетів використовують саме Windows для своїх працівників. В Windows користувач може залогінитись з будь-якого комп’ютера локальної мережі, оскільки дана система має функціональну можливість централізовано керувати користувачами. При вході користувача, Windows запитує аутентифікацію в центрального сервера, і доступ надається лише до власного домашнього каталогу. Користувач не має можливості змінювати щось за межами домашньої папки.

Linux не має можливості централізовано керувати користувачами. Облікові записи створюються і логуються безпосередньо на робочому комп’ютері. Робота з багатьма обліковими записами є типовою для серверів. Наприклад, кожен член команди може мати власний обліковий запис на сервері, перевагами чого є:

- команда не потребує root-доступу;

- для кожного користувача можна задати свою систему дозволів;
- можна відслідковувати журнали логування кожного користувача: хто, що і коли робив в системі.

Групи та дозволи

Керувати дозволами можна на двох рівнях:

- *Рівень користувача*. Передбачає безпосереднє керування дозволами для конкретного користувача.
- *Групування користувачів*: створення групи, додавання користувачів до групи, керування дозволами для групи. Приклади імен для груп: *devOpsGroup*, *adminGroup*, *devGroup.*

Файл */etc/passwd* зберігає дані про облікові записи. Кожен користувач може відкрити файл, проте лише супер-користувач може змінювати його. Записи в цьому файлі мають вигляд:

```
alexk:x:1001:1001:AlexK,,,:/home/alexk:/bin/bash
```

- *alexk* – логін для входу;
- *x* - заміщувач зашифрованого паролю, що зберігається в файлі /etc/shadow;
- *1001* – UID (User ID). Кожен користувач має свій унікальний ID. UID із значенням *0* зарезервовано для супер-користувача;
- *1001* - GID (Group ID). Головна група користувача. Групи зберігаються в файлі */etc/group*;
- *AlexK* – описові дані користувача;
- Наступні декілька параметрів, розділених “;” – опис та додаткова інформація про користувача;
- */home/alexk* – домашня директорія користувача;

- */bin/bash* – командна оболонка для користувача по замовчуванню;

Керування користувачами

Інформація про користувачів, групи, паролі міститься у відповідних файлах, які не слід редагувати вручну з текстового редактора. Для керування користувачами доцільно використовувати команди:

- `sudo adduser <username>` – додати нового користувача. При цьому автоматично присвоюються UID та GID. Створюється домашня директорія з базовою файловою структурою. При створенні користувача, для нього також створюється його основна група.
- `sudo passwd <username>` – змінити пароль користувача.
- `su - <username>` – switch user – залогінитись (перемикнутись) під даним користувачем. Потрібно ввести пароль користувача.
- `exit` – розлогінитись з даного користувача.
- `su -` – залогінитись під супер-користувачем. Зауважимо, що обліковий запис *root* по замовчуванню недоступний в Ubuntu. Таким чином, дана команда не виконається. *Керування групами*
- `sudo groupadd <groupname>` – створити групу

Як правило, в більшості дистрибутивів Linux є можливість створювати користувачів та групи парами команд: `useradd` та `adduser`, `groupadd` та `addgroup`.

- `adduser`, `addgroup` – дані команди більш інтерактивніші та дружелюбні: автоматичне отримання ідентифікаторів UID та GID, створення домашньої директорії з базовою

конфігурацією, введення даних відбувається в інтерактивному режимі.

- `useradd`, `groupadd` – дані команди передбачають передачу інформації через параметри. Вони є більш низького рівня, використовуються в shell-скриптах при автоматизації.
- аналогічно працюють команди для видалення користувачів та груп: `deluser`, `userdel`, `delgroup`, `groupdel`.

Додавання до груп та видалення з груп користувачів

- `sudo usermod -g <groupname> <username>` – зміна основної групи користувача.
- `sudo usermod -G <groupname1>,<groupname2> <username>` – додавання користувача до вторинних груп.
- `sudo usermod -aG <groupname1> <username>` – додавання користувача *додатково* до вторинних груп.
- `groups` – виводить групи, до яких належить поточний користувач.
- `groups <username>` – виводить групи заданого користувача.
- `sudo useradd -G <groupname> <username>` – створить користувача з одночасним додаванням його до заданої вторинної групи. При цьому первинною групою для користувача створиться однойменна група з іменем користувача.
- `sudo gpasswd -d <username> <groupname>` – видалення користувача з групи.

```
alexk@alexk-Vostro-15-3510:~$ sudo useradd -G devops nicole
[sudo] password for alexk:
alexk@alexk-Vostro-15-3510:~$ groups nicole
nicole : nicole devops
```

```
alexk@alexk-Vostro-15-3510:~$ sudo gpasswd -d nicole devops
Removing user nicole from group devops
```

Керування власністю та дозволами

Для виводу вмісту поточної папки з інформацією про дозволи та власність виконуємо команду `ls -l` – довгий (long) формат виводу вмісту поточної папки:

```
alexk@alexk-Vostro-15-3510:~$ ls -l
total 52
drwxr-xr-x  2 alexk devops 4096 жов  6 09:34  Desktop
drwxr-xr-x  4 alexk devops 4096 бер 24 14:49  Documents
drwxr-xr-x  6 alexk devops 4096 бер 27 14:03  Downloads
-rw-rw-r--  1 alexk devops    0 бер 29 15:00  file.txt
drwxrwxr-x  9 alexk devops 4096 гру  8 23:26  google-cloud-sdk
```

Керування власністю

Власність (Ownership) означає відомості, про користувачів, що володіють даним файлом (хто має всі дозволи на файл).

Кожен файл має *два рівні власності*:

- Користувач як власник файлу. Власність надається як правило тому, хто створив файл.
- Група як власник файлу. По замовчуванню, це первинна група користувача-власника.

```
drwxr-xr-x  2 alexk devops 4096 жов  6 09:34  Desktop
drwxr-xr-x  4 alexk devops 4096 бер 24 14:49  Documents
drwxr-xr-x  6 alexk devops 4096 бер 27 14:03  Downloads
```

Для *зміни власника* файлу або папки виконуються команди:

- `sudo chown <username>:<group> <filename>` – зміна користувача та групи як власників файлу

- `sudo chown <username> <filename>` – зміна лише користувача як власника файлу
- `sudo chgrp <group> <filename>` – зміна лише групи як власника файлу

Керування дозволами

Дозволи відносяться до читання, запису та виконання файлів. Дозволи на файл виводяться у рядку з 10 символів, наприклад "*drwxr-xr-x*":

```
drwxr-xr-x  2 alexk  devops 4096 жов  6 09:34  Desktop
drwxr-xr-x  4 alexk  devops 4096 бер 24 14:49  Documents
drwxr-xr-x  6 alexk  devops 4096 бер 27 14:03  Downloads
-rw-rw-r--  1 nicole devops    0 бер 29 15:00  file.txt
```

- Перший символ: `d` – директорія, `-` – регулярний файл, `l` – симлінк, `c` – файли для зовнішніх пристроїв тощо;
- Далі йде перша група з трьох символів `rwx` – дозволи для користувача, який є власником файлу: `r` – дозвіл на читання, `w` – дозвіл на запис, `x` – дозвіл на виконання, `-` – немає дозволу відповідно на читання/запис/виконання.
- Друга група символів – дозволи для групи, яка є власником файлу.
- Третій блок – інші користувачі або користувачі інших груп, які не є власниками файлу.

Для керування дозволами на файл можна використати три варіанти

- додавання/видалення одного дозволу (+/-);
- задання/зміни групи дозволів (=);
- задання/зміни одночасно всіх дозволів з використанням числових значень.

Приклади команд для *додавання або видалення одного дозволу на файл*:

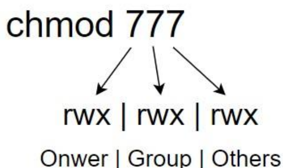
- `sudo chmod -x <filename>` – видалити дозвіл на виконання для всіх користувачів
- `sudo chmod g-w <filename>` – видалити дозвіл на запис для групи як власника файлу
- `sudo chmod g+x <filename>` – додати дозвіл на виконання для групи як власника файлу
- `sudo chmod u+x <filename>` – додати дозвіл на виконання для юзера як власника файлу
- `sudo chmod o+x <filename>` – додати дозвіл на виконання для інших користувачів, які не є власниками файлу

Приклад команди для *множинної зміни дозволів на файл*

- `sudo chmod g=rwx <filename>` – приклад зміни дозволів для групи (`u` – для юзера, `o` – для інших користувачів, `a` – для всіх користувачів)

Приклад команди для множинної зміни дозволів з використанням числових значень:

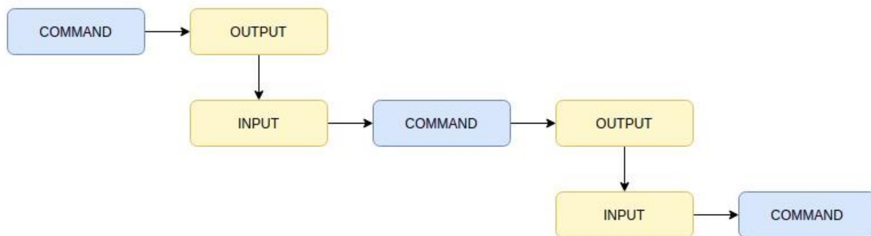
- `sudo chmod 777 <filename>` – кожна цифра відповідає групі дозволів для користувача-власника, групи-власника та всіх інших користувачів за такою відповідністю:



Десяткова форма	Рядкове представлення	Двійкова форма
0	---	000
1	--x	001
2	-w-	010
3	-wx	011
4	r--	100
5	r-x	101
6	rw-	110
7	rwx	111

2.8 Командні ланцюжки та перенаправлення виводу

Вивід з однієї команди може стати введенням для іншої команди. Передача параметрів з виводу на наступне введення називається *командним ланцюжком*:



Прикладом командного ланцюжка може бути передача виводу в *less*. *less* — це команда Linux, яка використовується для фільтрації та перегляду текстових даних на одній сторінці за екраном.

`cat /var/log/syslog | less` – читаємо дані з великого файлу і передаємо вміст файлу в *less*, який виведе їх з посторінковим розбиттям . При цьому дані всього файлу не виводяться одразу, а лише передаються в наступну команду.

З *less* можна здійснювати перегляд і навігацію по довгим текстовим файлам сегментами без необхідності завантажувати весь файл. Для навігації в *less* використовується *пробіл* та клавіша *b*. Також можна скролити. Для виходу натискаємо *q*.

Інші приклади командних ланцюжків:

- `ls /usr/bin | less`
- `history | less`

Використання *grep* для фільтрації виводу даних. *grep* - Globally Search for Regular Expression and Print out – інструмент для

пошуку за регулярним виразом та виводу всіх рядків, що містить даний вираз:

- `history | grep sudo`
- `history | grep “sudo chmod”`
- `history | grep sudo | less`
- `cat config.yaml | grep 80`
- `ls /usr/bin | grep python` ● `ls /usr/bin | grep php`

```
alexk@alexk-Vostro-15-3510:/$ history | grep "sudo chmod"
 1464  sudo chmod 777 models
 1465  sudo chmod 777 controllers
 1466  sudo chmod 777 views
 1484  sudo chmod 777 views
 1486  sudo chmod 777 product
 1507  sudo chmod 777 node_modules
 1676  sudo chmod 777 sandbox
 2016  history | grep "sudo chmod"
```

Перенаправлення виводу

Символ `>` – оператор перенаправлення – зберігає вивід у файл.

Символ `>>` – оператор перенаправлення – приєднує нові рядки до наявних у файлі.

Приклад виводу попередньої команди, що передається в поточну та зберігається у файл:

`history | grep sudo > sudo-commands.txt`

`cat sudo-commands.txt > sudo-rm-commands.txt` – створить копію файлу

`history | grep rm >> sudo-rm-commands.txt` – приєднає нові рядки до існуючих

Наведіть приклад пошуку в логах та збереження результатів у файл

Стандартне введення та вивід

Кожен застосунок має 3 вбудованих потоки:

- STDIN(0) – стандартне введення;
- STDOUT(1) – стандартний вивід;
- STDERR(2) – стандартна помилка.

Якщо при виводі спрацьовує помилка, вона не передається далі по ланцюжку команд.

Для задання послідовності незалежних команд використовується розділювач “;”: `clear; sleep 1; echo "Hello, World!"`

2.9 Написання bash-скриптів, синтаксис bash-скриптів

В загальному, клас комп’ютерних програм, які приймають команди, інтерпретують їх і передають операційній системі для обробки називається командними оболонками (*shell*).

Для отримання всіх shell-інтерпретаторів, що встановлені на нашій ОС, можна відкрити файл */etc/shells*:

```
cat /etc/shells
```

В системах Unix та Linux UNIX існує декілька різних оболонок:

- *sh*, або оболонка Борна (Bourne Shell) – одна з перших оболонок, яка була використана у UNIX-подібних середовищах. Це базова оболонка, маленька та з невеликим набором можливостей. Вона є де-факто стандартною оболонкою, та присутня на кожній системі із UNIX. На Лінукс /bin/sh може бути символьним лінком на bash. Це зроблено для того, щоб забезпечити сумісність з програмами UNIX.
- *bash* (Bourne again shell), або нова оболонка Борна (Bourne Again Shell) – стандартна оболонка Linux. У більшості користувачів Linux стандартною оболонкою

встановлено саме bash. У деякому сенсі bash — це розширена та вдосконалена надбудова над sh, набір доповнень та додаткових модулів. Таким чином, оболонка bash сумісна зі звичайною оболонкою sh: команди, що працюють у sh, будуть працювати і у bash, але не обов'язково навпаки.

Bash-скрипти — це потужний і універсальний інструмент для автоматизації завдань системного адміністрування, керування системними ресурсами та виконання інших рутинних завдань у системах Unix/Linux. Наведемо деякі переваги використання bash-скриптів:

Автоматизація. bash-скрипти дозволяють автоматизувати повторювані завдання та процеси, заощаджуючи час і знижуючи ризик помилок, які можуть виникнути під час ручного виконання.

Портативність. bash-скрипти можна запускати на різних платформах і операційних системах, включаючи Unix, Linux, macOS і навіть Windows за допомогою емуляторів або віртуальних машин.

Гнучкість. bash-скрипти легко налаштовуються та можуть бути легко змінені відповідно до конкретних вимог. Їх також можна комбінувати з іншими мовами програмування або утилітами для створення більш потужних сценаріїв.

Доступність. bash-скрипти легко створювати без потреби спеціальних інструментів чи програмного забезпечення. Їх можна редагувати за допомогою будь-якого текстового редактора, а більшість операційних систем мають вбудований інтерпретатор.

Інтеграція. bash-скрипти можна інтегрувати з іншими інструментами та програмами, такими як бази даних, веб-сервери та хмарні служби, що дозволяє виконувати більш складні завдання автоматизації та керування системою.

Налагодження. сценарії оболонки легко налагоджувати, і більшість оболонок мають вбудовані засоби налагодження та звітування про помилки, які можуть допомогти швидко виявити та виправити проблеми.

По суті, bash-скрипт — це файл з розширенням *.sh*, що містить послідовність команд, які виконуються bash-програмою послідовно рядок за рядком. Він дозволяє виконувати серію дій, наприклад, перехід до певного каталогу, створення папки та запуск процесу за допомогою командного рядка.

Наприклад, нам потрібно сконфігурувати сервер, виконавши послідовність команд:

```
useradd <username>
groupadd <groupadd>
mkdir <folder> touch
<filename> chmod 750
/path sudo apt install
docker docker run …
```

Зберігаючи ці команди в сценарії, їх виконання можна повторювати багаторазово або на декільках серверах, запускаючи відповідний сценарій. Також адміністраторам не потрібно пам'ятати конфігурацію, уникати рутинної роботи, здійснювати масові операції, надавати доступ до сценаріїв іншим адміністраторам. Часто при розгортанні потрібно реалізовувати деяку логіку: перевіряти дозволи або чи існує файл, або перевіряти версію тощо.

Простий приклад створення і запуску bash-скрипта

- Створимо і перейдемо в папку *projects*
- Створимо файл *demo.sh* `touch demo.sh`

- Відкриємо його в текстовому редакторі і перейдемо в режим редагування:

`vim demo.sh`

`i`

- В першому рядку вкажемо шлях до інтерпретатора: *#!/bin/bash*
- Додамо команду з виводом тексту: *echo “Setup and configure server”*
- Збережемо і закриємо файл: Esc → `:wq`
- Додамо дозвіл на виконання для користувача-власника:

`sudo chmod u+x demo.sh`

- Виконаємо скрипт:

  `./demo.sh` (або `bash demo.sh`)

Результат виконання наведемо на рисунку:

```
#!/bin/bash

echo "Setup and configure server"
~
```

```
alexk@alexk-Vostro-15-3510:~$ touch demo.sh
alexk@alexk-Vostro-15-3510:~$ vim demo.sh
alexk@alexk-Vostro-15-3510:~$ ls -l demo.sh
-rw-r--r-- 1 alexk sudo 47 бер 30 16:12 demo.sh
alexk@alexk-Vostro-15-3510:~$ sudo chmod u+x demo.sh
[sudo] password for alexk:
alexk@alexk-Vostro-15-3510:~$ ./demo.sh
Setup and configure server
alexk@alexk-Vostro-15-3510:~$
```

Змінні в bash-скриптах

Змінні використовуються для збереження даних і подальшого їх використанняподібно до змінних в мовах програмування загального призначення. Для створення змінної потрібно вказати її ім’я та присвоїти значення:

```
variable_name=value
```

Зауважимо, навколо знаку привоєння “=” не повинно бути пробілів. Приклад створення і використання змінних:

```
#!/bin/bash

echo "Setup and configure server"
#  Створюємо змінну file_name
file_name=config.yaml #
Створюємо змінну config_files
config_files=$(ls config)
echo "This is config file: $file_name" echo
"These are config files: $config_files"
```

Результат виконання

```
Setup and configure server
ls: cannot access 'config': No such file or directory
This is config file: config.yaml
These are config files:
alexk@alexk-Vostro-15-3510:~/projects$
```

Умовні конструкції

Умовні конструкції дозволяють розділяти керування потоком: виконують команду або послідовність команд, якщо деяка умова істинна, інакше можна виконати іншу послідовність команд:

```
if [condition]
then
  [statement]
elif [condition]
then
  [statement]
else
  [statement]
fi
```

```
#!/bin/bash

echo "Setup and configure server"

if [ -d config ]
then
        echo "Reading the content"
        config_files=$(ls config)
        echo "Config files are: $config_files"
else
        echo "Directory not found. Let's create one"
        mkdir config
fi
```

Результат виконання:

```
alexk@alexk-Vostro-15-3510:~/projects$ ./conditional.sh
Setup and configure server
Directory not found. Let's create one
alexk@alexk-Vostro-15-3510:~/projects$ ./conditional.sh
Setup and configure server
Reading the content
Config files are:
alexk@alexk-Vostro-15-3510:~/projects$
```

Типові логічні вирази при перевірках файлів

https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html#sect_07_01_01_01

Логічний вираз	Значення
[-a FILE]	True if FILE exists.
[-b FILE]	True if FILE exists and is a block-special file.
[-c FILE]	True if FILE exists and is a character-special file.
[-d FILE]	True if FILE exists and is a directory.
[-e FILE]	True if FILE exists.
[-f FILE]	True if FILE exists and is a regular file.
[-g FILE]	True if FILE exists and its SGID bit is set.
[-h FILE]	True if FILE exists and is a symbolic link.
[-k FILE]	True if FILE exists and its sticky bit is set.
[-p FILE]	True if FILE exists and is a named pipe (FIFO).
[-r FILE]	True if FILE exists and is readable.
[-s FILE]	True if FILE exists and has a size greater than zero.
[-t FD]	True if file descriptor FD is open and refers to a terminal.
[-u FILE]	True if FILE exists and its SUID (set user ID) bit is set.
[-w FILE]	True if FILE exists and is writable.

[-x FILE]	True if FILE exists and is executable.
[-O FILE]	True if FILE exists and is owned by the effective user ID.
[-G FILE]	True if FILE exists and is owned by the effective group ID.
[-L FILE]	True if FILE exists and is a symbolic link.
[-N FILE]	True if FILE exists and has been modified since it was last read.
[-S FILE]	True if FILE exists and is a socket.

Оператори відношення для числових значень Варіанти використання таких операторів:

- Перевірити значення порту; ● Контролювати кількість процесів;
- Враховувати кількість файлів тощо.

Оператор	**Опис**
arg1 -eq arg2	True if arg1 equals arg2
arg1 -ne arg2	True if arg1 is not equal to arg2
arg1 -lt arg2	True if arg1 is less than arg2
arg1 -le arg2	True if arg1 is less than or equal to arg2
arg1 -gt arg2	True if arg1 is greater than arg2
arg1 -ge arg2	True if arg1 is greater than or equal to arg2

Оператори відношення для рядків

Оператор	**Опис**
[-z STRING]	True of the length if "STRING" is zero.
[-n STRING] or [STRING]	True if the length of "STRING" is non-zero.
[STRING1 == STRING2]	True if the strings are equal. "=" may be used instead of "==" for strict POSIX compliance.
[STRING1 != STRING2]	True if the strings are not equal.
[STRING1 < STRING2]	True if "STRING1" sorts before "STRING2" lexicographically in the current locale.
[STRING1 > STRING2]	True if "STRING1" sorts after "STRING2" lexicographically in the current locale.

Приклади використання операторів відношень для рядків:

```
if [ "$user_group" == "alexk" ]
then
  echo "Configure the server"
else
  echo "No permission to configure the server"
fi
```

```
user_group=alexk

if [ "$user_group" == "alexk" ]
then
  echo "Configure the server"
elif [ "$user_group" == "admin" ]
then
  echo "Administer ther server"
else
  echo "No permission to configure the server"
fi
```

Передача аргументів в скрипт

В скрип аргументи можна передавати через позиційні параметри, тобто дані, що передаються в скрипт обробляються в тому ж порядку, в якому вони надсилаються. Індексація аргументів починається з 1.

```
config_dir=$1

if [ -d "$config_dir" ]
then
    echo "Reading the content of config directory"
    config_files=$(ls "$config_dir")
else
    echo "Config directory not found. Creating one"
    mkdir "$config_dir"
    touch "$config_dir/config.sh"
fi
```

Результат виконання скрипта з позиційним параметром

```
alexk@alexk-Vostro-15-3510:~$ ./demo.sh myFolder
Setup and configure server
Config directory not found. Creating one
No permission to configure the server
using config.yaml to configure something
here are all configuration files:
alexk@alexk-Vostro-15-3510:~$ ls
 config     Documents   google-cloud-sdk   myFolder         Public      test.txt
 demo.sh    Downloads   Music              Pictures         snap        Videos
 Desktop    file.txt    my                'Postman Agent'   Templates   WebstormProjects
alexk@alexk-Vostro-15-3510:~$ cd myFolder
alexk@alexk-Vostro-15-3510:~/myFolder$ ls
config.sh
alexk@alexk-Vostro-15-3510:~/myFolder$
```

Введення даних під час виконання скрипта

Для реалізації можливості введення даних під час виконання скрипта використовується команда read:

```
#!/bin/bash

echo "Reading user input"
read -p "Please enter the password: " user_pwd
echo "Thanks for your user password $user_pwd"
```

```
alexk@alexk-Vostro-15-3510:~$ ./demo.sh
Reading user input
Please enter the password: welcome
Thanks for your user password welcome
alexk@alexk-Vostro-15-3510:~$
```

Для отримання всіх вхідних параметрів одночасно, можна використати такі вказівники:

- `$*` – представляє всі аргументи в рядок.
- `$#` – кількість параметрів

```
#!/bin/bash

echo "All parameters: $*"
echo "Number of parameters: $#"
echo "Username: $1"
echo "Groupname: $2"
echo "configFolder: $3"
echo "configFile: $4"
```

```
alexk@alexk-Vostro-15-3510:~$ ./demo.sh alexk devops config setup.sh
All parameters: alexk devops config setup.sh
Number of parameters: 4
Username: alexk
Groupname: devops
configFolder: config
configFile: setup.sh
```

Циклічні конструкції в Bash-скриптах

Bash володіє такими циклічними операторами:

- for
- while
- until
- select

Цикл *for* – оперує зі списками елементів, а саме повторює набір команд для кожного елемента зі списку:

```
for var in <list>
do <commands>
done
```

Приклад циклу *for* з виводом всіх параметрів, що передаються в скрипт:

```
#!/bin/bash

echo "All parameters: $*"

for param in $*
 do
   echo "$param"
 done
```

Результат виконання скрипта:

```
alexk@alexk-Vostro-15-3510:~$ ./demo.sh
All parameters:
alexk@alexk-Vostro-15-3510:~$ ./demo.sh alexk devops config setup.sh
All parameters: alexk devops config setup.sh
alexk
devops
config
setup.sh
```

Приклад циклу *for* з вкладеним умовним оператором:

```
#!/bin/bash

echo "All parameters: $*"

for param in $*
 do
   echo $param
   if [ -d "$param" ]
   then
    echo "executing scripts in the config folder"
    ls -l "$param"
   fi
 done
```

Результат виконання скрипта:

```
alexk@alexk-Vostro-15-3510:~$ ./demo.sh alexk devops config setup.sh
All parameters: alexk devops config setup.sh
alexk
devops
config
executing scripts in the config folder
total 0
setup.sh
```

Цикл *while* повторює виконання набору команд допоки виконується певна умова. Варіанти використання циклу *while:*

- пінгуємо сервіс поки доступний;
- перевіряємо протягом 10 секунд доступ до застосунку на сервері;
- надсилаємо команду для установки і чекаємо поки інсталяція не завершиться. `while [ <some test> ] do <commands> done`

Приклад використання циклу *while* з оператором *break*:

```
sum=0
while true
do
 read -p "Enter the number or q to quit: " score
 if [ "$score" == "q" ]
 then
  break
 fi
 sum=$sum+$score
 echo "Total score: $sum"
done
```

Результат виконання скрипта:

```
Enter the number or q to quit: 1
Total score: 0+1
Enter the number or q to quit: 2
Total score: 0+1+2
Enter the number or q to quit: e
Total score: 0+1+2+e
Enter the number or q to quit: q
```

Приклад циклу *while* з використанням арифметичних операцій використовуємо `$(())`:

```
sum=0
while true
do
 read -p "Enter the number or q to quit: " score
 if [ "$score" == "q" ]
 then
  break
 fi
 sum=$(($sum+$score))
 echo "Total score: $sum"
done
```

Для Bash-скриптів в умовах *if* можна використовувати подвійні прямокутні дужки `[[]]`. Такий синтаксис має деякі покращення. Наприклад, змінні можна не записувати в лапках[1].

Функції в Bash-скриптах

Функції дозволяють розділити скрипт на окремі логічно завершені блоки для їх багаторазового повторного використання.

```
function_name () {
   <commands>
}
```

Приклад 1. Наведемо приклад створення і багаторазового виклику простої функції для виводу рядка:

---

[1] https://stackoverflow.com/questions/3427872/whats-the-difference-b etween-and-in-bash

```
#!/bin/bash

function hello_world {
  echo "Hello, World"
}

hello_world
hello_world
hello_world
```

Результат виконання буде виглядати наступним чином:

```
alexk@alexk-Vostro-15-3510:~$ ./demo_func.sh
Hello, World
Hello, World
Hello, World
alexk@alexk-Vostro-15-3510:~$
```

Приклад 2. Приклад передачі параметрів в функцію

```
function create_file() {
        file_name=$1
        touch "$file_name"
        echo "File $file_name is created"
}

create_file myDemoFile.txt
create_file config.yaml
create_file setup.sh
```

Результат виконання:

```
alexk@alexk-Vostro-15-3510:~$ ./demo_func.sh
Hello, World
Hello, World
Hello, World
File myDemoFile.txt is created
File config.yaml is created
File setup.sh is created
```

Приклад 3. Передача декількох параметрів в функцію

```
function create_file() {
        file_name=$1
        is_shell_script=$2
        touch "$file_name"
        echo "File $file_name is created"
        if [ "$is_shell_script" = true ]
        then
          chmod u+x $file_name
          echo "Added execute permission"
        fi
}

create_file myDemoFile.txt
create_file config.yaml
create_file setup.sh true
```

Результат виконання скрипта:

```
File myDemoFile.txt is created
File config.yaml is created
File setup.sh is created
Added execute permission
```

Приклад 4. Повернення значень з функцій:

```
function sum() {
        total=$(($1+$2))
        return $total
}

sum 1 2
result=$?

echo "The sum of 1 and 2 is $result"
```

Кращі практики при використанні функцій

- Не використовуйте занадто багато параметрів
- Застосовуйте принцип єдиної відповідальності: функція повинна робити лише одну справу
- Оголошуйте змінні зі змістовним іменем для позиційних параметрів всередині функцій

Підсумовуючи, ми розглянули особливості та варіанти використання bash-скриптів: створення та виконання, змінні, умовні конструкції, цикли, функції. В якості варіантів автоматизації роботи можна навести скрипти для створення резервних копій, моніторингу, конфігурацій для сервера тощо.

Зауважимо, що Bash-скрипти мають складний синтаксис, дещо схожий на синтаксис звичайних мов програмування, проте інколи не зовсім інтуїтивно зрозумілий. Кращими альтернативами є використання Python або інструментів конфігурації, наприклад Ansible. Вміння використовувати дані технології робить Devops-інженера досить цінним працівником.

2.10 Змінні середовища

Змінна середовища — це змінна, значення якої встановлюється поза програмним кодом, як правило, за допомогою функцій, вбудованих в операційну систему або службу. Змінна середовища складається з пари ім'я/значення, призначена для зберігання деякого значення і може бути створена та доступна для посилання в програмному коді, команді або скрипту командної оболонки.

Основним варіантом використання змінних середовища є обмеження необхідності модифікації програмного коду через зміни в конфігураційних даних.

Варто зазначити, що операційна система може мати декілька різних середовищ. Одним із таких середовищ є обліковий запис користувача, який може конфігуруватись власником, наприклад, задавати налаштування та застосунки по замовчуванню: шрифт, командна оболонка, текстовий редактор, браузер. Такі конфігурації повинні бути ізольованими від інших середовищ.

Задання значень змінним середовища

Змінним середовища можна задавати для них значення в терміналі: `VAR_NAME=Value`

Наприклад:

- `SHELL=/bin/bash`
- `HOME=/home/alexk`
- `USER=alexk`

При використанні такого виразу для змінної, якої не існує, термінал створить *змінну оболонки*, яка схожа на змінну середовища, але не впливає на поведінку інших програм.

Створення змінних середовища

Для створення змінної середовища змінну оболонки можна експортувати за допомогою команди *export*. Наприклад:

- `export DB_USERNAME=dbuser`
- `export DB_PASSWORD=mysecretpwd`
- `export DB_NAME=mydb`

При цьому змінні середовища доступні *лише в даній сесії терміналу*.

Для *перегляду змінних середовища* можна використати команди:

- `printenv`
- `printenv | less`
- `printenv USER`
- `printenv | grep USER`
- `echo $USER` – посилання на змінні середовища (використовується в командах, в bash-скриптах)

Приклади виконання команд виводу змінних середовища:

```
alexk@alexk-Vostro-15-3510:~$ printenv | grep DB
DB_PASSWORD=secretpwdvalue
DB_USERNAME=dbuser
DB_NAME=mydb
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1001/bus
alexk@alexk-Vostro-15-3510:~$ echo $DB_NAME
mydb
```

Видалення змінних середовища

Для видалення змінних середовища використовуємо команду `unset <VAR_NAME>`, наприклад: • `unset DB_USERNAME`

Створення постійних змінних середовища для певного користувача

Для створення постійних змінних облікового запису їх необхідно експортувати в локальному файлі *.bashrc*

```
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

export DB_PASSWD=mydbpassword
export DB_NAME=mydb
export DB_USER=mydbuser
```

Для примінення змін виконаємо команду:

- `source .bashrc`

Змінна середовища PATH

Дана змінна повідомляє інтерпретатору командного рядка, в яких папках потрібно шукати виконавчі файли для виконання введених команд. Вона зберігає список папок з виконавчими файлами та визначена у файлі */etc/environment*:

```
alexk@alexk-Vostro-15-3510:~$ cat /etc/environment
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/u
sr/local/games:/snap/bin"
```

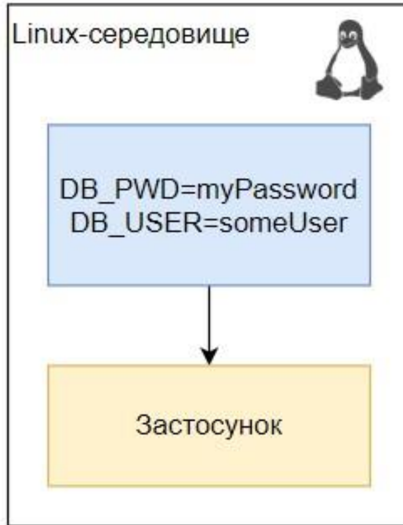
Таким чином, змінні середовища

- Складаються з пари *ім'я/значення*;
- Зберігають інформацію
- Доступні в різних місцях середовища
- Імена по стандарту задаються великими літерами
- Користувач може змінювати значення змінних середовища

Серед важливих варіантів використання змінних середовищ можна виділити також потребу застосунку в облікових даних для з'єднання з БД, секретному ключі для використання API. Використання таких даних в коді не дозволяється, тому доцільно

такі дані помістити в змінні середовища. А застосунок повинен мати до них доступ.

Передача змінних середовища в застосунок подібна до передачі параметрів в bash-скрипт. Кожна мова програмування має механізм доступу до змінних середовища.



Застосунок при цьому робиться більш гнучким: для різних середовищ змінні середовища можуть мати різні значення.

Практична робота 1.2. Створення власної команди в операційній системі Linux

- В домашній директорії створимо файл *welcome* з таким вмістом:

```
#!/bin/bash

echo "Welcome to our homeDirectory, $USER"
```

- Відкриємо файл *.bashrc* та додамо рядок для додавання шляху команди в змінну PATH:

```
PATH=$PATH:/home/alexk
```

- Застосуємо зміни: `source .bashrc`
- Виконаємо команду *welcome*, що запустить відповідний скрипт:

```
alexk@alexk-Vostro-15-3510:~$ welcome
Welcome to our homeDirectory, alexk
```

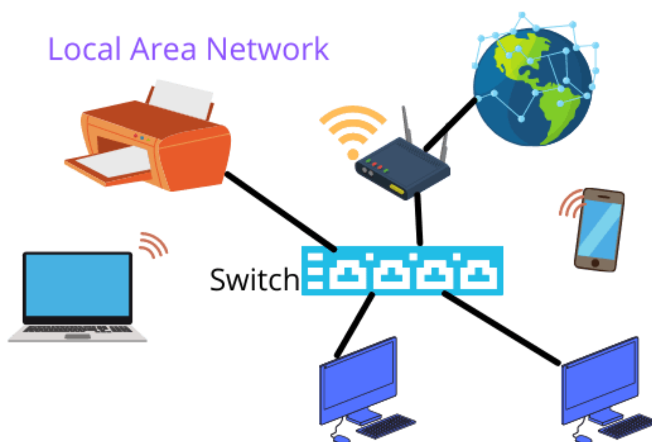
Розділ 3. Основи комп'ютерних мереж

3.1 Основи комп'ютерних мереж

В даному підрозділі розглянемо поняття локальної комп'ютерної мережі (LAN), ip-адреси, мережевого комутатора (switch), маршрутизатора (router), підмережі (subnet), перетворення мережевих адрес (NAT).

Локальна комп'ютерна мережа (LAN)

Локальна комп'ютерна мережа (Local Area Network, LAN) – локальна мережа – сукупність пристроїв, з'єднаних між собою в одній локації:



Пристрої, підключені до LAN, комунікують між собою через ір-адреси, причому кожен пристрій має свою власну унікальну ір-адресу.

IP-адреса

IP-адреса – адреса Інтернет-протоколу – це числова мітка, підключена до комп'ютерної мережі, яка використовує Інтернет-протокол для зв'язку. IP-адреса виконує дві основні функції: ідентифікацію мережевого інтерфейсу та адресацію розташування пристрою.

Інтернет-протокол версії 4 (IPv4) визначає IP-адресу як 32-розрядне число. Однак через розширення мережі Інтернет та зменшення доступних адрес IPv4 в 1998 році була стандартизована нова версія IP (IPv6), яка використовує 128 біт для IP-адреси. Розгортання IPv6 триває з середини 2000-х років.

IP-адреси записуються та відображаються у зрозумілих нотаціях, наприклад у 172.16.0.0 у версії IPv4.

Приклад IP-адреси для протоколу IPv4:

192.168.0.0

11000000.10101000.00000000.00000000

Як бачимо, IP-адреса (IPv4) являє собою 32-бітне значення, що групується на 4 октети, які можуть представлятись в т.ч. десятковим значенням. Діапазон кожного октету обмежений значеннями: 0000000 = 0

11111111 = 255

Т.ч. кожна ір-адреса належить інтервалу від 0.0.0.0 до 255.255.255.255.

Для керування адресацією і передачею даних в локальній мережі використовують пристрої – мережеві комутатори.

Мережевий комутатор (Switch)

Комутатор – це мережевий пристрій, який використовується з’єднання всіх пристроїв в локальній мережі, а також для сегментації мереж на різні підмережі, які називаються підмережами або сегментами локальної мережі. Він відповідає за фільтрацію та пересилання пакетів між сегментами локальної мережі на основі MAC-адреси.

Комутатори мають багато портів, і коли дані надходять на будь-який порт, спочатку перевіряється адреса призначення, також виконуються деякі перевірки, а потім вони обробляються на пристроях. Комутатор може підтримувати різні типи зв’язку: одноадресний, багатоадресний, широкомовний.

Маршрутизатор (Router)

Маршрутизатор – пристрій, що з’єднує локальну мережу з зовнішніми комп’ютерними мережами (WAN, Wide Area Network), а також надає доступ до мережі Internet. IP-адреса маршрутизатора називається *шлюзом* (*Gateway*).

Для визначення, чи знаходиться пристрій в LAN чи в WAN, розглянемо поняття підмережі.

Підмережа

Підмережа (Subnet) – логічний поділ мережі по сегментам IP-адрес. Для визначення, скільки біт зафіксовано для задання діапазону ip-адрес для однієї підмережі, використовується *маска підмережі*.

У конфігурації TCP/IP ми не можемо визначити, яка частина частина IP-адреси використовується як мережева, а яка належить до хост-адреси, якщо ми не отримаємо більше інформації з таблиці маски підмережі. *Маска підмережі* розділяє IP-адресу на мережеву та хостову адреси.

Нехай маскою підмережі є 255.255.255.0. IP-адресою пристрою є 192.168.123.132.

Співставивши двійкове представлення IP-адреси та маски підмережі разом, можна розділити мережеву та хостову частини адреси:

11000000.10101000.01111011.10000100

11111111.11111111.11111111.00000000

Таким чином перші 24 біти є фіксовані та ідентифікуються як мережева адреса, а останні 8 біт (нулі в масці підмережі) ідентифікуються як адреса хоста. Таким чином, можна зробити висновок, що

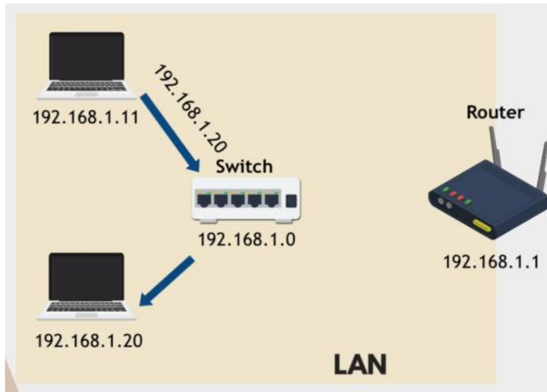
192.168.123.0 – мережева адреса

0.0.0.132 – хостова адреса

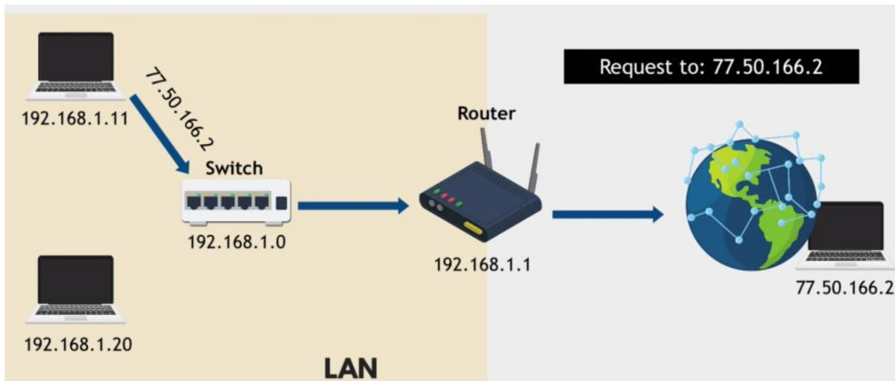
Отже, якщо пакет надходить у підмережу 192.168.123.0 і має адресу призначення 192.168.123.132, наш пристрій отримає його з мережі та обробить.

Інколи можна зустріти такий запис: 192.168.0.0/16. Тобто перші 16 біт визначають мережеву адресу. Даний метод для розподілу IP-адрес і IP-маршрутизації називається Classless Inter-Domain Routing (метод безкласової маршрутизації).

Запит на локальну ip-адресу:



Запит на зовнішню ip-адресу:



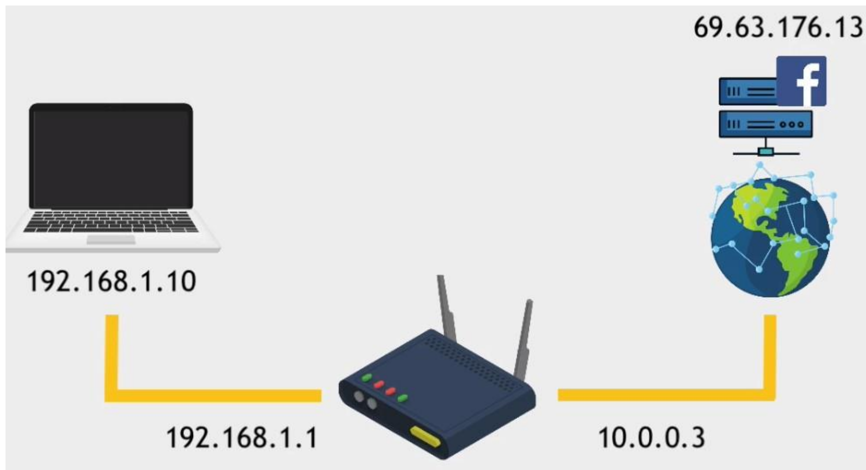
Отже, для обміну даними по мережі кожен пристрій повинен мати:

- ip-адресу
- маску підмережі
- інформацію про шлюз

Перетворення мережевих адрес (Network Gateway Translation, NAT)

В LAN діапазон IP-адрес задається адміністратором і кожен пристрій має свою унікальну IP-адресу. Нехай існує дві локальні мережі. Як пересвідчитись, що IP-адреси не конфліктують.

Справа в тому, що IP-адреси за межами LAN (в тому числі з мережі Internet) не доступні. Маршрутизатор, отримуючи пакет ззовні, здійснює трансляцію на локальну IP-адресу пристрою. Перетворення мережевих адрес є ключовою функцію маршрутизатора.



До переваг NAT віднесемо:

- безпеку та захист пристроїв в LAN
- повторне використання IP-адрес без конфліктів: різні LAN можуть мати один і той же діапазон локальних IP-адрес
- Всього кількість можливих публічних IP-адрес версії IPv4 – 4 294 967 296 (2^32), але пристроїв можна підключити набагато більше

3.2 Мережевий екран та мережеві порти

Мережевий екран (*firewall*) – система, що попереджує неавторизований доступ в приватну мережу. Через правила *мережевого екрану* можна визначити, які запити можна виконувати, а які – не дозволені.

Правила мережевого екрану визначають:

- яка ip-адреса в вашій мережі доступна;
- яка ip-адреса має доступ до сервера;
- або, наприклад, можна дозволити всім адресам мати доступ до сервера;
- які порти доступні на сервері.

Порт — це 16-розрядне числове значення в діапазоні від 0 до 65535. Порти можуть бути відкриті та використані програмними застосунками та службами операційної системи для надсилання та отримання даних через мережі (LAN або WAN), які використовують певні протоколи (наприклад, TCP, UDP). Порт асоціюється зі “входом” в певний “будинок”. Наприклад, у нашій повсякденній роботі ми використовуємо порт 80 для веб-запитів на основі HTTP та 443 для зашифрованих веб-сайтів на основі HTTPS.

Різні застосунки слухають на своїх визначених портах, але для багатьох застосунків можна визначити стандартні порти (по замовчуванню).

Ми можемо відкрити певні порти для загального доступу, а також певні порти для певних IP-адрес (гостей). Конфігурація мережевого екрану передбачає, які порти для яких користувачів можна відкрити, а також для яких запитів. Така конфігурація називається Port Forwarding (Port Mapping).

Підсумовуючи, порт є логічною формою для ідентифікації системних дій або різних мережевих служб, які використовуються для створення локальних або мережевих комунікацій.
Більшість веб-серверів слухають порт 80 (по замовчуванню)

Firewall сконфігурований з відкритим портом 80 для доступу до веб-застосунків з браузерів

3.3 Команди для роботи з мережею

Існує ряд команд для виводу інформації та конфігурування мережевих параметрів. Наведемо деякі типові команди та утиліти.

`ifconfig` (*interface configuration*). Дана утиліта в Unix та подібних системах використовується для налаштування та виведення параметрів мережевого інтерфейсу. Її можна використовувати, щоб призначити адресу мережевому інтерфейсу, налаштувати або переглянути інформацію про його конфігурацію: ip-адреса, маска підмережі, адреса шлюзу.

`iproute2` та її компонент `ip` – сучасна і оновлена альтернатива ifconfig.

`netstat` – утиліта, призначена для відображення поточного статусу підключень (вхідних та вихідних) по TCP/IP або UDP, таблиць маршрутизації, кількості мережевих адаптерів та статистики протоколів, активних інтернет-з’єднань, а також які застосунки активні та слухають порти.

`ps aux` – утиліта, що виводить дані про робочі процеси та задіяні порти.

`nslookup <server_name>` – запит на отримання інформації про веб-сервер (ip-адреса, назва). `ping` – перевіряє, чи доступний сервіс або застосунок.

3.4 Система доменних імен

Система доменних імен (DNS) – розподілена система для отримання інформації про домени. Свого роду, це телефонна книга Інтернету, яка зберігає відповідність між іменами та ip-адресами, а також ретранслює запити за доменними іменами на запити за ip-адресами.

Люди краще пам’ятають імена, а не числа, тому для отримання доступ до інформації в Інтернеті використовуються доменні імена, наприклад *nytimes.com* або *espn.com*. Веб-браузери взаємодіють через адреси Інтернет-протоколу (IP). DNS перетворює доменні імена в IP-адреси, щоб браузери могли завантажувати Інтернет-ресурси.

Розглянемо механізм роботи DNS.

Для завантаження веб-сторінки задіюються 4 типи DNS-серверів:

Рекурсивний DNS-сервер – сервер, призначений для отримання запитів від клієнтських машин через такі програми, як веб-браузери. Зазвичай рекурсивний DNS-сервер несе відповідальність за додаткові запити, щоб задовольнити DNS-запит клієнта.

Root-сервери[2] – сервери, які обслуговують кореневу зону DNS, широко відомі як «кореневі сервери», являють собою мережу із сотень серверів у багатьох країнах світу. Вони налаштовані в кореневій зоні DNS як 13 іменованих серверів. Root-сервери містять інформацію про доменні імена першого (верхнього) рівня: .mil, .edu, .com, .org, .net, .gov (6 оригінальних доменних імен), географічні імена першого рівня: .ua, .at, .ca, .in, .us, .fr, .uk тощо.

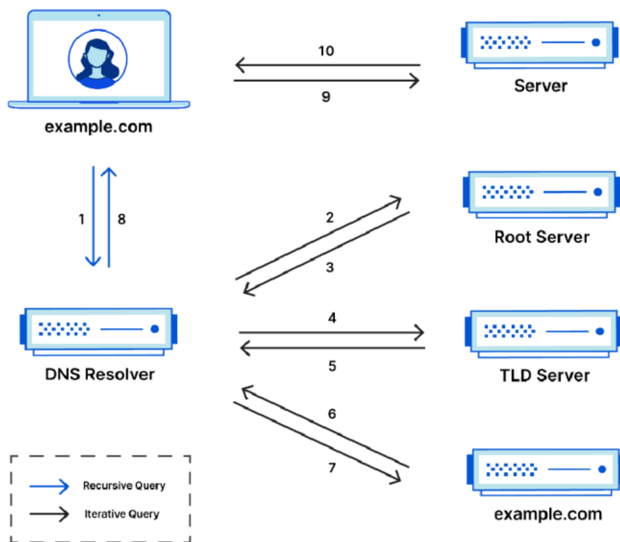
TLD-сервери – сервери доменів верхнього рівня (Top Level Domains). Дані сервери імен є наступним кроком у пошуку конкретної IP-адреси, і вони містять останню частину імені хоста (для прикладу, у example.com сервером верхнього рівня є «com»).

---

[2] Список root-серверів https://www.iana.org/domains/root/servers

Авторитетний сервер імен – останній сервер імен в циклі обробки запиту. Якщо авторитетний сервер імен має доступ до запитуваного запису, він поверне IP-адресу відповідного хоста назад DNS-рекурсору, який зробив початковий запит.

Механізм роботи DNS



1) DNS-клієнт запитує свій налаштований DNS-сервер про потрібний запис (наприклад, *https://webschool.com*) за допомогою рекурсивного запиту.

Заявлений DNS-сервер перевіряє, чи є він довіреним для потрібного запису. Якщо так, він повертає запитувану інформацію. Якщо DNS-сервер не є довіреним, він перевіряє свій локальний кеш, щоб визначити, чи нещодавно було вирішено запис. Якщо запис існує в кеші, він повертається клієнту.

2) Якщо запис не закешовано, DNS-сервер використовує серію ітераційних запитів до інших DNS-серверів, у яких запитує поданий запис. Він починається з кореневого сервера.

3) Запис повертається, якщо кореневий сервер є довіреним для запитуваного запису. В іншому випадку кореневий сервер повертає IP-адресу DNS-сервера, який є авторитетним для наступного домену нижнього рівня, у цьому випадку *.com*.

4) Первинний DNS-сервер (resolver) звертається до вказаного DNS-сервера .com за допомогою іншого ітераційного запиту.

5) DNS-сервер *.com* не є довіреним, тому повертає IP-адресу DNS-сервера *webschool.com*.

6) Оригінальний DNS-сервер звертається до вказаного DNS-сервера *webschool.com* за допомогою іншого ітераційного запиту.

7) DNS-сервер *webschool.com* є авторитетним, тому повертає необхідну інформацію — у цьому випадку це адреса IPv4 для *webschool.com*.

8) Первинний DNS-сервер кешує запис і повертає запитувану інформацію клієнту DNS.

Функції щодо створення, продажу та підтримки доменних імен делегуються організації ICANN (Internet Corporation for Assigned Names and Numbers). Дана організація керує розвитком TLD та архітектурою доменних імен, а також авторизує реєстраторів доменних імен, які реєструють та призначають доменні імена.

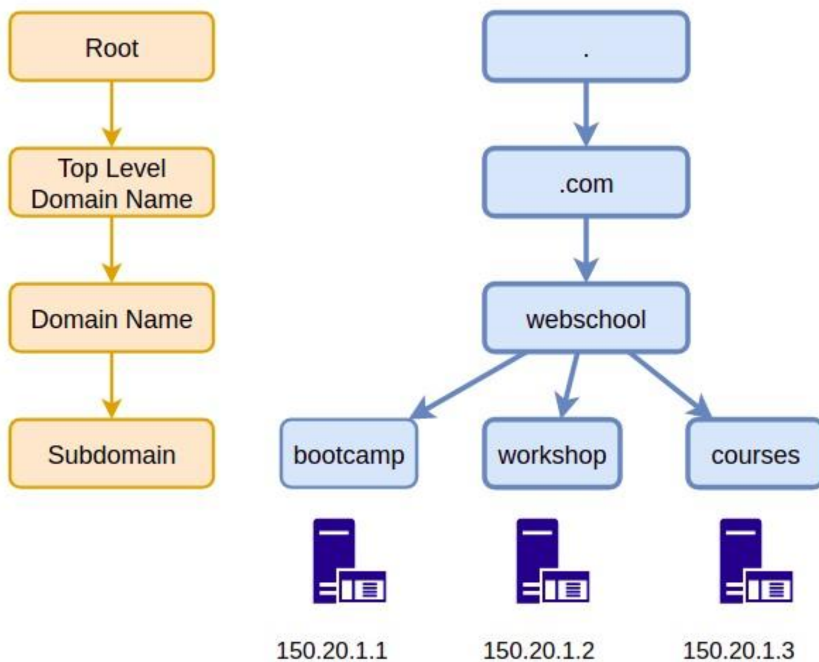
Структура доменного імені

Повне доменне ім’я містить кореневий домен в кінці (символ крапки, який можна не писати), домен верхнього рівня, домен другого рівня, домен третього рівня (у випадку субдоменів).

Домен верхнього рівня (– останній сегмент тексту в доменному імені, наприклад .com або .net. Домени верхнього рівня також називають розширеннями доменів, суфіксами доменів або розширеннями URL-адрес.

Домен другого рівня (SLD) — це ім'я ліворуч від домену верхнього рівня. Використовуються компаніями, організаціями або людьми. SLD – це перша точка контакту користувачів Інтернету з вашим веб-сайтом. Це частина URL-адреси, яка найбільше запам'ятовується, і тому найважливіша. Існує веб-сайт – example.com, який було зарезервовано для пояснення зв'язку між доменами верхнього рівня (TLD) і доменами другого рівня (SLD).

Домен третього рівня — це наступний найвищий рівень після домену другого рівня в ієрархії доменних імен. Це сегмент, який знаходиться безпосередньо ліворуч від домену другого рівня. Домен третього рівня часто називають «субдоменом».



3.5 Протокол SSH. SSH-аутентифікація

SSH (Secure Shell) – мережевий протокол, що надає захищену можливість користувачам отримати доступ до віддаленого комп’ютера (сервера) через мережу Інтернет, а також набір утиліт, що реалізують даний протокол.

Передача даних (файли, команди, запити) засобами SSH відправляються в зашифрованому вигляді. SSH має можливості щодо аутентифікації.

Типовими завданням при роботі з SSH є:

- скопіювати файл (.sh) на віддалений сервер; ● встановити ПЗ на новий сервер;
- виконати скрипт для розгортання застосунку.

Є 2 способи аутентифікації при роботі з SSH:

Спосіб 1. Використовуючи облікові дані: логін (username) та пароль (password), при умові якщо користувач зареєстрований на віддаленому сервері. Для цього адміністратор попередньо створює користувача на віддаленому сервері. Користувач може потім з’єднуватись, використовуючи логін та пароль.

Спосіб 2. Використання SSH-ключів. Опишемо детальніше схему аутентифікації з використанням пари SSH-ключів.

Користувач на своєму комп’ютері створює пару SSH ключів: приватний ключ (private key) та публічний ключ (public key). *Приватний ключ* – секретний. Зберігається захищено на машині користувача.

Публічний ключ використовується спільно, наприклад віддаленим сервером. Додавання публічного ключа на віддаленому сервері говорить йому про те, що клієнт, якому належить даний ключ, може здійснювати захищене з’єднання до даного сервера.

Клієнт може верифікувати публічний ключ, використовуючи приватний ключ. Суть проста. При аутентифікації, сервер надсилає деяке повідомлення на клієнт. Клієнт шифрує дане повідомлення використовуючи приватний ключ і надсилає його серверу. Сервер дешифрує дане повідомлення, використовуючи публічний ключ, що зберігається на сервері. Якщо повідомлення співпадають, клієнт автентифікований.

SSH для сервісів

Сервіси (такі як Jenkins, Git) часто потребують також з'єднання з віддаленим сервером через SSH для копіювання файлів або виконання скриптів. Для цього потрібно створити Jenkins-користувача на сервері з застосунком, згенерувати пару SSH-ключів на сервері Jenkins і додати публічний ключ у файл authorized_keys на сервері застосунку.

SSH та мережевий екран

Комунікація клієнта і сервера по SSH повинна бути явно дозволена через правило мережевого екрана на сервері. На багатьох серверах по замовчуванню, вона заблокована. В більшості ОС SSH-сервіс працює по замовчуванню на 22 порту. Таким чином, на сервері потрібно створити правило, що дозволяє з'єднання по 22 порту. Аутентифікація повинна відбуватись строго після встановлення з'єднання. SSH є досить потужним інструментом, що дозволяє на адміністративному рівні керувати сервером, тому вимагається обмежити доступ лише певним ір-адресам (а не всім користувачам).

Практична робота 2.1. Використання SSH

Мета роботи. Навчитись використовувати SSH при роботі з віддаленим сервером.

Завдання:

- створити віддалений сервер на хмарній платформі;
- згенерувати пару SSH ключів на локальній машині, реалізувати SSH-аутентифікацію;
- скопіювати Bash-скрипт на віддалений сервер; ● виконати скрипт на сервері.

Порядок роботи

1. Створюємо сервер та заходимо через root (супер-користувач):

- Реєструємось на *cloud.hetzner.com*, створюємо проект, додаємо сервер;
- На пошті отримуємо пароль;
- В терміналі заходимо на сервер спочатку як суперкористувач:

```
ssh root@[ip-address]
```

2. Генеруємо пару SSH-ключів:

- Перевіряємо наявність папки *.ssh* в домашній директорії на локальному комп'ютері
- Генеруємо ключі: `ssh-keygen -t rsa`

3. Додаємо публічний ключ до файлу *authorized_keys*:

- Копіюємо публічний ключ з файлу *id_rsa.pub* в файл *.ssh/authorized_keys* на сервері

authorized_keys – файл на сервері, що дозволяє йому аутентифікувати користувачів

При первинному вході на сервер, термінал запропонує додати адресу сервера до файлу *known_hosts.* Це дозволить при аутентифікації автоматично перевірити клієнту, що він з’єднується з довіреним хостом.

Після чого якщо з’єднатись, пароль вже не потрібно вводити.

Приватний ключ по замовчуванню зчитується з файлу *.ssh/id_rsa*
Команди для з’єднання з сервером:

`ssh root@[ip]` та `ssh -i .ssh/id_rsa roo@[ip]` є по суті однаковими.

4. Копіюємо Bash-скрипт на сервер та виконуємо його:

- На локальному комп’ютері створюємо bash-скрипт *test.sh* з командою, наприклад `echo "Hello, World"` ● Копіюємо файл на віддалений сервер:

```
scp test.sh root@[ip]:/root
```

При цьому приватний ключ по замовчуванню береться з .ssh/id_rsa.

- З’єднуємось з сервером і додаємо дозвіл на виконання і виконуємо команду запуску скрипта

Таким чином, в цьому розділі було здійснено огляд основ комп’ютерних мереж. Зокрема, ми розглянули:

- як працює комп’ютерна мережа;
- як комп’ютер з’єднується з мережею Інтернет;
- що таке IP-адреса, підмережа, маска підмережі, NAT;
- мережевий екран та мережеві порти;
- що таке DNS та механізм роботи DNS, доменне ім’я, структура доменного імені;
- деякі корисні команди для роботи з мережею;

- протокол SSH, аутентифікація з використанням SSH-ключів.

В наступному розділі познайомимось з технологією контейнеризації застосунків Docker.

Розділ 4. Системи контролю версій. Робота з Git

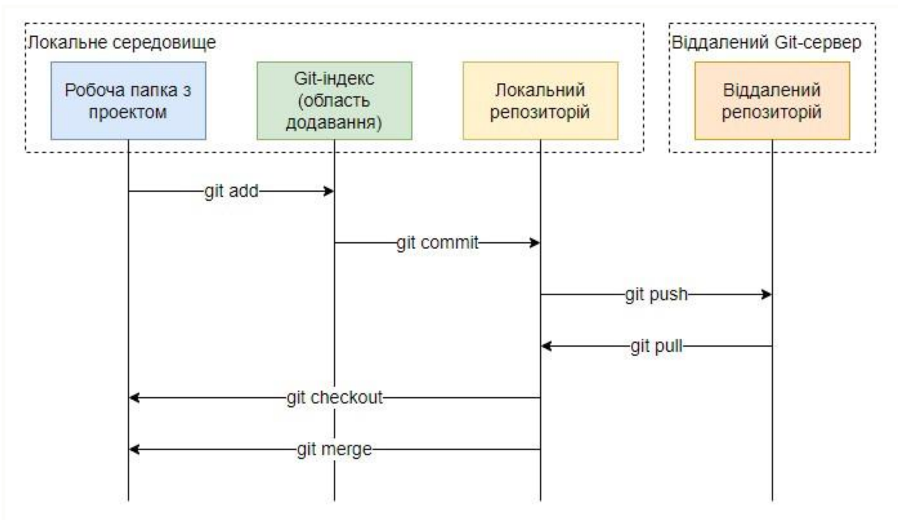
4.1 Вступ до систем контролю версій. Система Git

Система контролю версій призначена для спільної роботи над одним проектом, а також для ведення історії змін файлів з вихідним кодом. Можливостями такої системи є:

- Робота різних розробників над одним і тим же кодом;
- Код розміщений централізовано на віддаленому сховищі (репозиторії);
- Кожен розробник має повну локальну копію вихідного коду;
- Застосування змін відбувається в більшості автоматично;
- Повідомлення про конфлікти при виконанні злиття (наприклад, при зміні одного і того ж рядка коду). Такі конфлікти зазвичай вирішуються вручну;
- Кращою практикою є неперервна інтеграція: періодичне отримання і відправка змін на віддалений репозиторій;
- Повернення по історії до попередніх версій файлу (або файлів) засобами відміни зафіксованих змін;
- Кожна фіксація змін (commit) повинна супроводжуватись повідомленням; **Базові поняття Git.**

Git на сьогодні є найбільш популярною системою контролю версій (серед інших CVS, Subversion, Mercurial та інші). Git складається з таких структурних одиниць:

- віддалений репозиторій міститься на сервері (Github, Gitlab, Bitbucket). Як правило, має веб-інтерфейс користувача для взаємодії
- локальний репозиторій – локальна копія проекту, в якій можна робити зміни та відправляти їх на віддалений репозиторій
- історія змін (Git log)
- Git-індекс (область додавання) – файл, що зазвичай знаходиться в директорії Git і містить інформацію про те, що буде збережено у наступній фіксації змін (коміті).



Для взаємодії з локальним та віддаленим репозиторієм необхідно встановити Git-клієнт, який має інтерфейс командного рядка і також можливість використовувати GUI для виконання деяких команд.

4.2 Створення Git-репозиторію

Для створення віддаленого Git-репозиторію можна використати два варіанти:

- використання хмарної платформи для хостингу репозиторію (наприклад, Github або Gitlab);
- розгортання та використання власного серверу для хостингу Git-репозиторію (Bitbucket);

Репозиторії можна робити приватними і публічними. Приватні використовуються для лише для індивідуального доступу або доступу обмеженої групи розробників. Публічні репозиторії використовуються для хостингу проектів з відкритим кодом: бібліотек, технологій, мов програмування тощо.

Створення та клонування віддаленого репозиторію

Крок 1. Зареєструємось в системі Gitlab (https://gitlab.com/)**.** Підтвердимо реєстрацію в отриманому листі, здійснимо вхід на GitLab під власним логіном та паролем.

Крок 2. Створимо новий віддалений репозиторій з назвою «MyProject»:

Назва проекту (Project name): MyProject

Адреса проекту (Project URL): https://gitlab.com/{UserName}/myproject

Visibility Level: Private

Ініціювати репозиторій файлом README (Initialize repository with a README): зняти вибір

Крок 3. Встановимо Git-клієнт на власний комп’ютер. Достатньо обмежитись інтерфейсом командного рядка (Git Command Line Tool). Програмне забезпечення та керівництво по установці для необхідної операційної системи доступне за адресою: https://git-scm.com/downloads.

Крок 4. Відкриємо утиліту Git Bash та задамо налаштування Git: `git config --global user.name "firstName lastName" git config --global user.email "yourEmail@gmail.com"` **Крок 4.** Аутентифікація. Перед виконанням операцій з віддаленим репозиторієм, необхідно пройти аутентифікацію вашого комп'ютера з сервером Git.

Хорошою практикою є додавання публічного SSH-ключа до облікового запису системи Gitlab. Для генерації пари SSH-ключів використайте інструкцію з 5-ти кроків, доступну за адресою: https://docs.gitlab.com/ee/user/ssh.html#generate-an-ssh-key-pair.

Здійснивши SSH-аутентифікацію з використанням SSH-ключів, тепер немає потреби при роботі з віддаленим репозиторієм щоразу передавати логін і пароль.

Крок 5. Клонування репозиторію. Після аутентифікації даного комп'ютера в системі Gitlab, можна виконувати будь-які команди для взаємодії з віддаленим репозиторієм. Створимо папку *projects* в домашній директорії та здійснимо клонування репозиторію в дану папку: `git clone git@gitlab.com:<UserName>/myproject.git`

Крок 6. З Git Bash зайдемо в папку *myproject* щойно склонованого репозиторію. Виконаємо команду `ls -a` для виводу вмісту поточної папки і ми побачимо єдину папку *.git*, що є службовою папкою локального репозиторію. Дана папка містить всю інформацію про репозиторій: історію, індекс, гілки, конфігурацію, віддалений репозиторій тощо.

Крок 7. Виконаємо серію команд:

`cd projects/myproject` – змінюємо робочу папку на *myproject.*

`git switch -c main` – створюємо та перемикаємось на гілку *main.*

`git branch --show-current` – виводимо ім’я поточної гілки. `touch README.md` – створюємо файл README.md. `git status` – переглянемо статус локального репозиторію. При цьому файл README.md буде в списку невідстежуваих файлів. `git add README.md` – додаємо файл README.md в Git-індекс. `git status` – знову переглянемо статус локального репозиторію. Файл README.md вже буде в списку файлів, готових до фіксації. `git commit -m "add README"` – фіксуємо зміни.
`git push -u origin main` – відправляємо зафіксовані зміни на віддалений репозиторій *origin* в гілку *main.* `git log` – перегляд історії змін в локальному репозиторії.

Переконуємося, що на віддаленому репозиторії в гілці *main* з’явився відповідний коміт з щойно створеним файлом README.md.

Ініціалізація локального Git-репозиторію

Якщо потрібно створити репозиторій з проекту, що розроблений локально і не знаходиться на віддаленому репозиторії, то замість клонування сценарій дій буде дещо іншим:

1) створюємо віддалений репозиторій (наприклад, DemoNodeJSApp)
2) в корені проекту ініціюємо локальний репозиторій
3) для локального репозиторію додаємо інформацію про віддалений репозиторій
4) робимо відправку локального репозиторію на віддалений Нехай в нас є проект в папці *projects/DemoNodeJSApp* з файлом *index.js*. Для створення локального репозиторію та відправки його на віддалений виконаємо серію команд:

`cd projects/DemoNodeJSApp` – змінюємо робочу папку на *DemoNodeJSApp*

`git init` – створюємо локальний репозиторій `git checkout -b main` – створюємо та робимо поточною гілку *main*
`git remote add origin git@gitlab.com:<UserName>/demo-nodejs-app.git` – додаємо інформацію про віддалений репозиторій. *Origin* – ім’я, що використовується для посилання на віддалений репозиторій.
`git add .` – додаємо всі файли в Git-індекс `git commit -m "Initial commit"` – фіксуємо зміни `git push -u origin main` – відправляємо зафіксовані зміни на віддалений репозиторій *origin* в гілку *main*

Переконуємося, що на віддаленому репозиторії в гілці *main* з’явився відповідний коміт з кодом нашого проекту.

4.3 Робота з гілками Git

В роботі команди розробників типовим завданням є створення нових функцій або виправлення помилок в поточному застосунку. Для можливості розподілу роботи розробників а також оптимізації стану репозиторію впроваджена концепція гілок. Хорошою практикою вважається створення гілок для кожної функції та для кожного виправлення помилок. При цьому розробники можуть робити коміти в свої гілки, не турбуючись про ймовірність пошкодження головної гілки.

Протестувавши функцію і переконавшись у правильності її реалізації, або виправивши помилку, розробник здійснює злиття своєї гілки з головною гілкою. Це означає, що новостворена функція буде впроваджена з виходом нової версії застосунку. Зауважимо, що довготривала гілка зі складною функцією та великим обсягом коду збільшує ймовірність конфліктів при злитті з головною гілкою. Тому важливо правильно оптимізувати процес роботи з гілками, не робити їх занадто складними та розгалуженими, і водночас спілкуватись з іншими розробниками для попередження або вирішення конфліктів зі злиттям гілок. Система гілок передбачає, що головна гілка повинна завжди бути стабільною і готовою до випуску.

Іменування гілок бажано здійснювати у відповідності до стандартів, наприклад *feature/user-auth* (для нової функції), *bugfix/user-auth-error* (для виправлення помилки)

Створення гілки на віддаленому репозиторії

Крок 1. Перейдемо в розділ *branches*. Клацнемо *New Branch.*

Крок 2. Введемо ім’я гілки, наприклад *bugfix/user-auth-error*. В полі *Create from* виберемо *main*: дана гілка буде створена з головної гілки.

Крок 3. Передемо дані про нову гілку на локальний репозиторій. Для цього виконаємо команду:

`git pull` – передаємо зміни з віддаленого на локальний репозиторій

`git checkout bugfix/user-auth-error` – перемикаємось на гілку *bugfix/user-auth-error*

`git branch` – виводимо інформацію про наявні гілки та поточну гілку

Створення гілки через термінал

Часто зручнішим варіантом є створення гілки на локальному репозиторії з подальшою відправкою на віддалений. **Крок 1.** Виконаємо команди для створення нової гілки: `git checkout main` – перемикаємось на головну гілку `git checkout -b feature/database-connection` – створюємо нову гілку з гілки main, перемикаємось на неї

Крок 2. Для прикладу створимо файл *connect.js* з виводом рядка *“Connection to DB”*.

Крок 3. Робимо відправку на віддалений репозиторій: `git add .` – додаємо всі змінені файли в Git-індекс `git commit -m "Connect to DB feature"` – фіксуємо зміни

`git push --set-upstream origin feature/database-connection` – відправляємо на віддалений репозиторій нову гілку та коміт, зроблений в ній.

Примітка. Насправді нам не потрібно пам’ятати параметр `--set-upstream` для відправки нової гілки. Достатньо виконати `git push`, і система Git визначить, що потрібно створити нову гілку, і запропонує виконати дану команду з відповідним параметром.

Досить часто на практиці в репозиторії може бути декілька головних гілок. Наприклад, головна гілка для робочої версії (готова до випуску), головна гілка для версії розробки (develop

main branch, “кандидат” для наступного релізу). Протягом робочого процесу всі функції та виправлення помилок зливаються з головною development-гілкою. Після завершення робочого процесу, development-гілка зливається з production-гілкою.

Якщо процес розробки, тестування, збірки і доставки застосунку інтегрувати із системою неперервної інтеграції/доставки (CI/CD), то у цьому випадку необхідність використовувати головну гілку для розробки (як проміжну) відпадає. Кожна функція та кожне виправлення помилки одразу після завершення процесу розробки та тестування зливається з головною гілкою, пришвидшуючи таким чином випуск нових версій застосунку.

4.4. Робота з запитами на злиття (merge requests). Видалення гілок

Запити на злиття гілок

Хорошою практикою при злитті гілок є перевірка іншим розробником змін перед злиттям. Він переконується, що зміни стабільні та готові для робочої версії. Такі перевірки використовуються при додаванні складних функцій, або можливо при рішенні задач в специфічній галузі, що потребує експертної перевірки, або для уникнення конфліктів.

Запити на злиття створюються, якщо розробник працює в своїй гілці (наприклад, розробляє власну функцію) і бажає об'єднати свої зміни в головній гілці. Запити на злиття служать інструментом перевірки коду, і інший, як правило, більш досвідчений розробник може прийняти рішення щодо підтвердження запиту на злиття або запропонувати внести виправлення.

Для створення запиту на злиття і безпосередньо злиття гілок виконаємо такі кроки з використанням елементів UI системи Gitlab:

Крок 1. Клацнемо branches. В гілці, яку потрібно злити з головною, клацнемо Merge Request.

Крок 2. Задамо заголовок та опис запиту на злиття. Також поля "assignee" та "reviewer".

Reviewer – особа, яка переглядає запит на злиття. Власник гілки може надіслати запит на перевірку будь-якому з розробників. Assignee – особа, яка відповідає за обробку запиту на злиття після отримання коментарів і запитів на зміни від інших розробників.

Крок 3. Особа, яку призначили як Assignee, оцінює запит, переглядає коментарі від рецензентів і здійснює безпосередньо

злиття гілок. При цьому гілка, з якої відбулося злиття, може видалитись.

Видалення гілок

Після виконання злиття робочої гілки в головну, її можна залишити для подальшої роботи або видалити. В більшості дана гілка видаляється і після злиття при потребі можна створити нову гілку.

Нехай після виконання злиття гілка *feature/database-connection* на віддаленому репозиторії видалилась. Тому нам потрібно її також видалити в своєму локальному репозиторії. Для видалення гілок, що мають статус *gone*, можна використати послідовність команд:

```
# Перемикаємось на гілку main git checkout main
# Отримуємо зміни по злиттю git
pull
# Отримуємо інформацію про всі віддалені репозиторії, зокрема інформацію про видалені гілки на віддалених репозиторіях git
fetch --all --prune
# Вивести список гілок для визначення, які гілки видалені (gone)
git branch -vv
# Видалити потрібну гілку з локального репозиторію git
branch -D feature/database-connection
```

4.5 Як уникати комітів злиття (merge commits)

Припустимо, що в нас є локальний репозиторій. Ми зробили коміт, але відправку зробити не можемо, оскільки на віддаленому репозиторії інший користувач відправив новий коміт. Система запропонує спочатку отримати зміни:

```
git pull
```

В такому випадку додатково створюється так званий merge-коміт (коміт злиття віддаленої та локальної гілки). При наступній

відправці змін з локального репозиторію вже відправиться два коміти:

Jun 02, 2023

Merge branch 'feature/user-model' of gitlab.com:AlexKuzmenko/demo-nodejs-app...
Alexander Kuzmenko authored 7 minutes ago

Local commit
Alexander Kuzmenko authored 7 minutes ago

В ідеальному варіанті, merge-коміт не повинен бути частиною віддаленого репозиторію. Кращою практикою в цьому випадку є фіксація змін, а потім отримання віддалених комітів за допомогою параметра *rebase*:

Крок 1. Робимо коміт локальних змін.

Крок 2. Виконуємо `git pull --rebase <remote-name> <branch-name`

По суті, *rebase* спочатку видаляє наші коміти, які ми зробили на поточній гілці. Потім він застосує всі віддалені коміти, а потім застосує наші коміти поверх.

Крок 3. Відправляємо коміти на віддалений репозиторій: `git push`

4.6 Вирішення конфліктів злиття (merge conflicts)

Якщо паралельно робити зміни в одному файлі на різних репозиторіях, це може призвести до конфліктів. Нехай нам потрібно отримати з віддаленого репозиторію зміни файлу *user-model.js*, в якому додано 4-тий рядок коду: **`console.log("Second remote changes")`**

Причому в локальному файлі теж було додано 4-тий рядок коду: **`console.log("Second local changes")`**

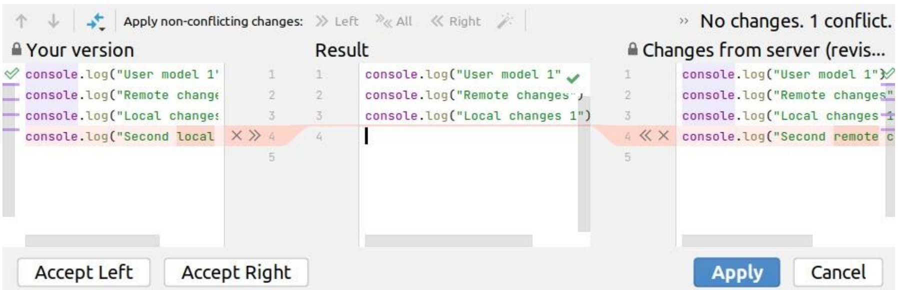
Здійснивши локальний коміт, отримаємо зміни з віддаленого репозиторію. При цьому отримаємо конфлікт при злитті: `CONFLICT`

```
(content): Merge conflict in user-model.js Automatic merge failed; fix conflicts and then commit the result.
```

При цьому сам код буде мати вигляд:

```
console.log("User   model   1")
console.log("Remote    changes")
console.log("Local changes 1")
<<<<<<< HEAD console.log("Second
local changes")
======= console.log("Second remote
changes")
>>>>>>> 45470ac30d6d887c2bbbb20d0d598064bd482929
```

Таким чином, потрібно вирішити даний конфлікт. В середовищі розробки це можна зробити з використанням візуального інструменту Resolve Merge Conflict. Під час вирішення конфлікту виводиться зліва версія локального файлу, справа – версія віддаленого файлу. По центру – результат:



Якщо прийняти ліву і праву частину, то результат вирішення буде наступним:

```
console.log("User model 1")
console.log("Remote changes")
console.log("Local changes 1")
console.log("Second local changes")
console.log("Second remote changes")
```

Вирішивши даний конфлікт, зробимо новий коміт та здійснимо відправку на віддалений репозиторій.

4.7 Виключення відстеження заданих файлів (.gitignore)

При створенні репозиторію в корені проекту розміщують файл .gitignore. В даний файл записують файли і папки, які потрібно повністю виключити з відстеження системою Git. Такими ресурсами є:

- Службові папки середовища розробки (.idea для WebStorm тощо);
- Папка build зі збіркою застосунку;
- Папка з залежностями (node_modules для NodeJS-застосунку);
- Локальні файли з конфігурацією з'єднання, обліковими даними тощо.

Крок 1. Створимо файл *.gitignore* та додамо в нього такі файли та папки:

```
node_modules/*
.idea/*
build/*
```

Крок 2. Зважаючи на те, що папка node_modules продовжує знаходитись в репозиторії, потрібно її видалити з репозиторію, залишивши локально:

```
git rm -r --cached node_modules
```

Крок 3. Робимо додавання змін в Git-індекс, фіксацію та віправку в поточну гілку: `git add . git commit -m "Added .gitignore" git push`

Переконуємось в наявності файлу *.gitignore* і що папка *node_modules* видалена з усім її вмістом на віддаленому репозиторії. З папками .idea та build (якщо вони присутні в репозиторії) виконуємо ті ж дії кроків 2 і 3.

4.8 Збереження незавершених змін (git stash)

Нехай ми зробили деякі зміни в коді в гілці feature/user-model. Далі потрібно перемикнутись на іншу гілку, проте коміт ми ще не хочемо робити. При спробі виконати команду `git checkout main`, система повідомить про помилку і не перемикне гілку:

```
error: Your local changes to the following files would be overwritten by checkout:
        user-model.js
Please commit your changes or stash them before you
switch branches.
```

Для можливості перемикання на іншу гілку із збереженням незавершених змін без необхідності робити коміт потрібно виконати команду: `git stash`

Дана команда приховає всі зміни в спеціальному сховищі. При цьому можна перемикнутись на іншу гілку. Потім знову перейти в дану гілку і повернути зроблені зміни назад: `git stash pop`

Інший корисний варіант використання є необхідність відмінити зроблені зміни в пошуках помилки. При цьому самі зміни можна запам'ятати в stash-сховищі і в подальшому відновити.

4.9 Можливість навігації по історії

Для виводу історії комітів можна виконати команду `git log`

Таким чином ми отримаємо список всіх зроблених комітів поточної гілки. Причому при потребі можна повернутись до стану будь-якого попереднього коміту (наприклад, якщо потрібно відслідкувати помилку). Для цього виконаємо команду: `git checkout <commit_hash>` де commit_hash – унікальний ідентифікатор коміту (SHA1-ідентифікатор), який можна отримати із списку історії.

Повернення до стану попередніх комітів створює відокремлений (detached) HEAD. Це означає, що після перемикання на коміт всі зміни, зроблені з цієї точки, не належатимуть до жодної гілки, якщо не буде створено нову, яка міститиме зміни з цього коміту. Для повернення до попередньої точки HEAD (останній коміт поточної гілки) виконаємо команду: `git checkout -`

4.10 Скасування або внесення змін до вже зроблених комітів

Типовою є ситуація, при якій потрібно скасувати останній коміт. Нехай ми зробили та зафіксували деякі зміни в коді. Проте згодом виявилось, що дані зміни нас не влаштовують. Таки чином потрібно скасувати коміт або декілька комітів, що знаходяться на верху історії:

```
git reset --hard HEAD~1
```

HEAD – вказівник на верхній коміт, 1 – кількість комітів, що скасовується. Параметр `--hard` відміняє і коміт і самі зміни в проекті. Якщо зміни потрібно залишити для подальшої роботи з ними, вказуємо параметр `--soft` (по замовчуванню).

Якщо поточні зміни потрібно додати в уже виконаний останній коміт, це можна зробити командою:

```
git commit --amend
```

При цьому слід мати на увазі, що дана команда не просто змінює останній коміт, вона повністю замінює його, що означає що даний коміт буде новою сутністю з власним посиланням. Для скасування коміту на віддаленому репозиторії після відповідної дії на локальному, необхідно виконати команду: `git push --force`

Важливо зауважити, що дії по скасуванню комітів ні в якому разі не можна робити в головній гілці (production або development), а лише в тій гілці, в якій ми працюємо і особисто контролюємо всю історію.

Для скасування коміту в головній гілці необхідно виконати команду:

```
git revert <commit_hash>
```

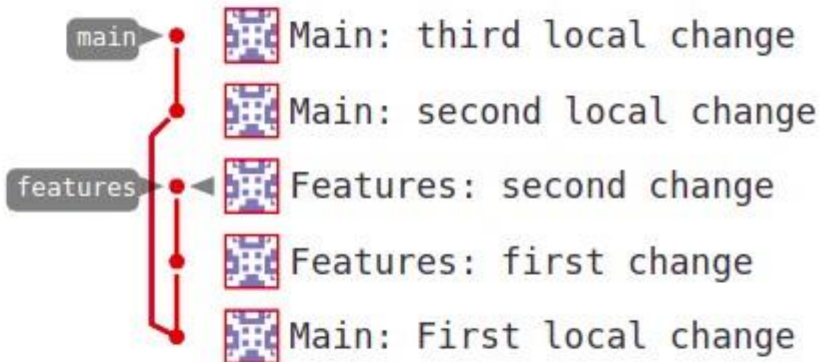
Дана команда створює новий коміт по скасуванню змін, що були присутні в даному коміті:

Revert "Fourth local changes" •••
Alexander Kuzmenko authored just now

Fourth local changes
Alexander Kuzmenko authored just now

4.11 Злиття гілок (git merge)

Нехай у нас є дві гілки: головна (main) і власна (features). Ми робимо коміти у власну гілку, а тим часом в головну гілку інші розробники відправляють власні зміни. Завдання полягає в тому, що потрібно зміни з головної гілки передати в власну гілку для підтримки актуального стану власної гілки та врахування поточних етапів розробки:

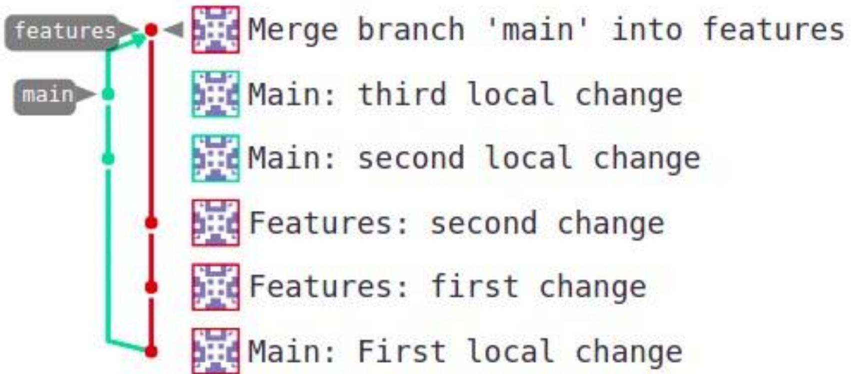


Для злиття комітів з головної гілки у власну гілку, виконаємо такі команди:

```
# Змінюємо поточну гілку на features
git checkout features
# Зливаємо коміти в поточну гілку з головної гілки
git merge main
# Переглядаємо історію і переконуємось, що останній коміт – злиття гілки main в гілку features
git log
```

Відправляємо зміни щодо злиття на віддалений репозиторій `git push`

Після цього всі зміни з гілки *main* об'єднаються з гілкою *features*, а граф репозиторію матиме такий вигляд:



Зауважимо при цьому, що зміни з гілки *features* не зіллються в гілку *main*.

4.12 Роль Git в неперервній інтеграції/неперервній доставці (CI/CD)

Таким чином, ми розглянули використання системи контролю версій Git для локальної розробки, керування віддаленим репозиторієм, послідовності виконання команд Git для вирішення ряду типових завдань. Тепер наведемо варіанти використання системи Git в контексті процесу CI/CD:

- репозиторії для керування *інфраструктурою як кодом,* наприклад для для зберігання та постачання конфігурацій, виконуваних скриптів, сценаріїв розгортання та інших ресурсів. Це можуть бути конфігураційні файли для Kubernetes, Terraform, Ansible, bash-скрипти або Python-скрипти тощо. Вимогами до таких файлів є можливість відстежувати їх історію, захищене зберігання в єдиному місці, а також можливість доступу членам команди DevOps та системним адміністраторам.
- при роботі з інструментами для автоматизації збірки застосунків, конвеєрами для CI/CD. Для цього необхідно підключити та налаштувати інтеграцію Git-репозиторію застосунку в інструмент для автоматизації збірки. Наприклад, типовим є сценарій, при якому інструмент збірки може відслідковувати надходження коміту в головну гілку і автоматично здійснювати запуск тестів, побудову та розгортання нової версії застосунку.

Розділ 5. Бази даних в процесі розробки веб-застосунків

5.1 Бази даних в веб-розробці

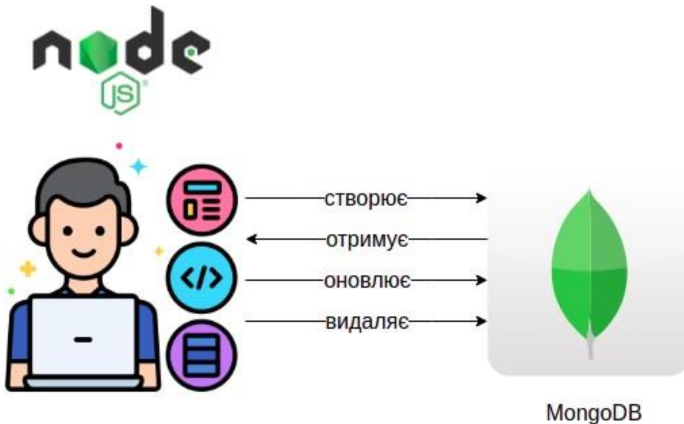
Щодня генеруються сотні мільйонів терабайт даних. Бази даних – основне сховище, в якому компанії зберігають свої дані. База даних — це організована за певними правилами інформація, що зберігається в спеціальній системі. Для обробки даних, що

зберігаються в системі, роль системи управління базою даних стає на перший план.

Системи керування базами даних (скорочено СКБД) — це інструменти, які допомагають компаніям керувати базою даних:

- організовувати структуру даних;
- працювати безпосередньо з даними (отримувати, створювати, оновлювати, оптимізувати, видаляти); ● керувати безпекою даних.

На даний час існує велика кількість систем керування базами даних, і кожна з них має унікальні функції та особливості. Тим не менш, виникає багато плутанини щодо того, яка система керування базами даних є найпопулярнішою з точки зору можливостей її функцій, має найкращий інструмент моделювання даних і ліцензування. Згідно з опитуванням 2022 року, Oracle має найвищий рейтинг, за нею йдуть MySQL, MS SQL Server, PostgreSQL, MongoDB, IBM Db2 та багато інших.



Варіанти використання баз даних в розробці

Оскільки при розробці застосунків, розробникам потрібна база даних, то існує декілька варіантів її використання:

Варіант 1. Кожен розробник встановлює базу даних локально, тобто має свою власну БД з власними тестовими даними. В цьому є перевага, оскільки ми уникаємо неузгодженості при роботі з тестовими даними. Недоліком такого варіанту є необхідність попередньо підготувати тестові дані, наближені до реальних, і наповнити ними базу даних.

Варіант 2. Встановлення і використання бази даних на віддаленому сервері (хмарний варіант). Перевагами є те, що не потрібно локальної установки, можна одразу почати писати код, а також на початку розробки одразу доступні тестові дані. Недоліком може стати неузгодженість при роботі з даними інших розробників, тому занадто експериментувати в даному варіанті не рекомендується (особливо, при роботі зі структурою даних, міграціями, імпортом тощо).

Ідеальним варіантом є можливість використовувати 2 середовища з перемиканням між локальною і хмарною БД.

5.2 Конфігурація з’єднання та спілкування застосунку з БД

Для з'єднання з БД потрібно враховувати:

- Встановлення з’єднання відбувається безпосередньо в програмному коді;
- Бібліотека і модулі для з’єднання і роботи з БД як правило наявні в будь-якій сучасній мові розробки;
- Для кожної БД бібліотеки як правило відрізняються, хоча існують засоби, що працюють одночасно з декількома БД (наприклад, розширення PDO для мови PHP);
- Розробник повинен зазначити, з якою БД потрібно з’єднатись та як здійснити аутентифікацію;

- Для локальної або віддаленої БД потрібно вказати адресу з’єднання (DB connection string)

- Якщо БД потребує логін та пароль, необхідно надати ці облікові дані для з’єднання

В коді NodeJS з’єднання виглядає наступним чином:

```
const   mongoose   =   require("mongoose")
const connectDB = async () => { const url
=
'mongodb://username:password@host:port/database?o
ptions...' await mongoose.connect(url);
}

connectDB().then(() => {
   console.log("Connected successfully")
}).catch((error) => { console.log(error.message)
})
```

Недоліком в даному коді є те, що рядок з’єднання, що містить в тому числі облікові дані, присутній в самому коді. Такого роду жорстке кодування слід уникати: адресу для БД та облікові дані потрібно виносити в змінні середовища, а конфігурація повинна зберігатись ззовні, окремо від коду самого застосунку.

Перевагами такого підходу є:

- можливість керування конфігурацією централізовано;
- гнучкість застосунку: в залежності від середовища (dev, test, prod) можна з’єднуватись з різними БД, кожна з яких потребує власні облікові дані та адресу; ● захист застосунку.

Для з’єднання визначаємо змінну в коді і надаємо їй значення із змінної середовища

Змінні середовища для з’єднання з БД можна передавати різними способами:

- з командного рядка (не зручно)
- налаштувати редактор коду (не зручно)
- створити окремий файл властивостей (або конфігурацій) для застосунку.

Можна створити різні файли для різних конфігурацій:

- *config.dev.json*
- *config.prod.json*
- *config.test.json*

Якщо файлі *.env* вказати назву поточного середовища, в залежності від якого в застосунку можна підключити той чи інший файл конфігурації:

```
MODE=dev
```

В програмному коді підключити відповідний файл конфігурації:

```
const config = require(__dirname + "/config." +
process.env.MODE + ".json")
```

Всю конфігурацію можна помістити в єдиний файл**.** До даної конфігурації в залежності від середовища можна додавати різні властивості:

- Логування для режиму розробки;
- Облікові дані для робочого режиму;
- Адреси сервісів (Service Endpoints)

```
{
 "dev": {
   "mongo": {
     "host": "localhost",
     "port": 27017,
     "database": "myDB"
   }
 },
 "prod": {
   "mongo": {
```

```
      "host": "prod-host",
      "port": 27017,
      "database": "myDBprod"
    }
  }
}
```

Бази даних в робочому режимі

Перед тим, як розгортати застосунок, необхідно встановити та сконфігурувати БД для робочого середовища (*production environment*). Для робочої версії БД враховується наступне:

- можна використовувати той же сервер або окремий;
- доцільно робити репліки (копії), бекапи (резервні копії);
- ведення моніторингу поведінки при високих навантаженнях;
- моніторинг та технічний супровід – це відповідальність системних адміністраторів або DevOps-інженерів, проте не розробників;
- в великих проектах базами даних, як правило, займається окрема команда;
- в невеликих проектах розробники можуть мати доступ до робочих баз даних і відповідальності по роботі з ними;

Межі відповідальності DevOps-інженерів

Оскільки функції DevOps-інженерів присутні на будь-якому етапі життєвого циклу застосунку, то варто відмітити їх основні компетенції для роботи з БД:

- конфігурування БД;
- встановлення БД;
- керування БД: реплікація, бекапи, відновлення;
- розуміння як розробники встановлюють з'єднання застосунку з БД.

5.3 Типи баз даних

Існує велика кількість різних БД: CassandraDB, influxDB, MongoDB, Cockroach DB, ElasticSearchDB, Redis, MemCached MySQL, PostgreSQL, Oracle DB.

Виникають запитання:

- Чому існує така велика кількість БД?
- Чим вони відрізняються?
- Яку БД вибрати для застосунку?

Очевидно, що всі бази даних відносяться до певних типів. Поширеними типами БД, що використовуються в веб-розробці є:

- Пари “*ключ-значення”;*
- Стовпчикові БД або БД з широкими стовпчиками;
- Документоорієнтовані БД;
- Реляційні БД;
- Графові БД; ● Пошукові БД.

Бази даних “Ключ-Значення”

База даних «ключ-значення» – це тип нереляційної бази даних, яка зберігає дані як асоціативні масиви, у яких ключ служить унікальним ідентифікатором. І ключі, і значення можуть мати будь-який тип: від простих типів до складних структурованих об’єктів. Прикладами таких БД є *Memcached, Redis, ETCD, DynamoDB*.

Як правило, в таких БД:

- кожен ключ є унікальним та вказує на певне значення;
- не має зв'язків та приєднаних даних;
- як правило швидкість досить висока: наприклад, дані Redis та MemCached зберігаються в оперативній пам'яті, в той час як інші типи зберігають дані в файлах;
- існує обмеженість по пам'яті;

Деякі варіанти використання БД "ключ-значення":

- часто використовується для кешування даних, що робить застосунок більш швидким та доступним в режимі реального часу;
- інколи використовується для роботи з чергою повідомлень;
- частіше всього використовуються поверх основних БД, що слугують для постійного зберігання даних.

БД "ключ-значення" використовуються великими компаніями. Наприклад, Twitter та Snapchat використовують Redis, Zoom та Netflix використовують Amazon DynamoDB. В Kubernetes є база даних ETCD, що зберігає стан кластера в режимі реального часу.

Будь-яка зміна в стані деякого компоненту кластера достатньо швидко оновить інформацію в сховищі ETCD.

Бази даних з широкими стовпчиками

Бази даних із широкими стовпчиками (стовпчикові БД) – це тип нереляційної бази даних NoSQL, у якій імена та формат стовпців можуть змінюватися в різних рядках у межах однієї таблиці. Записи в таких БД ідентифікуються за ключем, як і в попередньому типі, проте значення може бути розділене по стовпчикам.

Оскільки дані зберігаються в стовпцях, запити для певного значення в стовпці виконуються дуже швидко, оскільки весь стовпець можна завантажити та швидко виконати пошук. Пов'язані стовпці можна моделювати як частину однієї родини стовпців.

Перевагами стовпчикових БД є масштабованість, наявність мови запитів, схожої з SQL, а також гнучкість моделі даних.

Серед недоліків можна виділити відсутність приєднаних даних, обмежені можливості щодо формування і надсилання запитів, обмежену підтримку розширених функцій, складнощі в міграціях та імпорті. На відміну від реляційних БД, в стовпчикових БД не існує схеми даних. Зважаючи на це, даний тип вважається дещо простішим по відношенню до реляційних БД.

Ключ рядка	Дані рядків структуровані в стовпчиках		
10001	name	age	
	Jane	31	
10002	name	age	language
	John	41	ua

Бази даних із широкими стовпцями ідеально підходять для роботи з неструктурованими великими даними, особливо коли стовпці не завжди однакові для кожного рядка.

Варіанти використання стовпчикових БД:

- дані журналу логування;
- дані датчиків IoT (Інтернет речей), смарт карток;
- дані часових рядів, наприклад моніторинг температури або дані фінансової торгівлі;
- дані на основі атрибутів: налаштування користувача чи характеристики обладнання; ● аналітика в реальному часі.

Прикладами таких БД є *Apache Cassandra, ScyllaDB, Apache HBase, Google BigTable, Microsoft Azure Cosmos DB*.

Документоорієнтовані бази даних

В документоорієнтованих базах дані не строго структуровані і зберігаються у вигляді *документів*. Документи групуються в *колекції*, які можуть бути організовані в *ієрархію*.

Такі БД не мають можливості приєднання даних. В порівнянні з реляційними БД, вони повільніше записуються та оновлюються,

проте швидше читаються, оскільки всі дані вже можуть організовані в єдиному документі (денормалізовані).
Прикладами таких баз даних є MongoDB, DynamoDB, CouchDB.

```
{
 "users": [
   {
     "id": 1,
     "firstName": "Terry",
     "lastName": "Medhurst",
     "age": 50,
     "gender": "male",
     "email": "atuny0@sohu.com",
     "phone": "+63 791 675 8914",
     "address": {
        "address": "1745 T Street Southeast",
       "city": "Washington",
       "postalCode": "20020",
       "state": "DC"
     }
   }
 ]
}
```

Документоорієнтовані БД легко засвоюються розробниками. Підходять як основні (первинні) БД для розробки мобільних застосунків, ігор, CMS, застосунків різних типів (на відміну від попередніх типів). Не підходить для систем, в яких дані існують як окремі сутності, зв'язані між собою у формі графа з двосторонніми зв'язками (актори – фільми, товари – категорії,

друзі - пости - коментарі, соціальні мережі, тощо). Для таких варіантів підходять реляційні БД.

Реляційні бази даних

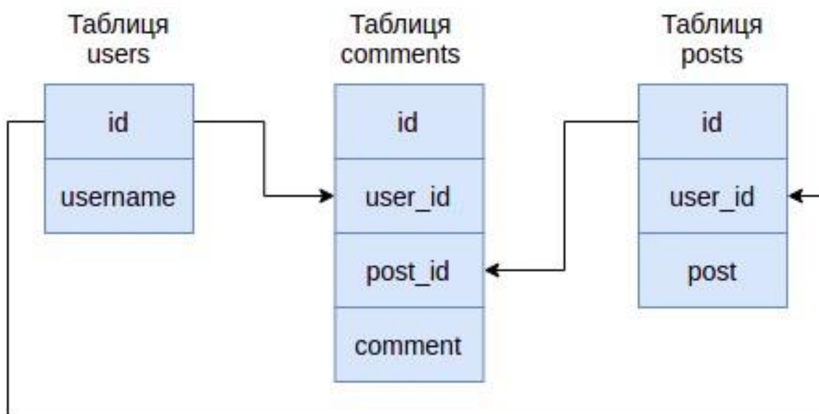
Реляційні бази даних (RDB) структурують інформацію в таблицях, рядках і стовпцях. RDB мають можливість встановлювати зв'язки (відношення) між інформацією через поля і отримувати зв'язані дані шляхом приєднання таблиць. Прикладами реляційних баз даних є *MySQL, MS SQL Server,*
PostgreSQL.

Зберігають чітко структуровані дані, вимагають строгої схеми і типізації даних, які повинні формуватись в першу чергу на етапі проектування. Для виконання запитів використовують мову структурованих запитів SQL.

Недоліки RDB: труднощі в контейнеризації, масштабуванні. Проте, наприклад *CockroachDB* спроектована так, щоб вирішувати проблему масштабованості.

RDB на сьогодні найбільш поширені і використовуються організаціями всіх типів і розмірів для широкого спектру інформаційних потреб. Це зумовлено такими можливостями:

- проста, і водночас потужна реляційна модель;
- гарантія цілісності даних під час транзакцій: при оновленні декількох таблиць, якщо падає сервер на півшляху, то транзакція скасовується, і дані не оновлюються взагалі (робиться відкат транзакції). Тобто, або всі зміни застосовуються під час транзакції, або жодна з них;
- використаннянормалізації для уникнення дублювання даних:



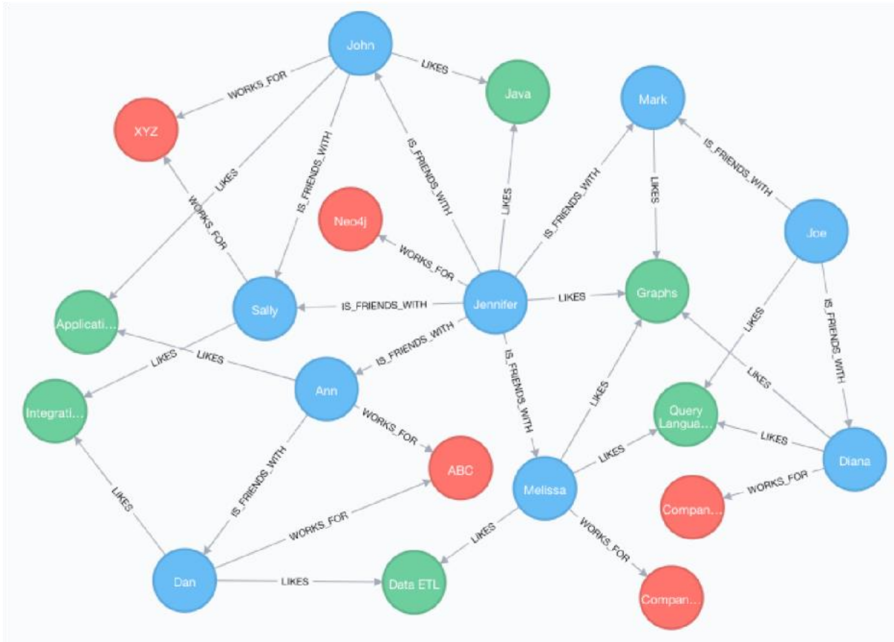
- відповідність ACID, тобто набору властивостей, що гарантують надійну роботу транзакцій бази даних: атомарність, узгодженість, ізольованість, довговічність. RDB використовуються відстеження запасів, банківських фінансових операцій, обробки транзакцій електронної комерції, оформлення замовлень, керування величезними обсягами критично важливої інформації про клієнтів і багато іншого.

Графові БД

Графові бази даних зберігають вузли та зв'язки замість таблиць або документів. Підходить для застосунків де основою є зв'язки між сутностями: графи, паттерни, рекомендації тощо, наприклад:

- зв'язки між користувачами
- користувачі та групи
- канали та підписники

Зв'язками в таких БД є ребра графа.



Прикладами графових баз даних є Neo4j, Amazon Neptune, Dgraph тощо. Використовуються такими великими компаніями як Meta (Facebook), Twitter, Amazon, Youtube .

Пошукові бази даних

Бази даних пошукових систем створені для індексування та запиту інформації. Вони оптимізовані для пошуку у великих обсягах даних на основі запитів користувачів, і повертають результати, упорядковані за релевантністю. Зазвичай вони вважаються є NoSQL(нереляційними)і використовують індекси для категоризації даних, щоб зробити пошук швидшим і ефективнішим.

Бази даних пошукових систем можуть обробляти як структуровані, так і неструктуровані дані та надавати такі можливості:

- повнотекстовий пошук

- складні пошукові вирази
- оптимізовані для пошуку алгоритми
- різні способи ранжування та структурування результатів пошуку

Прикладами пошукових баз даних є ElasicSearch, Solr, Lucene, InfluxDB.

Розглянувши особливості використання баз даних в веб-розробці, типи баз даних, варіанти їх використання, ми можемо без труднощів приймати рішення про вибір БД під потреби нашої системи або застосунку. Для великих інформаційних систем часто використовують поєднання декількох баз даних: реляційна (як первинна) + пошукова (для пошукового рушія) + ключ-значення (для кешування даних).

Розділ 6. Контейнеризація веб-застосунків. Використання Docker

6.1 Контейнеризація

Контейнеризація застосунків – це процес розгортання програмного забезпечення, який об'єднує код програми з усіма файлами та бібліотеками, необхідними для роботи в будь-якій інфраструктурі.

Традиційно (без контейнеризації), для запуску застосунку на комп'ютері нам потрібно встановити версію застосунку, а також середовище виконання та залежності, які відповідають операційній системі комп'ютера. Проте за допомогою контейнеризації ми можемо створити єдиний програмний пакунок або контейнер, який працює на всіх типах пристроїв і операційних систем.

Переваги контейнерів:

- *Портативність*. Контейнер з застосунком містить все необхідне для запуску і можуть бути запущені на будь-якій операційній системі. Розробникам не потрібно переписувати код для розгортання застосунку в різних середовищах. Всі оновлення до сучасних версій відбуваються також всередині контейнерів незалежно від ОС.
- *Масштабованість*. На одній машині можна легко додати створити і запустити декілька контейнерів для різних програм. При чому контейнерний кластер використовує обчислювальні ресурси однієї спільної операційної системи, але один контейнер не заважає роботі інших контейнерів.
- *Відмовостійкість*. Контейнери часто використовують для створення відмовостійких програм, а також для запуску мікросервісів у хмарі. Оскільки контейнеризовані мікросервіси працюють в ізольованих середовищах,

неробочий контейнер не впливає на інші контейнери. Це підвищує стійкість і доступність програми.

- *Мобільність*. Контейнерні застосунки працюють в ізольованих обчислювальних середовищах. Розробники можуть швидко виправити недоліки, змінити код, не втручаючись у роботу операційної системи, апаратного забезпечення чи інших служб програми. Таким чином, скорочується час на випуск нових версій та частіше виходять оновлення.

Варіанти використання контейнерів:

Контейнери стають все більш популярними, особливо на хмарних платформах. Багато організацій розглядають контейнери як заміну віртуальним машинам як універсальну обчислювальну платформу для своїх програм.

Перенесення на хмару – це програмна стратегія, яка передбачає перенесення застосунків (часто старі версії) у контейнери та їх розгортання в середовищі хмарних обчислень. Так, можна поступово модернізувати власні програми, без необхідності переписування всього програмного коду. Оскільки контейнери можуть працювати на будь-якій платформі з підтримкою Docker: на ноутбуках, у локальних та хмарних середовищах, вони є ідеальною базовою архітектурою для *комбінованих сценаріїв*, де робота організацій полягає у поєднанні кількох загальнодоступних хмар та власними центрами обробки даних.

Мікросервісна архітектура. Технологія контейнеризації – ключовий варіант використання при розробці і роботі мікросервісних систем на хмарних платформах. Кожен мікросервіс має унікальну та специфічну функцію. Сучасний хмарний додаток складається з кількох мікросервісів. Наприклад, інформаційна система може складатись із таких мікросервісів: серверна частина, клієнтська частина, сервіс для авторизації,

сервіс для роботи з БД, сервіс для відправки повідомлень, сервіс для проксирування запитів.

Контейнеризація мікросервісів надає інструменти для упаковки мікросервісів як програм, які можна розгортати на різних платформах. Поєднання архітектури мікросервісів та контейнерів є звичайним підходом для багатьох команд, які використовують DevOps як спосіб створення, доставки та запуску програмного забезпечення.

Пристрої Інтернету речей (IoT) містять обмежені обчислювальні ресурси, що робить ручне оновлення програмного забезпечення складним процесом. Контейнеризація дозволяє розробникам легко розгортати й оновлювати програми на пристроях IoT.

6.2 Порівняння віртуалізації та контейнеризації

Особливості організації роботи застосунків на віртуальних машинах

Розглянемо спочатку, як виглядає запуск кількох програм на сервері з *віртуальних машин*:

Інфраструктура. На нижньому шарі даного стеку розмістимо певний тип інфраструктури. Це може бути ноутбук, виділений сервер, що працює у центрі обробки даних, або віртуальний приватний сервер, який ви використовуєте в хмарі (DigitalOcean, екземпляр Amazon EC2).

Основна (хостова) операційна система: *MacOS, Windows, Linux*. Коли ми говоримо про віртуальні машини, вона як правило називається хост-системою.

Гіпервізор. Віртуальні машини фізично є звичайним файлом, в який запаковані всі можливості автономного комп'ютера. Гіпервізор – це драйвер, який дає можливість запускати цей файл. Популярними гіпервізорами 1 типу є HyperKit для macOS, Hyper-V для Windows та KVM для Linux. Популярними гіпервізорами 2 типу є VirtualBox і VMWare.

Гостьові операційні системи. Для запуску декількох ізольованих застосунків знадобиться запустити 3 гостьові операційні системи, кожна з яких контролюються гіпервізором. Кожна гостьова ОС сама по собі може бути займати достатньо багато пам'яті. В середньому 700 МБ. Крім того кожна гостьова ОС також потребує власний процесор та ресурси пам'яті.

Бінарні файли/бібліотеки. Кожна гостьова операційна система потребує власну копію різних двійкових файлів та бібліотек для роботи застосунку. Наприклад, якщо використовується Ruby, Python або NodeJS, необхідно встановлювати відповідні менеджери пакунків. Оскільки кожен застосунок має свої особливості, то очікується, що для нього потрібен свій власний набір бібліотек та залежностей.

Застосунки. Це вихідний код програми, яку ви створили, або скомпільовані виконавчі файли застосунку.
Отже, щоб кожна програма була ізольованою, нам необхідно запустити кожну з них у власній гостьовій операційній системі. Таким чином працює стек запуску віртуальних машин на сервері.

Особливості роботи застосунків в контейнерах. Можна помітити, що при такій організації все набагато легше за рахунок відсутності гостьових операційних систем. Розберемо даний стек:
Інфраструктура. Знову ж таки нам, як і для віртуальних машин, потрібна інфраструктура для запуску контейнерів. Як і віртуальні машини, це може бути ноутбук або сервер, розгорнутий на хмарній платформі.
Основна (хостова) операційна система. Будь-яка система з можливістю встановлення і запуску Docker: всі основні дистрибутиви Linux, macOS та Windows.
Daemon Docker. Демон Docker – це служба, яка працює у фоновому режимі на основні операційній системі та керує всім необхідним для запуску та взаємодії з контейнерами Docker.
Бінарні файли/бібліотеки. Замість того, щоб працювати на гостьовій операційній системі, вони вбудовуються у спеціальні пакети - образи Docker. За запуск таких образів відповідає демон Docker.
Застосунки. На останньому рівні стеку - наші застосунки. Вони можуть бути у окремих образах Docker, і демон Docker ними може керувати незалежно. Але зазвичай кожен застосунок та його залежності упаковуються в один образ Docker. При цьому, кожен застосунок залишається ізольованим. ***Переваги використання контейнеризації:***
Отже, з контейнеризацією нам не потрібно створювати та запускати віртуальну машину. Натомість демон Docker

спілкується безпосередньо з хостовою операційною системою і знає, як розподілити ресурси для запущених контейнерів Docker. Він також забезпечує ізоляцію кожного контейнера як від ОС, так і від інших контейнерів. Виграш також отримується і в часі: запуск контейнера Docker відбувається за кілька секунд. Заощадується дисковий простір та інші системні ресурси, а також не потрібна віртуалізація, оскільки Докер працює безпосередньо з хостовою ОС. Таким чином, до переваг контейнеризації віднесемо:

- заощадження дискового простору;
- швидкість запуску і горизонтальне масштабування;
- портативність, незалежність від платформи;
- підтримка сучасних тенденцій до розробки та архітектури;
- кожен компонент додатку може бути створений, розгорнений і масштабований окремо

Віртуальні машини в свою чергу непогано ізолюють системні ресурси та цілі робочі середовища. Наприклад, хостингові компанії для відокремлення системних ресурсів кожного клієнта використовують саме віртуальні машини.

Філософія Docker - ізоляція окремих програм, а не цілих систем. Ідеальним прикладом цього може бути створення образів для кожного застосунку в мікросервісній системі.

Отже, ми розглянули поняття контейнеризації і контейнерів, їх особливості і переваги використання по відношенню до віртуальних машин.

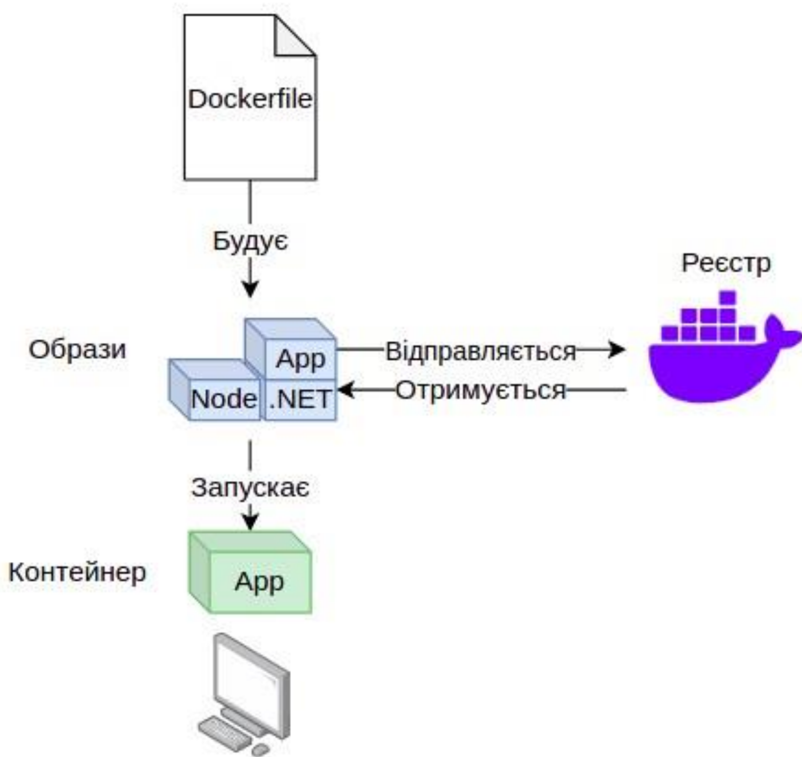
6.3 Контейнери, образи, реєстр

Контейнеризація оперує деякими важливими термінами: *контейнери*, *Dockerfile*, *образи*, *реєстр образів*.

Контейнер - це загальний пакет і середовище віртуалізації, які ми будемо використовувати. Застосунки будуть працювати всередині контейнерів. Все, що потрібно для роботи застосунку упаковується в контейнері: від залежностей до змінних середовища.

Використання контейнера дозволяє ізолювати і налаштувати середовище спеціально під потреби застосунку. Кожен застосунок працює у своєму власному контейнері.

Для запуску контейнера, спочатку потрібно визначити все, що нам потрібно: залежності, вихідний код і конфігурацію. Такі визначення можна описати в файлі *Dockerfile*, за яким будується *Docker-образ* (Docker Image).



Образ – упакована конфігурація, необхідна для запуску або розгортання контейнера. По суті, це креслення для створення, запуску і роботи контейнера. Часто образ може містити тільки застосунок, і при розгортанні контейнер займає невелику кількість місця на диску та запускається швидко. Образи, які ми створюємо для наших контейнерів, відправляються і зберігаються в реєстрі образів, наприклад *Docker Hub*.

Реєстр образів дуже схожий на репозиторій вихідного коду. Він спеціально використовується для зберігання та розгортання образів контейнерів, які повністю побудовані. Таким чином, коли приходить час для запуску образу контейнера, його немає

необхідності будувати заново. Найпоширенішим реєстром є *Docker Hub*, який дозволяє зберігати як приватні, так і загальнодоступні образи, які ми можемо використовувати. Інші приклади: *ACR (Amazon Elastic Container Registry), Google Container Registry* тощо.

6.4 Docker та його базові команди

Docker – це програмна платформа для створення, тестування, побудови, розгортання, запуску і роботи застосунків в контейнерах.

Docker пакує програмне забезпечення в стандартизовані блоки (контейнери), які містять усе, що потрібно програмному забезпеченню для роботи, включаючи бібліотеки, системні інструменти, код і середовище виконання. Використовуючи Docker, можна швидко розгортати та масштабувати програми в будь-якому середовищі з впевненістю, що код працюватиме.

Інструменти Docker:

Docker-клієнт – забезпечує засоби взаємодії розробників з докером через інтерфейс командного рядка (CLI): будувати контейнери, запускати та зупиняти їх. Через Docker-клієнт можна керувати повністю циклом життя контейнерів. В поєднанні з CLI можна використовувати застосунок Docker Desktop як GUI-реалізацію Docker-клієнту. Docker Desktop має дистрибутиви для MacOS, Linux і Windows.

Docker-compose – високорівневий інструмент для створення та запуску множини докер-контейнерів. Для конфігурації служб використовується файл з розширенням *.yml*. Далі однією командою можна створити і запустити всі сервіси. Docker Compose працює на всіх етапах процесу неперервної інтеграції та розгортання: розробки, тестування, стейджингу, розгортання, робочого стану.

Базові команди Docker: #
Отримати список контейнерів

```
docker ps [OPTIONS]
```

Приклади:

- `docker ps` – список працюючих контейнерів
- `docker ps --all (-a)` – список всіх контейнерів

- `docker ps --size (-s)` – список контейнерів з інформацією про дисковий простір
- `docker ps --quiet (-q)` – список ID контейнерів

```
# Завантаження образу з реєстру docker
pull [OPTIONS] NAME
# Побудова (збірка) образу
docker build [OPTIONS] PATH | URL
```

Приклади:

- `docker build .` – Побудова образу з Dockerfile у поточній папці
- `docker build -f <PATH>/Dockerfile` –Побудова образу з dockerfile за заданим шляхом

```
# Перегляд логів
docker logs
# Запуск контейнера з образу
docker run [OPTIONS] IMAGE [COMMAND]
```

- `docker run webapi -it bash` – запуск контейнера з образу та відкриттям терміналу bash з даного контейнера

```
# Зупинка контейнера docker
stop [OPTIONS] CONTAINER
# Зупинка всіх запущених контейнерів docker
stop $(docker ps -a -q)
# Примусова зупинка контейнера (процес не
намагається завершити роботу контейнера коректно)
docker kill [OPTIONS] CONTAINER
# Примусова зупинка всіх запущених контейнерів
docker kill $(docker ps -a -q)
# Видалення контейнера
docker rm CONTAINER
```

```
# Видалення образу
docker rmi image
```

Практична робота 5.1. Встановлення Docker та ознайомлення з базовими командами

Мета роботи: встановити Docker, ознайомитись з базовими командами для завантаження образів, запуску і зупинки контейнерів, перегляду статусу, видалення контейнерів та образів.

Порядок роботи

1. Встановіть Docker на вашу операційну систему. Запустіть Docker.

Як рекомендує офіційний сайт Docker, встановити Docker можна використовуючи GUI-застосунок Docker Desktop, який місить рушій Docker, Docker CLI, Docker Compose. Інструкції щодо встановлення для потрібної операційної системи можна отримати за посиланнями: https://docs.docker.com/desktop/.

2. Завантажте та запустіть образ *hello-world* (однією командою)

```
docker run hello-world
```

3. Завантажте і запустіть образ веб-сервера *nginx* на порту 1234. Протестуйте роботу контейнера в браузері (127.0.0.1:1234)

```
docker run --name mynginx1 -p 1234:80 -d nginx
```

4. Знайдіть в документації Dockerта виконайте такі команди:

- Перевірте статус встановленого Docker `docker --version` – отримуємо версію встановленого Docker;

`docker info` – отримуємо розширену інформацію про встановлений Docker;

- Отримайте список Docker-образів на вашій машині `docker image ls` або `docker images`
- Отримайте список запущених контейнерів на вашій машині

`docker ps`

- Отримайте список *всіх* контейнерів `docker ps --all`
- Отримайте список ідентифікаторів *всіх* контейнерів `docker ps -q --all`

5. Здійсніть зупинку контейнера *nginx* `docker stop <container_id>`
6. Видаліть всі контейнери

`docker rm <container_id>` – видаляємо один контейнер;
`docker rm $(docker ps -q --all)` – видаляємо всі наявні контейнери по ідентифікаторах.

7. Видаліть всі образи `docker rmi <image_id>` – видаляємо один образ; `docker image prune` – видаляємо всі "висячі" образи (нашарування попередніх вже видалених образів, що вже не зв'язані з жодним робочим образом); `docker rmi $(docker images -a -q)` – видаляємо всі образи.

Лабораторна робота 5.1. Використання Dockerfile для створення і запуску образу Docker. Контейнеризація NodeJS-застосунку

Мета роботи: отримати знання про розгортання та запуску NodeJS-застосунку в режимі розробки у Docker-контейнері. ***Передумови:*** встановлений Docker і базове розуміння структури NodeJS-застосунку.

Адреса репозиторію для клонування:

https://gitlab.com/web-systems-docker/docker-nodejs-app-lab1

Хід роботи:

1. Створіть новий проект в середовищі розробки *NodeJSDockerLab01*
2. Перейдіть у папку *NodeJSDockerLab01* та здійсніть клонування репозиторію з простим NodeJS-застосунком:

```
cd NodeJSDockerLab01 git
clone <repository_url> .
```

3. Створіть файл Dockerfile із такими директивами для створення образу:

```
# базовий образ, основа нашого образу
FROM node:20-alpine3.17
# створення робочої папки застосунку
WORKDIR /usr/src/app
# копіювання package.json та встановлення залежностей
COPY package*.json ./
RUN npm install
# копіюємо вихідний код в робочу папку COPY
. .
# повідомляємо Docker, що в контейнері є застосунок, що прослуховує даний порт
EXPOSE 3001
# реєструємо команду для необхідності запуску застосунку при запуску контейнера
CMD [ "npm", "start" ]
```

4. Створіть файл *.dockerignore* для запобігання копіювання локальних папок і файлів в Docker-образ:

```
node_modules
.idea npm-
debug.log
```

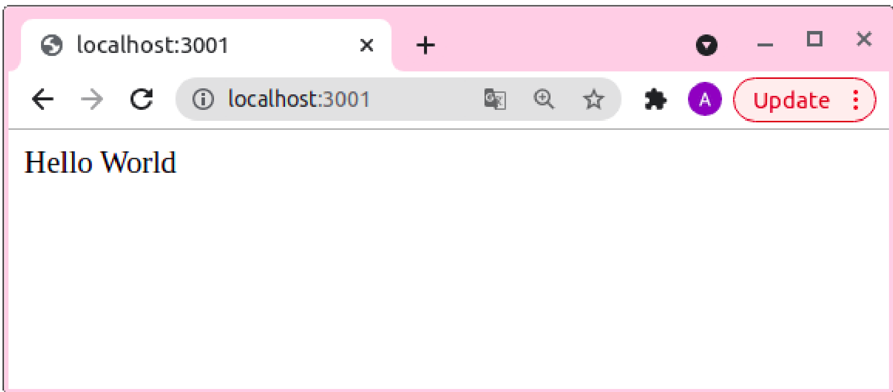
5. Побудуємо образ:

```
docker build . -t <your username>/nodejs-docker-lab01
# переконаємось, що образ побудований
docker image ls
```

6. Запустимо образ:

```
docker run -p 3001:3001 -d <your username>/nodejs-docker-lab01
```

7. Перевіримо роботу застосунку в браузері: *http://localhost:3001*



або скористаємось командою:

```
curl -i localhost:3001
```

8. Отримаємо список працюючих контейнерів і зупинимо даний контейнер: `docker ps`

```
docker stop <container_id>
```

9. Видаліть контейнер. Зробіть висновки про пророблену роботу.

Отже, ми розглянули основи роботи з Docker: стандартні команди, поняття образу, контейнера, реєстра, а також створили власний образ з Dockerfile і запустили застосунок в Docker-контейнері створеного з даного образу.

6.5 Інструмент Docker Compose

Docker Compose – це інструмент для створення та запуску багатоконтейнерних застосунків. Для цього створюється так званий маніфест – конфігураційний файл для налаштування всіх сервісів системи. Після налаштування необхідно створити і запустити всі сервіси, визначені в конфігурації. Це робиться виконанням лише однією команди.

Docker Compose можна визначити для різних середовищ по різному: розробка, тестування, стейджинг, робоче, а також у робочих процесах CI. Він також містить команди для керування всім життєвим циклом вашої програми:

- Запуск, зупинка та відновлення сервісів ● Перед статусу запущених сервісів
- Виведення журналу запущених сервісів

Docker Compose встановлюється поверх Docker. Найпростіший і рекомендований спосіб встановити Docker Compose – встановити Docker Desktop. Docker Desktop містить Docker Compose разом із рушієм Docker і Docker CLI.

Практична робота 5.2. Використання Docker Compose

Мета роботи: створити маніфест docker-compose.yml та односервісний застосунок. Налаштувати і запустити API-сервіс. Створити і використати змінну оточення.

Передумови: встановлений Docker Compose. Для перевірки можна виконати команду `docker compose --version`

Порядок роботи

1. Створіть папку для проекту *NodeJSDockerPract02*
2. Перейдіть в дану папку і створіть папку для сервісу *api*
3. В папці *api* створіть папку *src* та розмістіть файли з лабораторної роботи 5.2: *package.json*, *Dockerfile*,

.dockerignore, src/app.js.

4. В кореневій папці проекту створіть файл *docker-compose.yml* з таким вмістом:

```
version: "3"
services:
 api:
   build: ./api
   command: npm start
   ports:
       -   "3001:3001" environment:
       -   PORT=3001
       -   HOST=localhost
```

Даний файл *docker-compose.yml* визначає лише один веб-сервіс *api*. Сервіс використовує образ, створений із *Dockerfile* у каталозі *api*. Далі він реєструє команду *npm start* для запуску застосунку в контейнері і прив'язує контейнер і хост-машину до відкритого порту *3001* (зліва) з переадресацією на внутрішній порт *3001* (порт самого застосунку, записаний справа). Також в сервісі визначено дві змінні середовища: *PORT* і *HOST*.

5. Виконайте побудову котейнеризованого застосунку і його запуск:

```
docker compose up -d
```

6. Якщо потрібно перебудувати застосунок (після внесення змін в коді наприклад) і запустити, виконуємо команду:

```
docker-compose up --build -d
```

7. Перегляньте список працюючих контейнерів:

```
docker ps
```

8. Для зупинки контейнерів виконуємо команду:

```
docker compose stop
```

9. Для зупинки і видалення контейнерів виконуємо команду: `docker compose down`
10. Використаємо змінні середовища *PORT і HOST*. Змінні середовища містять інформацію, що можуть використовувати виконавчі файли програм. В подальшому ми створимо змінні середовища для портів, налаштування БД, конфігурації поштового серверу тощо.

Використання змінних оточення в коді застосунку:

```
const PORT = process.env.PORT;
const HOST = process.env.HOST;
```

Отже, ми розглянули створення простої конфігурації проекту з використанням інструменту Docker Compose та використали базові команди Docker Compose для побудови образів, запуску і зупинки контейнерів. В наступній лабораторній роботі підготуємо Docker Compose для роботи з двома сервісами: серверним застосунком та базою даних.

Лабораторна робота 5.2. Використання Docker Compose для роботи з базою даних.

Мета роботи: створити конфігурацію Docker Compose з двома сервісами: серверний застосунок api та сервіс для баз даних api_db.

Адреса репозиторію для клонування:
https://gitlab.com/web-systems-docker/docker-nodejs-app-lab2

Хід роботи:

1. Створіть новий проект в середовищі розробки *NodeJSDockerLab02*
2. Перейдіть у папку *NodeJSDockerLab02* та здійсніть клонування в поточну папку репозиторію з NodeJS-застосунком, що працює з БД:

```
cd NodeJSDockerLab02 git
clone <repository_url> .
```

3. Огляньте архітектуру застосунку. З'ясуйте, як працюють модулі, модель, маршрутизатор, як отримується конфігурація із змінних середовища.
4. В кореневій папці проекту створимо файл-маніфест *docker-compose.yml*:

```
version: "3" services:
 api:
   build: ./api
   command: npm start
   ports:
     -     "3001:3001" environment:
     -     PORT=3001
     -     HOST=localhost
     -     MONGO_URL=mongodb://api_db:27017/api
   depends_on:
     -     api_db api_db:
   image: mongo:latest
```

Даний файл *docker-compose.yml* визначає 2 сервіси: веб-сервіс *api* та сервіс БД *api_db*. Сервіс *api_db* не використовує власний Dockerfile, натомість його образ будується одразу з образу *mongo:latest*. Для сервісу api визначили змінну середовища *MONGO_URL* з рядком з'єднання з БД.

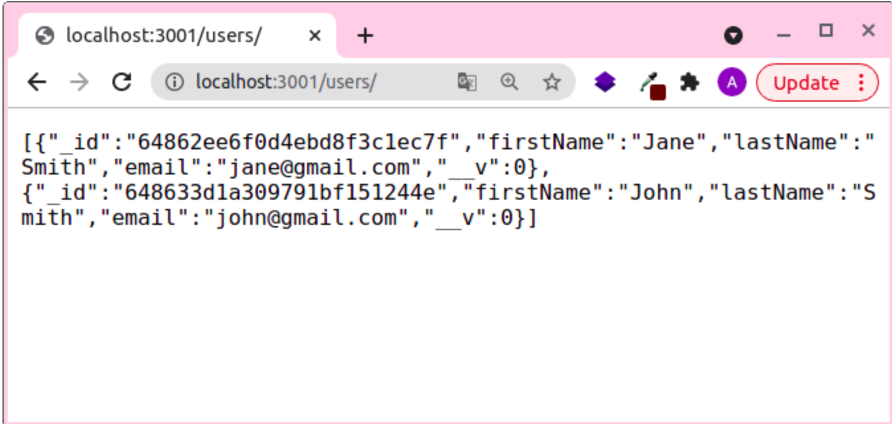
5. Виконайте побудову котейнеризованого застосунку і його запуск:

```
docker-compose up --build -d
```

6. Для виконання POST-запиту додавання нового користувача в БД можна скористатись утилітою *curl*: `curl -X POST http://localhost:3001/users -H "Content-Type: application/json" -d '{"firstName":`

```
"Jane", "lastName": "Smith", "email": "jane@gmail.com"}'
```

7. Виконайте декілька разів запит на додавання різних користувачів і протестуйте в браузері: *http://localhost:3001/users*



6.6 Використання томів Docker

Docker-томи (Docker Volumes) – це сховища даних (фізично файлова система), які створюються та використовуються Docker-контейнерами і керуються виключно Docker.

Механізм Docker-томів вирішує декілька проблем при роботі з Docker-контейнерами:

1) кожного разу при перебудові контейнера бази даних, сама база даних видаляється. Тому її варто перенести в окреме сховище (іменований том);

2) часто в проекті потрібно періодично вносити незначні зміни (в основному, в версії розробки). Для вирішення даної проблеми будемо використовувати неіменований том. *Переваги та особливості Docker-томів*

- засоби для постійного зберігання інформації;
- відокремленість від контейнерів;
- простота в здійсненні резервного копіювання та переміщення;
- можливість спільного користування різними контейнерами;
- організація ефективного читання і запису даних;
- можливість розміщувати томи на хмарних ресурсах;
- можливість шифрування даних томів;
- можливість надавати імена;
- контейнер може організувати завчасне наповнення томів даними;
- зручність для тестування.

Базові команди для роботи з томами

```
# Створення тому docker volume
create --name my_volume
```

```
# Переглянути список томів docker
volume ls
# Переглянути інформацію про конкретний том docker
volume inspect my_volume
# Видалити том docker
volume rm my_volume #
Видалити всі томи, що не
використовуються docker
volume prune
```

Створення тому під час створення і запуску контейнера

```
docker run --mount
type=volume,source=volume_name,destination=/path/in/con
tainer,readonly my_image
```

- *type* - тип монтування. Можливі значення: *bind, volume, tmpfs*;
- *source* - джерело монтування (ім’я тому);
- *destination* - шлях, до якого файл або папка монтується в контейнері;
- *readonly* - створює том, призначений тільки для читання.

Створення і використання томів з Docker Compose Для створення сховищ в yml-файлі потрібно створити новий розділ *volumes*:

```
volumes:
 mongodb_api:
```

Дана інструкція дозволить використати іменований том *mongodb_api* в будь-якому сервісі.
Створений Docker-том *mongodb_api* можна використати для підміни ним папки */data/db*, яка знаходиться в контейнері та по замовчуванню містить базу даних:

```
api_db: image:
 mongo:latest
 volumes:
   - mongodb_api:/data/db
```

Запустимо застосунок, додамо дані, зупинимо, видалимо контейнер та запустимо знову. Дані повинні зберегтися. Для перегляду списку томів виконаємо:

`docker volume ls`

Приклад використання томів для версії розробки

Для локальної розробки (dev mode) не потрібно перебудовувати повністю весь контейнер при внесенні незначних змін в файли проекту.

Створимо docker-compose.dev.yml (для версії розробки). В ньому ми не будемо перезаписувати всю конфігурацію, а лише змінимо і додамо ту конфігурацію,яку вимагає суто середовище розробки:

1) команду запуску змінюємо на *npm run dev*
2) створюємо рядковий (неіменований) том для підміни посилання робочої папки контейнера на папку з вихідним кодом. Такі посилання називаються символьними лінками (симлінками). Тепер всередині робочої директорії нічого не буде перезаписуватись. Натомість запускатись будуть скрипти з директорії, на яку веде симлінк. Загальний виглядфайлу docker-compose.dev.yml:

```
version: "3" services:
 api:
   command: npm run dev
   volumes:
```

```
  - ./api/src:/usr/src/app/src
```

3) Для запуску сервера в режимі розробки в спільноті розробників NodeJS є досить популярний пакет nodemon. Він дозволяє автоматично перезапускати серверні NodeJS-застосунки при внесенні в них змін. Директиву встановлення *nodemon* потрібно вказати в Dockerfile:

```
# …
# Встановлення залежностей
RUN npm install
# Встановлення nodemon
RUN npm install -g nodemon
# …
```

4) В режимі розробки ми використовуватимемо іншу команду: *npm run dev*. В package.json додамо скрипт для даної команди в режимі прослуховування змін:

```
"scripts": {
 "start": "node src/app",
 "dev": "nodemon --legacy-watch src/app"
},
```

5) Виконаємо команду для побудови і запуску NodeJS-застосунку в режимі розробки:

```
docker-compose -f docker-compose.yml -f docker-compose.dev.yml up --build
```

При цьому налаштування файлу *docker-compose.yml* заміняться відповідними налаштуваннями *docker-compose.dev.yml*.

Це означає, що в режимі розробки ми будемо замість реальної робочої папки проекту використовувати переадресацію на локальну папку проекту (система може запитати дозвіл на доступ до цієї папки).

Лабораторна робота 5.3. Використання Docker томів.

Мета роботи: навчитись створювати і використовувати Docker-томи

Завдання:

- створити іменований том в багатоконтейнерній Docker-системі для постійного зберігання даних
- розділити конфігурацію запуску і роботи додатку для робочого середовища (Production Mode) і середовища розробки (Development Mode)

Передумови: дана робота є продовженням лабораторної роботи 5.2, тому завдання виконуємо в тому ж проекті.

Хід роботи:

1. Зупиніть контейнери, видаліть контейнер *api_db* та перезапустіть застосунок повторно. Переконайтесь, що дані в БД не збереглися.
2. Створіть *іменований том* з іменем *mongodb_api*, для постійного зберігання даних. Перебудуйте застосунок. Переконайтесь, що видалення контейнеру *api_db* не впливає на зберігання даних.
3. Створіть *конфігурацію для середовища розробки* (файл docker-compose.dev.yml):
   - Для сервісу *api* задайте команду запуску *npm run dev*
   - Для сервісу *api* створіть *рядковий неіменований том* для переадресації з робочої папки контейнера на папку з вихідним кодом
4. В Dockerfile додайте директиву для встановлення *nodemon* та в *package.json* додайте скрипт для команди *npm run dev*

5. Здійснимо запуск контейнерів в середовищі розробки, враховуючи що потрібно використати два файла *docker-compose.yml* та поверх нього *docker-compose-dev.yml.*
6. Переконайтесь, що при роботі додатку в середовищі розробки запускатись будуть скрипти з директорії, на яку веде симлінк.

Хід перевірки виконаної роботи:

Без томів:

- запустити застосунок, додати дані в БД, зупинити застосунок;
- видалити контейнер;
- знову запустити застосунок, переконатись що БД порожня.

З томом для БД:

- запустити зазстосунок, додати дані в БД, зупинити застосунок;
- видалити контейнер;
- знову запустити застосунок, переконатись що БД не порожня.

З томом для середовища розробки:

- запустити застосунок в середовищі розробки;
- зробити незначні зміни в коді;
- переконатись, що Docker реагує на зміни в коді і перезапускає контейнер автоматично.

Розділ 7. Автоматизація збірки застосунків

7.1 Автоматизація збірки. Інструмент Jenkins

До даного моменту ми:

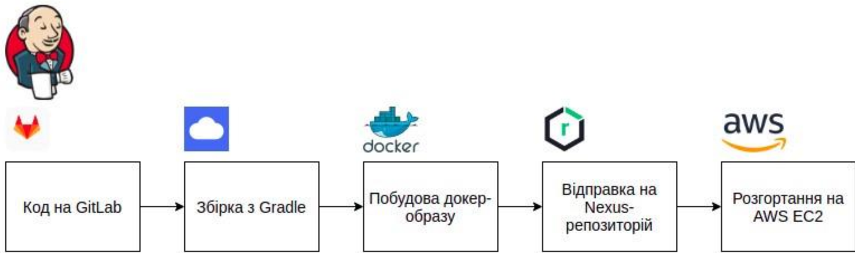
- здійснювали збірку локально
- будували докер-образи
- здійснювали відправку образів в репозиторій

При цьому даний процес супроводжується рутинними діями: зберегти робочі зміни і перенести у головну гілку, залогінитись до Docker-репозиторію, налаштувати середовище для запуску тестів, виконати тести, побудувати докер-образ та відправити його на репозиторій, опублікувати новий реліз.

Процес, який керує тестуванням коду, збіркою застосунку, будуванням докер-образу, відправкою на репозиторій, розгортанням на сервері називається *автоматизацією збірки*. Одним із популярних інструментів для автоматизації збірки є Jenkins – програмне забезпечення, що встановлюється на сервер, має UI для конфігурації, інтегрується з іншими необхідними для роботи інструментами (Docker, Gradle/Maven/npm, тощо), конфігурує задачі (запуск тестів, збірка, розгортання тощо). Таким чином, Jenkins:

- виконує тести;
- збирає артефакти;
- публікує артефакти; ● розгортає артефакти; ● надсилає сповіщення тощо.

Jenkins має плагіни для інтеграції з Docker, репозиторіями, інструментами збірки, серверами для розгортання. Припустимо, що наш застосунок написаний на Java і знаходиться на Gitlab. Нам потрібно його зібрати з Gradle, побудувати Docker-образ, відправити на Nexus-репозиторій та розгорнути на AWS EC2. Таким чином, для Jenkins потрібно використати плагіни для роботи з Gitlab, Gradle, Docker, Nexus, AWS EC2.



Для кожного етапу роботи (тестування, збірка, публікація) нам потрібно встановити і сконфігурувати відповідні інструменти та надати до них доступ сервісу Jenkins.

Таким чином користувач Jenkins повинен мати доступ до всіх цих інструментів та платформ. Підготовка всієї інфраструктури здійснюється лише один раз. При цьому, Jekins, плагіни, інструменти, облікові дані можна використовувати повторно і для різних проектів.

7.2 Приклад збірки NodeJS та ReactJS застосунку засобами Jenkins

У цьому розділі продемонструємо збірку простого NodeJS-застосунку і ReactJS-застосунку за допомогою npm. Даний застосунок створює веб-сторінку з вмістом «Ласкаво просимо в React» для простої перевірки правильної роботи.

Передумови:

- ОС MacOS, Linux або Windows;
- 256 Mb ОЗП, проте рекомендовано 2 Гб;
- 10 Гб для Jenkins та усіх необхідних Docker-контейнерів;
- Встановлене ПЗ: Docker, Git

Етап 1. Запускаємо Jenkins в Docker-контейнері

Крок 1. Створимо мостову Docker-мережу. Для цього в вікні терміналу виконаємо команду:

```
docker network create jenkins
```

Крок 2. Для того щоб виконувати команди Docker у Jenkins-контейнерах, завантажимо та запустимо образ *Docker:dind* за допомогою команди *docker run*:

```
docker run --name jenkins-docker --rm --detach \
    --privileged  --network  jenkins  --network-alias
docker \
    --env DOCKER_TLS_CERTDIR=/certs \
    --volume jenkins-docker-certs:/certs/client \
    --volume jenkins-data:/var/jenkins_home \
    --publish 3000:3000 --publish 5000:5000 --publish
2376:2376 \ docker:dind --storage-driver
    overlay2
```

Пояснення до параметрів:

- *--name jenkins-docker* – (необов'язковий) – визначає ім'я контейнера Docker для запуску образу. За замовчуванням Docker створить унікальну назву для контейнера.
- *--rm* – (необов'язковий) – автоматично видаляє контейнер при його зупинці.
- *--detach* – (необов'язковий) – запускає контейнер Docker у фоновому режимі. Контейнер можна зупинити пізніше командою docker stop jenkins-docker.
- *--privileged* – запуск Docker у Docker наразі потребує привілейованого доступу для належної роботи.
- *--network jenkins* – відносимо до мережі, що створена на попередньому кроці.

- *--network-alias docker* – робить Docker у docker-контейнері доступним через ім’я хоста docker в мережі jenkins.
- *--env DOCKER_TLS_CERTDIR=/certs* – вмикає використання TLS на сервері Docker. Завдяки використанню привілейованого контейнера це рекомендовано, хоча це вимагає використання спільного тому, описаного нижче. Ця змінна середовища керує кореневим каталогом, де зберігаються сертифікати Docker TLS.
- *--volume jenkins-docker-certs:/certs/client* – адресує каталог /certs/client всередині контейнера на Docker-том jenkins-docker-certs.
- *--volume jenkins-data:/var/jenkins_home* – адресує каталог /var/jenkins_home всередині контейнера на том jenkins-data. Це дозволить іншим контейнерам Docker, керованим демоном Docker цього контейнера Docker, монтувати дані з Jenkins.
- *--publish 3000:3000 --publish 5000:5000* – відкриває порти 3000 і 5000 від докера в докер-контейнері
- *--publish 2376:2376* – (необов’язковий) – відкриває порт демона Docker на головній машині. Це корисно для виконання команд докерів на головній машині для керування цим внутрішнім демоном Docker.
- *docker:dind* – власне docker-образ
- *--storage-driver overlay2* – драйвер зберігання для docker-тому

Крок 3. Налаштуємо офіційний Docker-образ Jenkins, виконавши наведені нижче два кроки:

- Створимо Docker-файл з наступним вмістом:

```
FROM jenkins/jenkins:2.387.3
USER root
```

```
RUN apt-get update && apt-get install -y lsb-release
RUN curl -fsSLo /usr/share/keyrings/docker-archive-keyring.asc \
  https://download.docker.com/linux/debian/gpg
RUN echo "deb [arch=$(dpkg --print-architecture) \
  signed-by=/usr/share/keyrings/docker-archive-keyring.asc] \
  https://download.docker.com/linux/debian \
  $(lsb_release -cs) stable" > /etc/apt/sources.list.d/docker.list
RUN apt-get update && apt-get install -y docker-ce-cli
USER jenkins
RUN jenkins-plugin-cli --plugins "blueocean docker-workflow"
```

- Побудуємо новий образ докера з цим Dockerfile та задамо назву образу *myjenkins-blueocean:2.387.3-1*: `docker build -t myjenkins-blueocean:2.387.3-1 .`

Крок 4. Запустимо власний образ *myjenkins-blueocean:2.387.3-1* як Docker-контейнер:

```
docker run \
  --name jenkins-blueocean \
  --detach \
  --network jenkins \
  --env DOCKER_HOST=tcp://docker:2376 \
  --env DOCKER_CERT_PATH=/certs/client \
  --env DOCKER_TLS_VERIFY=1 \
  --publish 8080:8080 \
  --publish 50000:50000 \
```

```
  --volume jenkins-data:/var/jenkins_home \
  --volume jenkins-docker-certs:/certs/client:ro \
  --volume "$HOME":/home \
  --restart=on-failure \
--env
JAVA_OPTS="-Dhudson.plugins.git.GitSCM.ALLOW_LOCAL_CHEC
KOUT=true" \ myjenkins-
  blueocean:2.387.3-1
```

Пояснення до параметрів:

- *--name jenkins-blueocean* – (необов'язковий) визначає назву Docker-контейнера для даного екземпляра образу Docker.
- *--detach* – (необов'язковий) запускає поточний контейнер у фоновому режимі і виводить ідентифікатор контейнера.
- *--network jenkins* – підключає даний контейнер до мережі jenkins. Це робить демон Docker із попереднього кроку доступним для цього контейнера Jenkins через ім'я хоста *docker*.
- *--env DOCKER_HOST=tcp://docker:2376*
- *--env DOCKER_CERT_PATH=/certs/client*
- *--env DOCKER_TLS_VERIFY=1* – змінні середовища, які використовуються Docker, Docker-Compose та іншими інструментами Docker для підключення до демона Docker з попереднього кроку.
- *--publish 8080:8080* – зіставляє порт 8080 поточного контейнера з портом 8080 на головній машині.
- *--publish 50000:50000* – (необов'язковий) зіставляє порт 50000 поточного контейнера з портом 50000 на головній машині. Це необхідно при налаштуванні одного або кількох вхідних агентів Jenkins на інших машинах, які, у

свою чергу, взаємодіють із даним контейнером jenkins-blueocean («контролером Jenkins»).

- *--volume jenkins-data:/var/jenkins_home* – зіставляє каталог /var/jenkins_home у контейнері з Docker-томом із назвою jenkins-data.
- *--volume jenkins-docker-certs:/certs/client:ro* – зіставляє каталог /certs/client на раніше створений том jenkins-docker-certs. Це робить клієнтські сертифікати TLS, необхідні для підключення до демона Docker, доступними за шляхом, указаним у змінній середовища DOCKER_CERT_PATH.
- *--volume "$HOME":/home* – зіставляє каталог $HOME на головній машині на каталог /home у контейнері.
- *--restart=on-failure* – налаштування політики перезапуску Docker-контейнера на перезапуск у разі помилки.
- env *JAVA_OPTS="-Dhudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true"* – Дозволити локальну перевірку коду з репозиторію.
- *myjenkins-blueocean:2.387.3-1* – назва створеного на попередньому кроці Docker-образу.

Етап 2. Майстер налаштувань

Крок 1. Виконаємо запит http://localhost:8080/ та введемо пароль для розблокування Jenkins. Пароль можна знайти за шляхом */var/jenkins_home/secrets/initialAdminPassword* або в логах запуску контейнера: `docker logs jenkins-blueocean`

```
*************************************************************

Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:

f3081c7f2cf2475f8f6fd2398dd9f30d

This may also be found at: /var/jenkins_home/secrets/initialAdminPassword

*************************************************************
```

Крок 2. Після розблокування Jenkins з'являється сторінка налаштувань. Натиснемо "Установити запропоновані плагіни" (Install suggested plugins). Процес ходу налаштування Jenkins і встановлення запропонованих плагінів може тривати кілька хвилин.

Крок 3. Створення першого користувача-адміністратора Далі Jenkins запропонує створити свого першого користувача адміністратора. Коли з'явиться сторінка *«Create First User Admin»*, укажіть свої дані у відповідних полях і натисніть *«Зберегти та завершити»*. Після чого можна почати використання Jenkins.

Етап 3. Створіть форк та склонуйте репозиторій Крок 1. Увійдіть у свій обліковий запис GitHub.

Крок 2. Створіть форк простого проекту *simple-node-js-react-npm-app* (https://github.com/jenkins-docs/simple-node-js-react-npm-app). Для цього на сторінці репозиторію натисніть на кнопку "Fork" **Крок 3.** Склонуємо репозиторій *simple-node-js-react-npm-app* локально на наш комп'ютер. Для цього відкриємо термінал та перейдемо в папку для репозиторію:

- macOS - */Users/<username>/Documents/GitHub/*
- Linux - */home/<username>/GitHub/*
- Windows - *C:\Users\<username>\Documents\GitHub*

Виконаємо клонування репозиторію:

```
git clone
https://github.com/ACCOUNT/simple-node-js-react-npm-app
```

, де `ACCOUNT` – ім’я облікового запису на GitHub

Етап 4. Створимо Pipeline-проект у Jenkins

Крок 1. Створимо нове завдання в *Jenkins* з назвою *“New Item”*. Вкажемо назву у полі *“Enter an Item Name”* (наприклад, *simple-node-js-react-npm-app*).

Крок 2. Виберемо тип *Pipeline* і натиснемо OK.

Крок 3. У полі *Definition* виберемо параметр *Pipeline Script From SCM (Source Control Management)*. Дана опція повідомляє Jenkins отримати pipeline-конвеєр від Source Control Management (SCM), яким є локальний Git-репозиторій. У полі *SCM* виберіть *Git*.

Крок 4. У полі *Repository URL* вкажемо шлях до каталогу локального репозиторію відносно домашнього каталогу, зіставленого з каталогом */home* контейнера Jenkins:

- Для macOS – */home/Documents/GitHub/simple-node-js-react-npm-app*
- Для Linux – */home/GitHub/simple-node-js-react-npm-app*
- Для Windows – */home/Documents/GitHub/simple-node-js-react-npm-app*

Крок 5. Натисніть *Save* для збереження нового pipeline-проекту. Тепер можна створити Jenkinsfile, який ви будете перевіряти у своєму локально клонованому сховищі Git.

Етап 4. Створимо початковий pipeline-конвеєр з використанням файлу Jenkinsfile

Pipeline-конвейер автоматизуватиме створення застосунку Node.js і React у Jenkins. Даний конвеєр буде створено як Jenkinsfile, який буде закріплено у локальному репозиторії Git (simple-node-js-react-npm-app).

Спочатку створимо початковий конвеєр для завантаження образу Node Docker і запустимо його як контейнер Docker (який створить наш простий NodeJS- і React-застосунок). Також додамо етап *Build* до конвеєра, який почне оркеструвати весь цей процес.

Крок 1. Створимо новий текстовий файл *Jenkinsfile* у кореневому каталозі локального Git-репозиторію *simple-node-js-react-npm-app*.

Крок 2. Використайте наступний код та вставте його у файл Jenkins:

```
pipeline {
    agent {
        docker {
            image 'node:lts-buster-slim'
            args '-p 3000:3000'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'npm install'
            }
        }
    }
}
```

Параметр *image* в розділі *agent* завантажує Docker-образ *node:lts-bullseye-slim* (якщо він ще не завантажений) і запускає цей образ як окремий контейнер. Це означатиме що, ми матимемо окремі контейнери *Jenkins* і *Node*, які працюватимуть локально в Docker-мережі. При цьому, контейнер *Node* стає агентом, який Jenkins використовуватиме для запуску нашого Pipeline-проекту. Однак цей контейнер недовговічний — його тривалість життя дорівнює лише тривалості виконання нашого конвеєра.

Параметр *args* робить контейнер *Node* (тимчасово) доступним через порт 3000.

Директива `stage('Build')` визначає етап “збірки”, який відображається в інтерфейсі користувача Jenkins.

Крок *sh* (у розділі *steps*) виконує команду *npm* для завантаження всіх залежностей, необхідних для запуску нашого застосунку, що будуть збережені в каталог *node_modules* (у робочий каталог проекту

/var/jenkins_home/workspace/simple-node-js-react-npm-app у Jenkins-контейнері).

Крок 3. Збережемо відредагований Jenkins-файл і зробимо його коміт у локальному Git-репозиторії *simple-node-js-react-npm-app*.

У каталозі simple-node-js-react-npm-app виконайте команди:

- `git add .`
- `git commit -m "Add initial Jenkinsfile"`

Крок 4. Повернемось до Jenkins та виберемо *Open Blue Ocean* ліворуч, щоб отримати доступ до даного інструменту.

Крок 5. Виконаємо збірку, клацнувши *Run*. Під час збірки можна переглянути, як Jenkins збирає pipeline-проект, клацнувши посилання *Open*, яке ненадовго з’явиться в нижньому правому куті.

Створивши клон локального Git-репозиторію simple-node-js-react-npm-app, Jenkins:

- спочатку ставить проект у чергу для запуску на агенті;
- завантажує Docker-образ Node і запускає його в контейнері на Docker;

> Check out from version control

> Checks if running on a Unix-like node

> docker inspect -f . "$JD_TO_RUN" — Shell Script

> Checks if running on a Unix-like node

docker pull "$JD_TO_PULL" — Shell Script

```
1   + docker pull node:lts-bullseye-slim
2   lts-bullseye-slim: Pulling from library/node
3   9e3ea8720c6d: Pulling fs layer
4   22cee24855b2: Pulling fs layer
5   1f1858138a90: Pulling fs layer
6   6e736ef1cbb5: Pulling fs layer
7   c4ef7605ff4f: Pulling fs layer
8   6e736ef1cbb5: Waiting
9   c4ef7605ff4f: Waiting
10  22cee24855b2: Verifying Checksum
11  22cee24855b2: Download complete
12  6e736ef1cbb5: Download complete
13  c4ef7605ff4f: Verifying Checksum
14  c4ef7605ff4f: Download complete
```

- запускає етап збірки (визначений у файлі Jenkins) у контейнері Node. Протягом цього часу npm завантажує багато залежностей, необхідних для запуску Node.js і програми React, які зрештою будуть зберігатися в каталозі робочої області node_modules (у домашньому каталозі Jenkins)ж
- Якщо Jenkins успішно зібрав застосунок Node.js і React, інтерфейс Blue Ocean стає зеленим:

Branch: — 2m 4s No changes

Commit: — 21 minutes ago Started by user Oleksandr

Build - 1m 36s Restart Build

Check out from version control	<1s
npm install — Shell Script	1m 36s

Крок 6. Для повернення до основного інтерфейсу Blue Ocean, натиснемо “X” у верхньому правому куті.

Етап 5. Додамо етап тестування до pipeline-конвеєра Крок 1. Повернемось до редагування файл Jenkins. Додамо наступний блок-декларацію для конвеєра безпосередньо під етапом збірки:

```
stage('Test') {
    steps {
        sh './jenkins/scripts/test.sh'
    }
}
```

Директива `stage('Test')` визначає етап, який відображається в інтерфейсі користувача Jenkins.

Крок `sh` запускає сценарій виконавчого bash-скрипта test.sh, розташований у каталозі *jenkins/scripts* відносно кореня проекту simple-node-js-react-npm-app. Сценарій виконання описано в самому файлі *test.sh*. Таким чином, винесення сценаріїв в окремі

виконавчі файли зробить код Jenkins-конвеєра простішим і зрозумілішим, що полегшить процес його обслуговування.

Крок

2. Збережемо відредагований файл *Jenkins* і зафіксуємо зміни у локальному Git-репозиторії. Для цього у каталозі *simple-node-js-react-npm-app* виконаємо команди:

- `git stage .`
- `git commit -m "Add 'Test' stage"`

Крок 3. Повернемось до середовища Jenkins, при необхідності здійснимо вхід і перейдемо до інтерфейсу Blue Ocean Jenkins. Запустимо збірку, клацнувши *Run*, і одразу клацнемо посилання *Open*, яке ненадовго з'явиться в нижньому правому куті, щоб побачити, як Jenkins запускає змінений проект Pipeline. В разі неможливості натиснути *Open*, клацнемо верхній рядок інтерфейсу *Blue Ocean*, щоб отримати доступ до цієї функції.

Примітка. Під час цього запуску ми помітимо, що Jenkins більше не потрібно завантажувати образ Node Docker. Натомість йому потрібно лише запустити новий контейнер із завантаженого раніше образу Node. Крім того, жодних нових залежностей npm не потрібно завантажувати на етапі побудови. Таким чином, запуски нашого конвеєра на даний і наступні спроби має бути набагато швидшим.

Якщо змінений Pipeline запрацював успішно, ось як має виглядати інтерфейс *Blue Ocean*. Зверніть увагу на додатковий етап *Test*. Також можна вибрати попередній етап *Build*, щоб отримати доступ до результатів цього етапу.

Крок

4. Повернемось до основного інтерфейсу Blue Ocean, натиснувши *X* у верхньому правому куті.

Етап 6. Додамо останній етап (доставки) до конвеєру

Крок 1. Відкриємо файл Jenkins для редагування, скопіюємо та вставимо наведену нижче декларацію конвеєра безпосередньо під етапом тестування:

```
stage('Deliver') {
    steps {
        sh './jenkins/scripts/deliver.sh'
            input message: 'Finished using the web site? (Click "Proceed" to continue)'
        sh './jenkins/scripts/kill.sh'
    }
}
```

- Директива `stage('Deliver')` визначає новий етап – етап доставки, який з'являється в інтерфейсі користувача Jenkins.
- Крок `sh './jenkins/scripts/deliver.sh'` запускає bash-скрипт *deliver.sh*, розташований у каталозі

Крок

jenkins/scripts відносно кореня проекту. Сценарій роботи описано в самому файлі *deliver.sh*.

- `input message` – крок введення (надається плагіном Pipeline: Input Step) призупиняє запущену збірку та пропонує користувачеві (за допомогою спеціального повідомлення) продовжити або перервати виконання.
- Крок `sh './jenkins/scripts/kill.sh'` запускає сценарій *kill.sh*, розташований у каталозі *jenkins/scripts*. Пояснення щодо роботи сценарію, описано в самому файлі *kill.sh*.

Крок 2. Збережемо відредагований файл Jenkins і зафіксуємо зміни у локальному Git-репозиторії, виконавши команди:

- `git stage .`
- `git commit -m "Add 'Deliver' stage"`

3. Повернемось до Jenkins і перейдемо до інтерфейсу Blue Ocean Jenkins.

Клацнемо *Run* для побудови. Якщо змінений Pipeline запрацював успішно, в ланцюжку з'явиться додатковий етап *Deliver*.

Крок 4. В розділі *Deliver* натиснемо зелений крок ./jenkins/scripts/deliver.sh, щоб розгорнути його вміст, і прокрутимо вниз до посилання http://localhost:3000.

Крок 5. Клацнемо посилання для перегляду запущеного застосунку Node.js і React (у режимі розробки) у новій вкладці веб-браузера. Ми повинні побачити сторінку/сайт із заголовком Welcome to React.

Крок

Крок 6. Можна поекспериментувати та отримати доступ до терміналу/командного рядка Docker-контейнера Jenkins і з редактором vi змінимо та збережемо вихідний файл App.js. Результати з'являться на веб-сторінці *Welcome to React*. Для цього виконаємо такі команди:

- `docker exec -it <docker-container-name> bash`

- `cd /var/jenkins_home/workspace/simple-node-js-reactnpm-app/src`
- `vi App.js`

Крок 7. Після завершення перегляду сторінки, натиснемо кнопку *Proceed* для завершення виконання конвеєра.

Крок 8. Повернемось до головного інтерфейсу *Blue Ocean*, де перераховані ваші попередні запуски Pipeline у зворотному хронологічному порядку.

Підсумки

Таким чином ми використали Jenkins для створення простого застосунку Node.js і React з допомогою npm. Етапи «Build», «Test» і «Deliver», які ми створили вище, є основою для створення більш складних NodeJS- і React-застосунків в Jenkins, а також які інтегруються з іншими стеками технологій. Jenkins досить розширюваний, тому його можна модифікувати та налаштувати для обробки практично будь-якого аспекту автоматизації побудови та розгортання застосунків.

Практична робота 6.1 Встановлення і використання Jenkins на віртуальному сервері Digital Ocean

Мета роботи: встановити Jenkins на віртуальному сервері Digital Ocean.

Хід роботи:

Існує 2 способи встановлення Jenkins на віртуальному сервері:

1) безпосередньо на операційну систему:
   - завантажити пакет і встановити;
   - створити користувача Jenkins на сервері;
2) запустити Jenkins в Docker-контейнері

Ми будемо використовувати 2 спосіб. Наведемо кроки по встановленню:

1. Створюємо віртуальний сервер на Digital Ocean. Для навчальних цілей буде достатньо мінімальний тарифний план. Для робочих проектів та довготривалого користування можна вибрати план: 4Gb RAM/2CPU/80Gb SSD/4 Tb transfer.
2. Створюємо та конфігуруємо мережевий екран для Jenkins. В ньому дозволяємо SSH (порт 22), а також додаємо власне правило безпосередньо для Jenkins: протокол TCP, порт 8080. Зв'язуємо мережевий екран з сервером Jenkins.
3. Отримуємо рядок з'єднання та з'єднуємось з локального комп'ютера: ssh root@<server_name>
4. Встановлюємо середовище Docker:

- `apt update`
- `apt install docker.io`

5. Запускаємо Jenkins в контейнері та перевіряємо роботу сервісу:

- `docker run -p 8080:8080 -p 50000:50000 -d -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts`
- `docker ps`

6. В браузері виконуємо запит: http://<ip_address>:8080 та отримуємо майстер первинних налаштувань. Копіюємо шлях для отримання паролю адміністратора.
7. В консолі заходимо в контейнер та відкриваємо файл з паролем:

- `docker exec -it <container_id> bash`
- `cat var/jenkins_home/secrets/initialAdminPassword`

8. Копіюємо пароль та вставляємо його в поле Administrator Password в браузері для ініціалізації Jenkins. Після ініціалізації можна встановлювати плагіни які пропонуються.
9. Створюємо користувача – адміністратора Jenkins. Після встановлення Jenkins отримуємо доступ до панелі керування Jenkins. UI-елементи відносяться до відповідних ролей для Jenkins-застосунків
   - Адміністратор (DevOps): керує Jenkins-сервісом, встановлює сервер, плагіни, створює резервні копії Jenkins-даних. В UI є відповідна секція “Manage Jenkins”
   - Користувачі (Developers або DevOps): фактично створюють завдання для запуску робочих процесів, використовуючи елементи графічного інтерфейсу

Практична робота 6.2 Встановлення інструментів для збірки

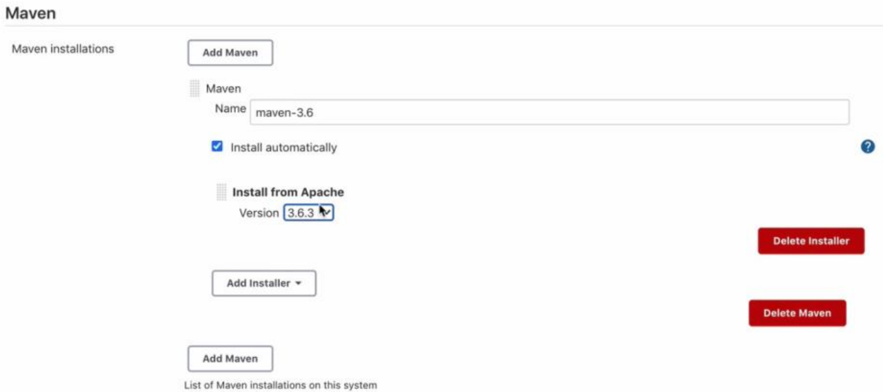
Мета роботи: встановити інструменти для безпосередньої збірки.

Пояснення: Нехай ви створили Java-застосунок. Потрібно запустити тести та зібрати JAR-файл. Таким чином Jenkins повинен мати плагін *Maven*. Інший приклад – застосунок написаний на NodeJS. Потрібно запустити тести, упакувати та відправити на репозиторій. Jenkins повинен використовувати плагін для *npm*. Є 2 способи встановлення та конфігурування інструментів для збірки на Jenkins:

1) Установка інструментів як плагінів для Jenkins (засобами UI);
2) Установка безпосередньо на сервер:
   a) на рівні ОС;
   b) всередині Docker-контейнера, в якому працює Jenkins.

Хід роботи:

1. Першим способом можна наприклад встановити та сконфігурувати *Maven*. Для цього потрібно перейти в *Global Tool Configuration* та встановити плагін *Maven*:



2. Другим способом можна навести приклад встановлення NodeJS та NPM в Docker-контейнері. Наведемо команди для встановлення:

```
# Отримуємо ідентифікатор контейнера з Jenkins:
docker ps
# Заходимо в контейнер:
docker exec -u 0 -it <container_id> bash
# Отримуємо версію дистрибутиву: cat
/etc/issue
# Оновлюємо індекс apt:
apt update
# Встановлюємо curl:
apt install curl
# Завантажуємо скрипт для NodeJS:
curl -sL https://deb.nodesource.com/setup_10.x -o
nodesource_setup.sh
```

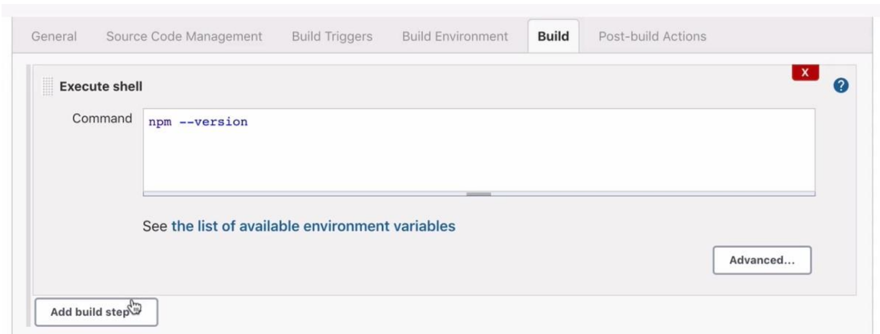
```
# Запускаємо скрипт для завантаження nodeJS:
bash nodesource_setup.sh
# Встановлюємо NodeJS:
apt install nodejs
# Перевірямо NodeJS:
nodejs -v
# Перевіряємо NPM:
npm -v
```

Практична робота 6.3 Створення простих завдань, конфігурація плагінів

Мета роботи: створити декілька простих завдань, використовуючи різні можливості Jenkins.

1. Виконаємо кроки по створенню *простого довільного завдання*:

- Створюємо новий елемент завдання (New Item) та вводимо назву завдання *my_job*. Задаємо тип *Freestyle job*
- В групі налаштувань *Build* вибираємо *Execute shell*.
- В полі *Execute shell* (командний рядок Jenkins-контейнера), вводимо команду `npm --version`:



- Додаємо наступний крок: *Invoke top-level Maven targets*. Вибираємо версію *Maven* і в полі *Goals* записуємо будь-яку команду Maven. Наприклад `--version`;
- Зберігаємо завдання;
- Будуємо завдання: *my-job – Build Now*;
- Після побудови, в контексті отриманої збірки можна переглянути логи виконання завдання: *my-job – #<build_version> – Console Output*. Зокрема в логах можна віднайти шлях до встановленого *Maven*.

2. *Завдання для самостійного виконання.* В середовищі *Jenkins UI* встановіть плагін для *NodeJS*, додайте новий крок в завдання із вказанням дії виконання *NodeJS*-скрипта. Порівняйте UI для різних кроків при різних плагінах (*Maven*, *NodeJS*, команди *Shell*). Який варіант більш зручний? Який більш гнучкий?
3. *Завдання для самостійного виконання.*
    - закомітьте на репозиторій файл з bash-скриптом *freestyle-build.sh* в якому запишіть команду для виводу версії *npm*;
    - створіть Jenkins-завдання, в якому задайте команду для виконання даного файлу:
        - `chmod +x freestyle-build.sh`
        - `./freestyle-build.sh`
    - запустіть завдання, перегляньте логи та переконайтесь у правильності виконання.