Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

СИСТЕМА КЕРУВАННЯ КОНФІГУРАЦІЯМИ КОМПОНЕНТІВ МЕРЕЖІ ПІДПРИЄМСТВА

Текстова частина до кваліфікаційної роботи
за спеціальністю "Комп'ютерні науки" 122

Керівник курсової роботи
к. т. н. Черкасов Д.І.

_____

(підпис)

"____" _____ 2023 р.

Виконала студентка КН-4
Пономаренко К.О.
"____" _____ 2023 р.

Київ 2023

Міністерство освіти і науки України

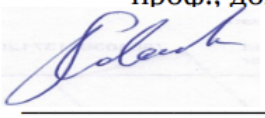
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мережних технологій,

проф., доктор фіз.-мат. наук



_____ Малашонок Г. І.

(підпис)

„____”_____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студентки 4 року навчання БП "Комп’ютерні науки"

Пономаренко Катерини Олександрівни

на тему:

Система керування конфігураціями компонентів мережі підприємства

Зміст ТЧ до кваліфікаційної роботи:
Індивідуальне завдання
1. Вступ. Аналіз предметної області.
2. Огляд можливих рішень для систем керування конфігураціями компонентів мережі підприємства
3. Структурна розробка рішення
4. Детальна розробка окремого компоненту рішення
Висновки
Список використаних джерел
Додатки (за необхідністю)
Дата видачі: „___” _____ 2022 р.

Керівник: к. т. н. Черкасов Д.І._____ (підпис)

Завдання отримала _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапу кваліфікаційної роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на кваліфікаційну роботу	30 грудня 2022	
2.	Огляд літератури за темою роботи	31 січня 2023	
3.	Дослідження існуючих рішень для систем керування конфігураціями компонентів мережі підприємства	28 лютого 2023	
4.	Вибір компонентів системи керування конфігураціями компонентів мережі підприємства, що розробляється	10 березня 2023	
5.	Структурна розробка рішення	02 квітня 2023	
6.	Детальна розробка окремого компоненту рішення	19 квітня 2023	
7.	Написання текстової частини роботи.	29 квітня 2023	
8.	Створення презентації	05 травня 2023	
9.	Корегування роботи згідно з результатами перевірки роботи науковим керівником.	10 травня 2023	
10.	Остаточне оформлення пояснювальної роботи та презентації.	17 травня 2023	
11.	Подання роботи на кафедру для перевірки на плагіат.	22 травня 2023	
12.	Захист курсової роботи.	Кінець травня 2023	

Зміст

Перелік використаних скорочень

СККМ - Система керування конфігураціями компонентів мережі

ПЗ – програмне забезпечення

ВМ – віртуальна машина

ЦП – центральний процесор

API – Application Programming Interface

CLI – Command Line Interface

DSL – Domain-Specific Language

FTP – File Transfer Protocol

HTTP – HyperText Transfer Protocol

RAL – Resource Abstraction Layer

SDK – Software Development Kit

SFTP – Secure File Transfer Protocol

SNMP – Simple Network Management Protocol

SSH – Secure Shell

YAML – Yet Another Markup Language

Анотація

У цій роботі досліджується важливість систем керування конфігураціями компонентів мережі підприємства, а також розглядаються можливі підходи до їх проектування та розробки у відповідності до різноманітних вимог, яким повинні відповідати такі системи.

В практичній частині роботи проводиться структурна розробка власного рішення для мережі, яка містить значну кількість розподілених у просторі пристроїв та потенційно може вимагати масштабування. Додатково виконується детальна розробка одного з компонентів розробленої системи.

Вступ

Комунікаційна мережа є невід'ємною складовою частиною інформаційної інфраструктури сучасних підприємств. Основними показниками ефективності її функціювання є швидкість передачі даних, запобігання втратам інформації, а також висока доступність за рахунок стійкості до відмов, які можуть виникати під час її роботи. [1]

Сучасна комунікаційна мережа містить велику кількість пристроїв, кожен з яких має індивідуальну конфігурацію. Взаємне узгодження конфігурацій пристроїв забезпечує ефективність функціювання мережі. Некоректне налаштування конфігурації, або пошкодження призводять до неправильного функціонування пристрою чи припинення його роботи, та, в результаті, до негативних, а в деяких випадках і фатальних, наслідків для всієї мережі. Втрата конфігурації пристрою суттєво ускладнює відновлення його працездатності, збільшує загальний час усунення неполадок.

Для надійної роботи мережі необхідно мати технологію та інструментарій для збереження конфігурацій пристроїв та їх оперативного відновлення у разі потреби.

Оперативне керування конфігураціями пристроїв може бути реалізованим за допомогою системи керування конфігураціями компонентів мережі.

Серед основних переваг використання такої системи можна перелічити наступні:

- швидке відновлення працездатності пристроїв з втраченою або пошкодженою конфігурацією [2],
- контроль змін, які були внесені в конфігурацію під час експлуатації пристрою, які могли впливати на зміну показників функціювання мережі,
- автоматизація збору інформації щодо поточних налаштувань пристроїв,
- ефективне колективне адміністрування мережі,

- централізоване коригування конфігурацій,
- використання надійного сховища конфігурацій пристроїв,
- можливість створення історії змін у конфігураціях (з висвітленням інформації про час зміни та її виконавця),
- підвищення ефективності процесу пошуку помилок.

Кожне підприємство має свою індивідуальну структуру мережі та певні технічні і організаційні відмінності, які мають бути врахованими під час створення системи. Проте можливо виділити основні складові частини будь-якої ефективної системи керування конфігураціями компонентів мережі:

- високонадійне сховище даних,
- система, що буде завантажувати конфігурації на девайси,
- система контролю версій,
- підсистема, що буде надавати можливість автоматизовано збирати конфігурації,
- база даних,
- веб-інтерфейс,
- API.

Основним фокусом даної роботи є дослідження можливих варіантів архітектури такої системи, їх порівняння між собою. Також метою роботи є розробка структури власного рішення та компоненту застосунку системи керування конфігураціями компонентів мережі.

Розділ 1. Огляд можливих рішень для систем керування конфігураціями компонентів мережі підприємства

1.1. Завдання, які повинна ефективно виконувати система керування конфігураціями компонентів мережі підприємства

Система керування конфігураціями компонентів мережі (далі - СКККМ) повинна виконувати всі зазначені нижче задачі.

Керування конфігураціями.

СКККМ повинна забезпечити періодичне зчитування і збереження поточної конфігурації (із застосуванням текстового формату), а також збереження хронологічної інформації щодо поточної та попередніх конфігурацій.

Всі зміни в конфігурації повинні вноситися тільки у разі їх впровадження та / або погодження адміністратором, і СКККМ має містити механізми для такого погодження. При цьому необхідно також передбачити процедуру обов'язкового збереження резервних копій безпосередньо перед внесенням змін.

У випадку відхилення змін адміністратором, СККМ повинна автоматично відновити попередню або наперед визначену конфігурацію зі сховища.

Контроль поточного стану мережі.

СКККМ повинна забезпечити періодичне зчитування показників поточного стану мережі в текстовому форматі, а також автоматичне збереження показників поточного стану та фіксацію його змін.

Про будь-які зміни поточного стану СКККМ повинна інформувати адміністратора в режимі реального стану.

Керування резервними копіями конфігурацій.

СККККМ повинна відправляти на пристрої команди для створення резервних копій конфігурації, завантажувати ці копії та передавати на зберігання до сховища. Крім того, СККККМ повинна забезпечувати можливість відновлення конфігурацій маршрутизаторів з резервних копій, що зберігаються у сховищі.

Керування всіма елементами програмного забезпечення мережі.

СККККМ має містити механізми та алгоритми для перевірки поточної версії програмного забезпечення, встановленого на пристроях, в тому числі на їх відповідність базовій версії. Також завданням СККККМ є завантаження на пристрої файлів програмного забезпечення (в тому числі тих, що знаходяться на локальних файлових системах).

Для побудови СККККМ, яка буде ефективно виконувати всі зазначені вище завдання, необхідно перш за все визначитися з архітектурою та платформою, на якій таку систему найбільш доцільно будувати.

1.2. Архітектурні шаблони для побудови СККККМ - порівняння монолітної та модульної архітектури. Мікросервіси як одне з перспективних рішень модульної архітектури.

Для побудови системи керування конфігураціями компонентів мережі підприємства найбільш доцільним є використання наступних архітектурних шаблонів - монолітна архітектура та модульна архітектура.

1.2.1. Монолітна архітектура

Монолітна архітектура - це традиційна модель побудови програмного забезпечення як окремої обчислювальної мережі з єдиною базою коду. Така

обчислювальна мережа працює незалежно від інших додатків і має високий ступінь автономності.

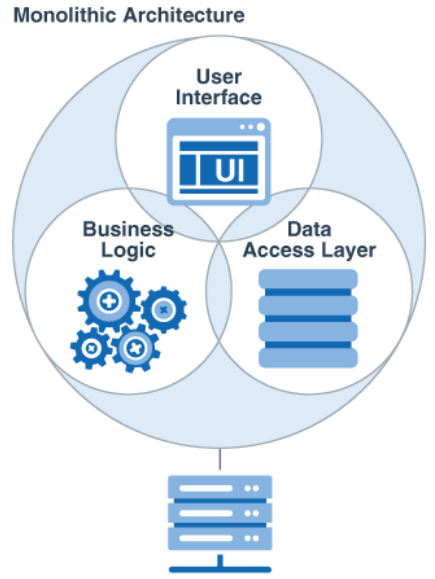


Рис.1 Візуальне подання монолітної архітектури

Основними перевагами монолітної архітектури є:

a) простота організації процесу розробки, так як використовується одна база коду;
b) простота впровадження (розгортання), так як використовується один файл або каталог, що виконується;
c) відсутність затримок відгуку ПЗ та оптимальне навантаження на мережевий трафік, так яку централізованій базі коду та репозиторії всі функції виконує один інтерфейс API (на відміну від модульних систем, які використовують більш розгалужену мережу для спілкування між автономними модулями);
d) швидке наскрізне тестування, так як система побудована як єдиний централізований модуль, а не як розподілена програма;
e) зручне налагодження, так як весь код знаходиться в одному місці і тому легше виконувати запити та знаходити проблеми.

Разом з перевагами, монолітна архітектура має свої обмеження та недоліки:

a) обмежене масштабування – монолітна архітектура дозволяє масштабувати тільки всю систему в цілому, навіть якщо є потреба у масштабуванні однієї частини [3];
b) труднощі імплементації для систем з багаторівневою структурою (містять всі архітектурі рівні: front-end, middleware, back-end) через те, що неможливо внести зміни до одного рівня не чіпаючи інші, так як зв'язки між рівнями дуже сильні;
c) повне повторне тестування всієї системи після внесення змін – через те, що при зміні окремого компонента системи впливає на систему в цілому, необхідно проводити повне тестування всієї програми, а також повний перерозподіл всіх екземплярів;
d) недостатньо високий рівень надійності системи – при наявності помилки в одному компоненті системи вся СККККМ може стати недоступною для користувача;
e) перешкоди впровадженню нових технологій - будь-які зміни в інфраструктурі чи мові розробки впливатимуть на додаток цілком, що може призводити до зростання вартості вдосконалення та обслуговування СККККМ.

1.2.2. Модульна архітектура

Модульна архітектура передбачає, що система складається з певної кількості достатньо автономних модулів та спільної частини, яка відповідає за ініціалізацію та запуск всіх цих модулів.

Кожному модулю властиві наступні якості:

a) виконується незалежно від коду інших модулів;
b) має власне задокументоване API;
c) всі модулі взаємодіють тільки з API інших модулів без використання мережевих протоколів; в деяких випадках модуль може надавати частковий доступ до свого API через мережу;
d) кожен модуль може використовувати власне сховище даних та має власну конфігурацію;
e) кожен модуль може знаходитись або в репозиторії самої системи, або мати окремий репозиторій;
f) можливість здійснити автономний перезапуск модулю або масштабувати окремий модуль (незалежно від всієї системи).

Так як кожний компонент має власні деталі реалізації, то залежність різних компонентів один від одного достатньо слабка, тому команди розробників можуть працювати автономно над окремими компонентами системи [4]. При цьому, так як процеси ініціалізації та запуску централізовані (виконуються загальною частиною), необхідно забезпечити стабільність та надійність роботи API між компонентами.

Застосування модульної, а не монолітної архітектури є більш ефективним рішенням для складних, багаторівневих систем, так як дозволяє мінімізувати всі недоліки моноліту, про які йшла мова у розділі 2.2.1.

Але в деяких випадках модульні рішення не надають достатньої гнучкості системі та автономності її модулів. В першу чергу, це стосується гіпер-масштабних

застосунків по типу Google, Netflix, Uber і т.п. В цьому випадку можна використовувати мікросервіси, які дозволяють використовувати різні стеки технологій при розробці різних мікросервісів, а також забезпечують можливість незалежного розгортання, масштабування та ініціалізації окремих мікросервісів.

1.2.3. Мікросервісна архітектура

Мікросервісна архітектура – підхід до розробки програмного забезпечення, в якому програмне рішення складається з декількох невеликих та автономних компонентів (мікросервісів), які розбивають великі завдання на певну кількість незалежних баз коду, що мають власну бізнес-логіку та базу даних.

Кожен мікросервіс виконує конкретну функцію та може бути змінений, масштабований чи розгорнутий окремо від інших мікросервісів.

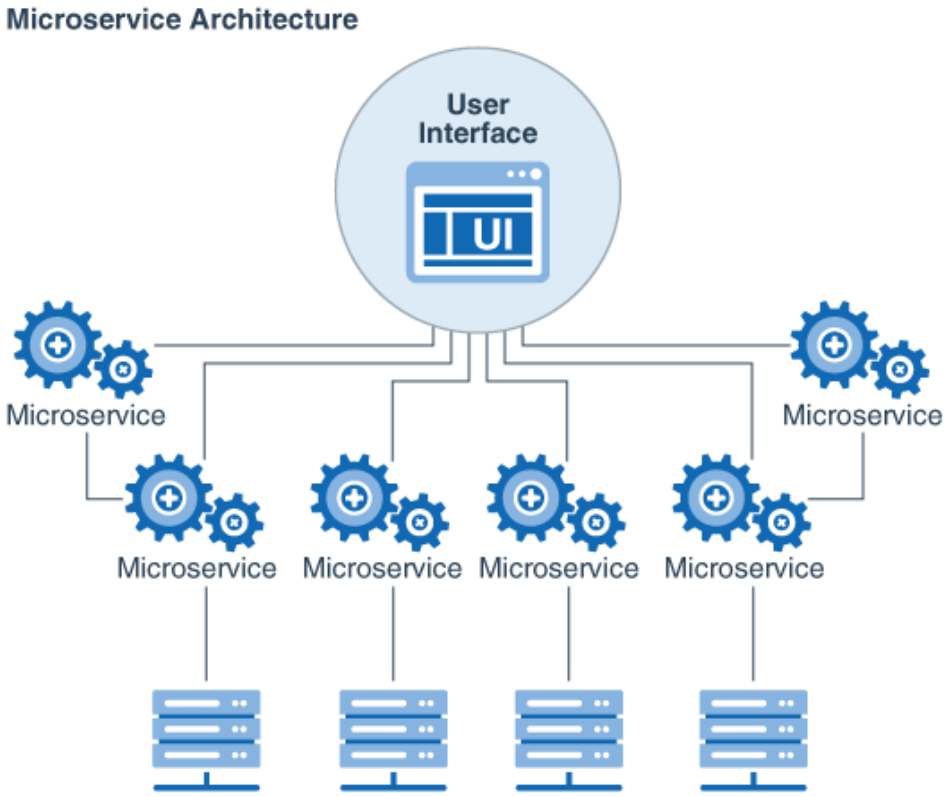


Рис.2 Візуальне подання мікросервісної архітектури

Особливість даної архітектури дає розробникам більшу гнучкість та полегшує процес модифікації та розвитку застосунків. Використання такої архітектури доцільно для керування конфігураціями компонентів систем, що мають велику кількість пристроїв, так як розбиття об’ємних задач на невеликі функціональні блоки надає додаткову гнучкість та простоту реалізації.

Основними перевагами мікросервісної архітектури є:

a) висока швидкість запуску системи та її розгортання - кожен окремих сервіс системи може бути розгорнутий незалежно від інших, які продовжуватимуть працювати;

b) гнучкість у виборі елементів системи - для різних мікросервісів є можливість використання різних мов програмування, технологій зберігання даних та інших елементів побудови системи;

c) можливість масштабування та повторно тестування окремих частин (сервісів) системи, а не всієї системи в цілому;

d) оновлення як системи в цілому, так і її окремих частин (при цьому інші компоненти системи можуть продовжувати безперебійну роботу);

e) легкість обслуговування та можливість незалежного розгортання окремих функцій – можна легко впроваджувати нові функції та повертатися до попередньої конфігурації при виникненні будь-яких проблем з їх реалізацією;

f) висока стійкість системи - проблеми з одним мікросервісом не поставлять під загрозу надійність функціонування всієї системи;

g) краща організаційна структура мікросервісів - кожен з них виконує свою специфічну функцію і не залежить від роботи інших сервісів, тому принципи їх роботи є більш зрозумілими, а тестування та підтримка - простими.

Недоліками та обмеженнями мікросервісної архітектури є:

a) процеси створення та розгортання розподіленої системи є досить складними;

b) зростання інфраструктурних витрат - кожний окремий мікросервіс може потребувати індивідуального комплекту тестів, інструкцій з розгортання, інструментів моніторингу, інфраструктури хостингу та інше [5];

c) ризик ускладнення параметрів системи – коли для різних мікросервісів будуть використовуватися різні мови програмування або засоби моніторингу.

Незважаючи на труднощі у впровадженні мікросервісної архітектури, вона є оптимальним інструментом для реалізації складних корпоративних додатків в умовах їх постійного розвитку. Використання мікросервісної архітектури дозволяє максимально пришвидшити реагування системи на зміну зовнішніх умов, що додає більше гнучкості та конкурентоспроможності, незважаючи на додаткову складність проектів, які будуються на мікросервісній архітектурі.

На даний момент багато великих корпорації (Netflix, Spotify, PayPal, Uber, Amazon) успішно використовують саме цей архітектурний шаблон для розвитку своїх рішень.

1.3. Платформи для побудови СККМ

Для побудови системи керування конфігураціями компонентів мережі підприємства можна використовувати наступні платформи:

- віртуальні машини на одному або декількох хостах;
- контейнери;
- Docker-контейнери.

Всі ці платформи можуть бути використані для віртуалізації середовища розгортання СККМ, проте необхідно зважати на особливості та обмеження кожної з них.

1.3.1. Віртуальні машини (ВМ)

Віртуальна машина (ВМ) — це програмна емуляція фізичної комп'ютерної системи, що дозволяє кільком операційним системам працювати на одній фізичній машині, при цьому кожна ОС працює у своєму власному ізольованому середовищі [6].

Віртуальна машина імітує апаратне забезпечення:

- центральний процесор;
- пам'ять;
- пристрої зберігання даних;
- мережеві інтерфейси тощо.

Це дозволяє гостьовій операційній системі, запущеній всередині віртуальної машини, взаємодіяти з нею так, ніби вона працює безпосередньо на фізичному пристрої.

За допомогою процесу віртуалізації, що використовує ВМ, можна створювати декілька екземплярів операційних систем на одній фізичній машині.

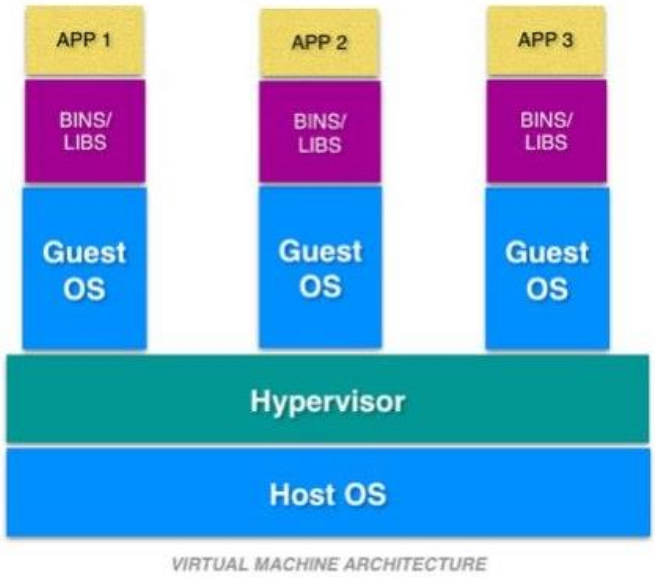


Рис.3 Візуальне подання віртуальної машини

Віртуальні машини можна використовувати в архітектурі мікросервісів, наприклад коли конфігурація містить системи чи програми, які неможливо легко контейнеризувати, або коли існують певні вимоги безпеки чи відповідності, які вимагають використання віртуальних машин. В цьому сценарії кожен мікросервіс буде працювати на власній віртуальній машині, що забезпечить більшу ізоляцію та безпеку між окремими компонентами та службами. А гіпервізор керуватиме віртуальними машинами, дозволяючи кільком операційним системам і програмам працювати на одній фізичній машині.

Технологія віртуальної машини найчастіше використовується в монолітних архітектурах, де весь застосунок розгортається в одній віртуальній машині. Цей підхід дозволяє легко керувати програмою та її залежностями, забезпечує легшу масштабованість і керування ресурсами.

Прикладами СКККМ, які можуть працювати на віртуальних машинах, є SolarWinds NCM, ManageEngine Network Configuration Manager, NetBrain. Ці системи можна інсталювати на віртуальних машинах, наприклад VMware або Hyper-V. Але використання віртуальних машин для мікросервісної архітектури у більшості випадків не є оптимальним. Найбільш поширеним варіантом платформи для мікросервісної архітектури є Docker-контейнери.

1.3.2. Контейнери

Контейнер - одиниця програмного забезпечення, яка дозволяє запускати процеси ізольовано від інших процесів на поточному хості, пакуючи код та всі його залежності, налаштування та бібліотеки середовища [7]. Що всі контейнери, розташовані на одній машині, спільно використовують ядро операційної системи хоста.

Таким чином, за допомогою технології контейнеризації стає можливим створення одного контейнеру (пакету), який буде працювати на будь-якому приладі та у будь-якому середовищі. Один контейнер може бути використаним як для запуску чогось невеликого (наприклад одного мікросервісу), так і для розгортання цілого застосунку.

До головних переваг контейнерів відносяться:

- високий показник відмовостійкості: контейнеризовані процеси є ізольованими один від одного, некоректна робота одного процесу ніяк не впливає на інші компоненти системи;
- ефективна масштабованість: процес додавання нових контейнерів на одну машину є легким та не потребує значних ресурсів;

- портативність:можна розгортати застосунки у різних середовищах без необхідності редагування програмного коду чи залежностей;
- гнучкість: використання контейнерів дає можливість швидше та ефективніше переносити застосунок або його окремі частини у інші середовища (наприклад, з фізичного пристрою у хмарне середовище та навпаки).

Для підвищення ефективності та структурованості робочого процесу з контейнеризованими застосунками зазвичай використовуються спеціальні системи керування контейнерами, які автоматизують створення, розгортання, масштабування та видалення контейнерів, тобто забезпечують керування та підтримку програмного забезпечення.

Найбільш поширеною системою керування контейнерами є платформа Docker, що оптимізує процес розгортання та створення застосунків, пакуючи програмне забезпечення в стандартизовані блоки, що називаються Docker-контейнери.

1.3.3. Docker-контейнери

Docker-контейнер — це автономний і виконуваний пакет програмного забезпечення, який містить усе необхідне для запуску програми:

- код;
- системні бібліотеки;
- системні інструменти;
- середовище виконання.

Docker-контейнер використовує технологію контейнеризації, яка дозволяє працювати декільком ізольованим контейнерам на одній хост-операційній системі, використовуючи спільне ядро з іншими контейнерами [6].

Docker-контейнери дуже портативні та можуть бути розгорнуті в різних середовищах (локальні машини, хмарні екземпляри або центри обробки даних) без необхідності додаткового налаштування. Вони забезпечують простий і ефективний спосіб зберігання, розповсюдження та запуску програм, а також керування залежностями та конфліктами між різними програмними компонентами. Також docker-контейнери дуже легко масштабуються, так як необмежена кількість контейнерів може бути запущеною з одного docker-образу (виконуваний пакет, що містить в собі всі необхідні компоненти для запуску застосунку).

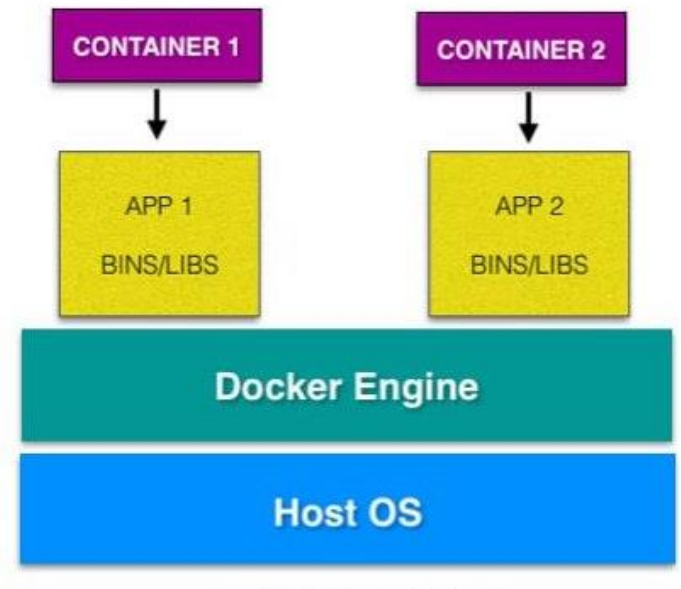


Рис.4 Docker - архітектура

Docker-контейнери можна створювати, керувати ними та запускати за допомогою механізму Docker, легкого середовища виконання та інструменту керування контейнерами.

Docker-контейнери, в свою чергу, не віртуалізують апаратне забезпечення - замість цього вони віртуалізують операційну систему. Докер-контейнери містять саму програму та її залежності, спільно використовуючи ядро операційної системи хоста, що дозволяє їм бути набагато ефективнішими за віртуальні машини.

Використання Docker-контейнерів є обґрунтованим та доцільним для моделювання СККККМ, які постійно знаходяться під динамічним навантаженням, через різні фактори:

- додавання або видалення пристроїв,
- термінова потреба в оновленні програмного забезпечення,
- незапланована зміна конфігурації мережі та інші операційні вимоги.

Тому СККККМ повинні розроблятися з урахуванням масштабованості, продуктивності та ефективного балансування навантаження. Завдяки відсутності гіпервізора, Docker-контейнери потребують менше пам'яті та ресурсів ЦП, дозволяють швидше розгортати та масштабувати СККККМ систему, легко копіюються та переміщуються між середовищами, тому є оптимальним рішенням для ефективного використання апаратних ресурсів СККККМ, оскільки вони потребують менше пам'яті та ресурсів ЦП, ніж віртуальні машини. Також використання Докер-контейнерів дозволить швидше розгортати та масштабувати СККККМ систему, та полегшить керування, так як Докер контейнери можна легко копіювати та переміщувати між середовищами.

При всіх перевагах Docker-контейнерів вони мають певні вразливості, якщо використовуються для СККККМ з високими вимогами до надійного захисту від можливих зловмисницьких сценаріїв, таких як несанкціонований доступ до управління конфігураціями пристроїв.

Один з головних ризиків для безпеки при використанні Docker-контейнерів обумовлений необхідністю впровадження для них власної мережевої інфраструктури, яка дозволяє їм зв'язуватися один з одним, або з хостами для спільного використання ресурсів та даних [8].

Незважаючи на те, що сучасні контейнерні мережі спрямовані на оптимізацію та стандартизацію процесу передачі даних між контейнерами, шляхом створення

ізольованих просторів, що обмінюватись даними безпечно та більш ефективно, у застосунках великого масштабу мережа може бути дуже складною. Це підвищує ризик наявності безпекових вад у мережі та потенційно може дозволити зловмиснику використовувати платформу для своїх цілей.

1.3.4. Використання систем управління та координації Docker-контейнерами: Docker Compose та Kubernetes

Для більш ефективного управління та координації роботи Docker-контейнерів можна також застосовувати наступні системи оркестрації:

- Docker Compose;
- Kubernetes.

Оркестрація - це автоматизоване управління та координація кількох контейнерів або служб у розподіленій системі, що включає в себе керування всім життєвим циклом контейнера (надання, розгортання, масштабування, балансування навантаження та відновлення після збою).

Переваги оркестровки контейнерів включають:

a) масштабованість - оркестрування дозволяє горизонтально або вертикально масштабувати додатки, забезпечуючи підвищену доступність і надійність;

b) ефективність використання ресурсів - організовуючи контейнери, стає легше ефективно використовувати ресурси, що призводить до економії коштів;

c) автоматизація - інструменти оркестрування автоматизують багато ручних завдань, пов’язаних із керуванням та розгортанням контейнерів;

d) висока доступність - оркестрування контейнерів забезпечує автоматичне перемикання після відмови та механізми відновлення;

е) портативність - розгортати контейнери та керувати ними можна в різних середовищах (загальнодоступні та приватні хмари, локальні центри обробки даних, гібридні середовища).

Загалом оркестрація контейнерів надає організаціям можливість розгортати й керувати великомасштабними контейнерними програмами з більшою ефективністю, надійністю та гнучкістю.

Docker Compose

Docker Compose — це програма - надбудова над докером, написана на Python, яка дозволяє запускати кілька взаємодіючих контейнерів одночасно та маршрутизувати потоки даних між ними, використовуючи маніфести, в даному випадку у форматі YAML. Docker Compose автоматично створює мережу для застосунку та забезпечує спілкування контейнерів в ньому один з одним. Docker Compose дозволяє визначати змінні середовища та інші параметри конфігурації застосунку, що полегшує налаштування та розгортання програми в різних середовищах. Важливо зазначити, що його використання є доцільним, якщо для забезпечення функціонування системи використовується кілька різних сервісів.

Docker Compose використовується тільки для роботи на одному хості, так як він може керувати лише контейнерами, які працюють на тій самій машині, де є встановленим [9]. Проте, застосовуючи зовнішні оркестратори (наприклад Kubernetes або Docker Swarm), Docker Compose може бути використаним для керування контейнерами на кількох хостах.

Kubernetes

Kubernetes — це платформа з відкритим кодом для оркестрації контейнерів, яка автоматизує розгортання, масштабування та керування контейнерними програмами.

Kubernetes має високу масштабованість і може керувати тисячами контейнерів, що працюють на кількох серверах. Він надає ряд функцій для забезпечення високої доступності [10], яка є дуже важливим аспектом СККККМ, включаючи автоматичне перемикання після відмови, самовідновлення та балансування навантаження. Однією з ключових переваг Kubernetes є його гнучкість. Він підтримує широкий спектр середовища виконання контейнерів, включаючи Docker container.

Kubernetes може бути потужним інструментом для керування та розгортання системи СККККМ. Його здатність автоматизувати розгортання та масштабування служб і компонентів, а також можливості моніторингу та самовідновлення можуть гарантувати, що система СККМ буде завжди доступною для користувачів.

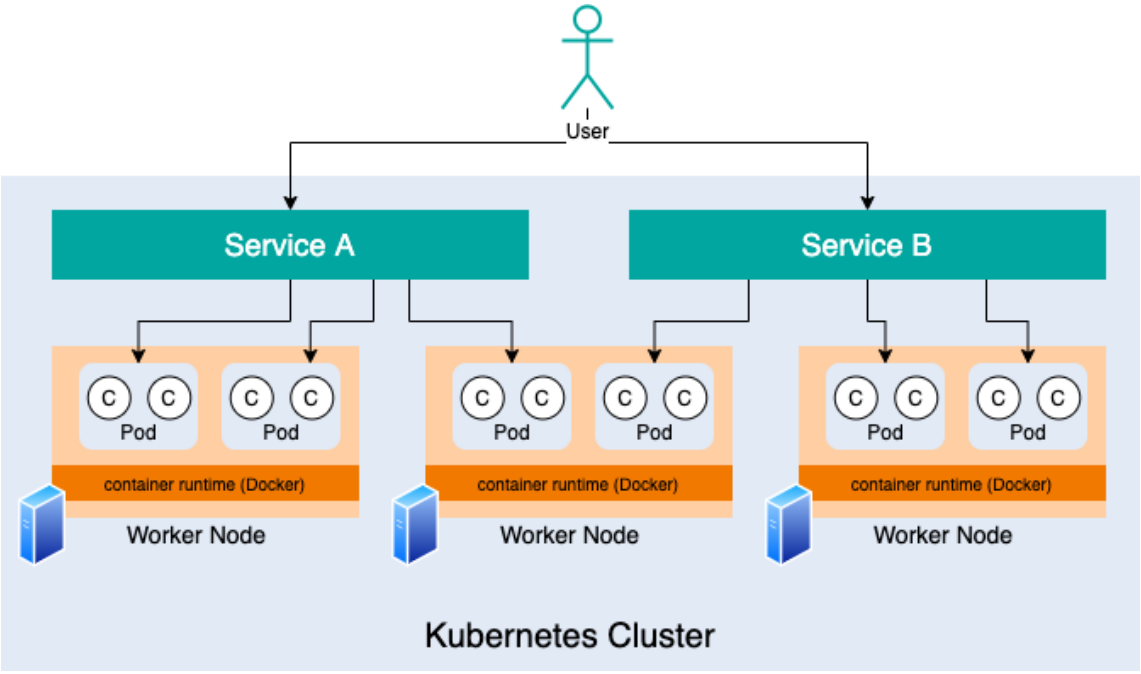


Рис.5 Кластер-Kubernetes

Кластер-Kubernetes, що розгортає контейнеризовані застосунки, складається з nodes, які запускають контейнерні програми та складаються з таких компонентів:

- Kubelet - агент, що відповідає за керування контейнерами певного вузла шляхом взаємодії з API;
- Container Runtime - програмне забезпечення, яке застосовується для запуску контейнерів на вузлі (Docker, CRI-O);
- Pods - найменші одиниці програмного забезпечення, які можна розгортати у Kubernetes, та які використовуються для групування одного або декількох контейнерів з однаковим життєвим циклом. Тобто всі контейнери одного pod запускаються та зупиняються одночасно, спільно використовують мережу та ресурси зберігання, використовують спільний мережевий простір імен;
- Kube-Proxy - агент, що балансує мережеве навантаження між nodes в кластері.

Існує два типи nodes - master nodes та worker nodes - що виконують різні функції для коректної роботи Kubernetes кластера.

Worker nodes - запускають середовища виконання контейнерів та забезпечують зв'язок із Control Plane, використовуючи Kubelet; відповідають за моніторинг та збір показників контейнерів; запускають Kube-Proxy для керування мережею.

Master nodes у Kubernetes - це панель управління (Control Plane), що координує діяльність worker nodes та керує загальним станом кластера.

У підсумку стає зрозумілим, що вибір між Docker Compose і Kubernetes залежить від складності застосувань та масштабу розгортання. Використання Kubernetes є доцільним для великих компаній або проектів, де потрібне гнучке налаштування всієї інфраструктури, тоді як Docker Compose є хорошим вибором для невеликих розгортань із простішими конфігураціями.

1.4. Вибір компонентів системи керування конфігураціями компонентів мережі підприємства

Для побудови ефективної системи керування конфігураціями компонентів мережі підприємства (СКККМ) необхідно визначити параметри її основних компонентів:

- високонадійне сховище даних та система контролю версій;
- підсистема автоматизованого збору конфігурацій;
- база даних;
- веб-інтерфейс;
- API.

1.4.1. Сховище даних та система контролю версій

Для зберігання даних СКККМ можна використовувати різні варіанти сховищ даних та окремі або вбудовані системи контролю версій. Основною вимогою при виборі є надійність, зручність та простота використання.

Для СКККМ застосовують різні види сховищ даних: дискова підсистема, серверний кластер, хмарне сховище, об’єктне сховище тощо.

В першу чергу перспективним є використання об'єктних сховищ, в яких кожен об'єкт є цілісним та самодостатнім сховищем, що містить описову інформацію, пов'язану з об'єктом (метадані), дані і унікальний ідентифікаційний номер, який дозволяє легко та швидко знаходити необхідні об'єкти у сховищі. Сховища об'єктів можуть бути розподіленими між кількома географічними регіонами, забезпечують майже необмежену масштабованість, мають високу доступність та можуть обробляти як структуровані, так і взагалі неструктуровані дані.

В якості системи контролю версій для складних та масштабних СККККМ досить часто використовується Git, який є розподіленою системою контролю версій та надає дуже обширний функціонал:

- керування змінами в коді (розгалуження, злиття);
- контроль версій;
- колаборація розробників в роботі над продуктом;
- керування великими кодовими базами;
- відстеження помилок;
- автоматизація процесу розробки, тощо.

Але, якщо для СККККМ настільки обширний функціонал не є необхідним, то використання Git не є оптимальним рішенням через його складність. При створенні СККККМ середнього або меншого масштабу доцільно використовувати сховища даних із вбудованими в них системами контролю версій, наприклад Amazon S3 або опенсорсне об'єктне сховище MinIO, сумісне з Amazon S3 API.

Amazon S3

Amazon S3 - сервіс зберігання об'єктів, розроблений Amazon Web Services (AWS), який надає широкий спектр інструментів та функцій для керування та зберігання даних.

Об'єктне сховище Amazon S3 має управління версіями, яке дозволяє зберігати, отримувати та відновлювати кожну версію будь-якого об'єкта, що зберігається [11]. Ця особливість Amazon S3 є дуже корисною для СККККМ, так як при роботі з СККККМ часто виникають проблемні ситуації, коли пристрої в мережі необхідно відкотити до попередньої конфігурації, чи провести аналіз попередніх конфігурацій.

Додатковим функіоналом, корисним для СККККМ, є можливість гнучко організувати колаборативну роботу адміністраторів, використовуючи функціонал

S3, а саме вбудовані інструменти для обміну конфігураціями між різними фізичними місцями чи групами користувачів та спільної роботи над поточною конфігурацією мережі.

У разі аварійної ситуації, ефективним є використання вбудованих можливостей реплікації та відновлення системних даних S3. Це робить S3 надійним сховищем для систем по типу СККМ, для яких втрата попередніх версій даних є критичною.

MinIO

MinIO - це мульти-хмарне обєктне сховище, що є сумісним з Amazon S3 API, нативним для Kubernetes та може бути розгорнутим локально чи на будь-якій хмарі [12]. Цей продукт є ефективним для створення розподіленої інфраструктури збереження даних.

До основних переваг використання MinIO можна віднести:

- високий рівень безпеки - досягається тим, що MinIO використовує шифрування на стороні сервера, політики контролю доступу та шифрування SSL/TLS;
- висока доступність - інструмент автоматично реплікує дані, що гарантує постійний доступ до них, навіть у разі апаратних збоїв чи інших проблем;
- масштабованість - MinIO динамічно пристосовується до масштабу сховища, який необхідний користувачу;
- продуктивність - платформа має високу пропускну здатність та низький рівень затримки навіть при великих навантаженнях.

Одним з розповсюджених сценаріїв роботи з MinIO є використання його в якості об’єктного сховища для Kubernetes. Це дозволяє контейнеризованим програмам ефективно зберігати та отримувати неструктуровані дані через

S3-сумісний API, до якого можливо доступитися за допомогою AWS SDK або інших S3-сумісних бібліотек.

Таким чином MinIO є дуже гнучким, легким в плані інтеграцій та надійним інструментом для зберігання та керування даними в контейнеризованих програмах.

1.4.2. Підсистеми автоматизованого збору конфігурацій: Ansible та Puppet

При роботі з дуже великими та розподіленими СККМ доречним є використання систем автоматизованого збору конфігурацій, так як мануальна підтримка конфігурацій всіх мережевих девайсів призводить до збільшення ризику помилки та втрати ефективності і швидкості роботи. Використання систем автоматизованого збору конфігурацій полегшує процес масштабування інфраструктури та ПЗ без необхідності збільшувати робоче навантаження на адміністраторів мережі чи їх кількість. Двома найпопулярнішими системами для автоматизованого збору конфігурацій є Ansible та Puppet.

Ansible

Ansible має відкритий код є популярним інструментом автоматизації, який надає широкий спектр модулів для різноманітних завдань, надаючи в тому числі функціонал для написання власних модулів [13].

Ansible є безагентним, тобто не вимагає встановлення ПЗ на керованих пристроях чи вузлах, використовуючи ssh-з'єднання для входу на пристрої чи вузли, що робить його розгортання доволі легким, а керування простішим [14].

Для управління конфігураціями та розробки сценаріїв використовуються playbooks, що є впорядкованими наборами завдань. Playbooks виконуються на обраних хостах та визначають роботу, яка має бути виконаною. Playbooks пишуться за допомогою мови YAML, що є досить легкою та інтуїтивно зрозумілою.

Таким чином playbooks можна використовувати для налаштування віддалених пристроїв, запуску асинхронних та синхронних завдань, забезпечення злагодженої роботи всієї мережі.

Ansible має два типи вузлів: control node - машина, з якої запускаються інструменти Ansible CLI; та managed nodes(hosts) - цільові пристрої, які керуються за допомогою Ansible (зазвичай Ansible не є встановленим на managed nodes).

Типово Ansible має push-архітектуру, проте використовуючи ansible-pull можна отримати віддалені копії Ansible на кожному з managed nodes. Це дозволяє інвертувати типову push-архітектуру в pull-архітектуру, та мати майже безмежний потенціал масштабування.

Puppet

Puppet - інструмент керування конфігураціями, що використовується для автоматизації та централізації процесу управління налаштуваннями. Puppet використовує клієнт-серверну архітектуру [15], де одна система у будь-якому кластері працює як сервер (puppet master), а інша працює як клієнт на вузлах, що називаються slave.

Так як Puppet використовує декларативну мову, то з його допомогою можна визначити бажаний стан інфраструктури за допомогою коду, який потім автоматично застосовується до систем.

Puppet використовує доменно-орієнтовану мову (DSL), яка базується на мові програмування Ruby та забезпечує простий й інтуїтивно зрозумілий синтаксис для опису бажаного стану інфраструктури.

Puppet забезпечує ідемпотентність інфраструктурної конфігурації. Тобто, навіть якщо конфігурація запускається кілька разів, кінцевий результат буде кожен раз однаковим. Це допомагає уникнути дрейфу (відхилення) конфігурацій та забезпечує їх узгодженість.

Для керування компонентами інфраструктури Puppet використовує RAL - абстракцію, яка відокремлює різноманітні ресурси (файли, пакети, служби та користувачів) від своєї реалізації для більш зручного керування ними. За допомогою цієї абстракції, незалежно від ресурсів і ОС цільової машини, Puppet автоматично налаштовує конфігурацію.

1.5. Огляд існуючих інструментів для керування конфігураціями пристроїв в мережі: RANCID, Oxidized

RANCID (Really Awesome New Cisco confIg Differ) — безкоштовний інструмент із відкритим кодом, призначений для керування конфігурацією мережевих пристроїв, в основному таких як маршрутизатори, комутатори та брандмауери [16]. RANCID був розроблений компанією Shrubbery Networks у 1997 році і наразі є одним із найбільш використовуваних інструментів.

Його перевагами є універсальність та здатність отримувати конфігурації з різних типів мережевих пристроїв (Cisco, Juniper, HP тощо). Цей інструмент може ефективно використовуватися в неоднорідних високомасштабних мережевих середовищах, які складаються з великої кількості різних пристроїв, так як він здатний контролювати конфігурації на значній кількості пристроїв одночасно, зберігаючи та відстежуючи всю історію конфігурацій та їх змін. Це полегшує роботу мережевих адміністраторів щодо швидкого виявлення та вирішення проблем, пов’язаних з налаштуваннями пристроїв.

RANCID використовує простий текстовий формат для зберігання та отримання конфігурацій, що полегшує його використання та інтеграцію з іншими інструментами. Він також підтримує різноманітні механізми автентифікації, включаючи SSH, Telnet і SNMP, що забезпечує безпечний доступ до мережевих пристроїв.

Після встановлення RANCID на хост виконується налаштування списку груп (набір мережевих пристроїв). Групи дозволяють розділити пристрої згідно архітектурних особливостей мережі на окремі множини, сповіщаючи про внесення змін чи проблеми в роботі пристроїв, при цьому сповіщення отримують тільки користувачі, які є адміністраторами даної групи.

Також корисним функціоналом є зберігання всіх конфігурацій пристроїв із групи в окремому каталозі, що дозволяє краще структурувати інформацію.

Після створення груп, мережевий адміністратор повинен налаштувати доступ до пристроїв, редагуючи та додаючи дані стосовно доступу у файлі .cloginrc, який знаходиться в домашньому каталозі користувача. Виконавши ці початкові налаштування необхідно виставити часовий проміжок, через який користувачі будуть отримувати сповіщення від RANCID у разі зміні конфігурацій пристроїв.

Після запуску команди rancid-run кожного разу, коли RANCID виявлятиме зміну, він створюватиме нову версію та зберігатиме те, що змінилося з моменту останнього запуску у каталозі configs кожної групи.

Додатково, RANCID оптимізує роботу у випадках, коли конфігурація має бути змінена для великої кількості пристроїв: створюється файл, у якому зберігаються всі необхідні команди, що мають бути виконаними для певного типу пристроїв. Таким чином зникає потреба конфігурувати кожний пристрій окремо, адже стає достатнім лише один раз визначити конфігурацію, автоматично застосувавши її до всіх пристроїв.

Oxidized — це інструмент резервного копіювання конфігурацій мережевих пристроїв, розробники якого позиціонують цей продукт як заміну RANCID. Oxidized має ширший функціонал та використовує більш сучасні та актуальні технології та розроблений з використанням модульної архітектури: кожна функціональна одиниця оформлена в окремий модуль, що надає користувачам можливість легко адаптувати функціонал продукту для їх цілей та потреб, а розробникам спрощує процес інтегрування нового функціоналу [17].

На даний момент продукт складається з 4 модулів:

- **Source** завантажує список вузлів і застосовує додаткові параметри (name, model, input, output, prompt) до них для кожного девайсу, підтримуючи на даний момент всі типи файлів з одним записом на рядок (наприклад router.db).
- **Input** взаємодіє з елементами мережі, за потреби емулюючи CLI-користувача, для отримання конфігурацій з нодів. На даний момент Oxidized підтримує ssh, telnet, ftp, tftp, http.
- **Output** модуль призначений для зберігання конфігурацій, маючи реалізованими git та file.
- **Model** описує, які дані мають бути отриманими від вузла, і додатково обробляє вихідні дані, використовуючи різні методи для роботи з конфігураціями пристроїв.

Основні особливості інструменту Oxidized:

- використовує потоки для кожного окремого пулу конфігурацій;
- автоматично пристосовується до кількості потоків, надаючи можливість користувачам встановлювати ліміти на кількість потоків;
- написаний на Ruby;

- моделі є Ruby-класами;
- реалізує DSL для взаємодії та зручності;
- надає інструментарій для роботи з складними контекстно-залежними моделями, використовуючи допоміжні методи та змінні.

1.6. Висновки до розділу 1

Наразі існує досить багато ефективних інструментів та типових рішень для реалізації СККККМ з урахуванням різних масштабів корпоративних мереж та індивідуальних потреб споживачів. Більшість з цих інструментів є універсальними і можуть комбінуватися один з одним, що дозволяє отримувати широкий спектр можливостей для роботи з конфігураціями мережевих пристроїв різних типів.

Такі готові та універсальні продукти можуть бути ефективно використані спеціалістами під час моделювання та імплементації власної СККККМ з метою оптимізації часу моделювання окремих компонентів та СККККМ в цілому та для автоматизації виконання окремих та комплексних задач, які вимагаються від СККККМ.

Розділ 2. Структурна розробка власного рішення

В межах цієї кваліфікаційної роботи структурно розроблена система керування конфігураціями пристроїв в мережі. При цьому мережа містить значну кількість пристроїв, розподілених у просторі, та потенційно може вимагати глобального масштабування.

Основними вимогами до СККККМ є її надійність, можливість асинхронного управління конфігураціями, високий рівень автономності.

СККККМ має модульну архітектуру, що полегшує її масштабування у разі потреби, та дозволяє користувачам з різними рівнями доступу управляти конфігураціями пристроїв, які адмініструються цією СККККМ та проводити детальний аналіз конфігурацій.

Одним з найважливіших функціоналів системи є регулярне створення копій поточних конфігурацій на пристроях. Також система дозволяє користувачам аналізувати наявність, стан і особливості конфігурації кожного пристрою.

Якщо користувачі роблять запит до системи для отримання структурованої інформації про окремі пристрої чи хочуть отримати звіти про різницю в конфігураціях пристроїв, то система надає таку інформацію.

Далі наведена структурна схема СККККМ.

2.1. Структурна схема СККККМ

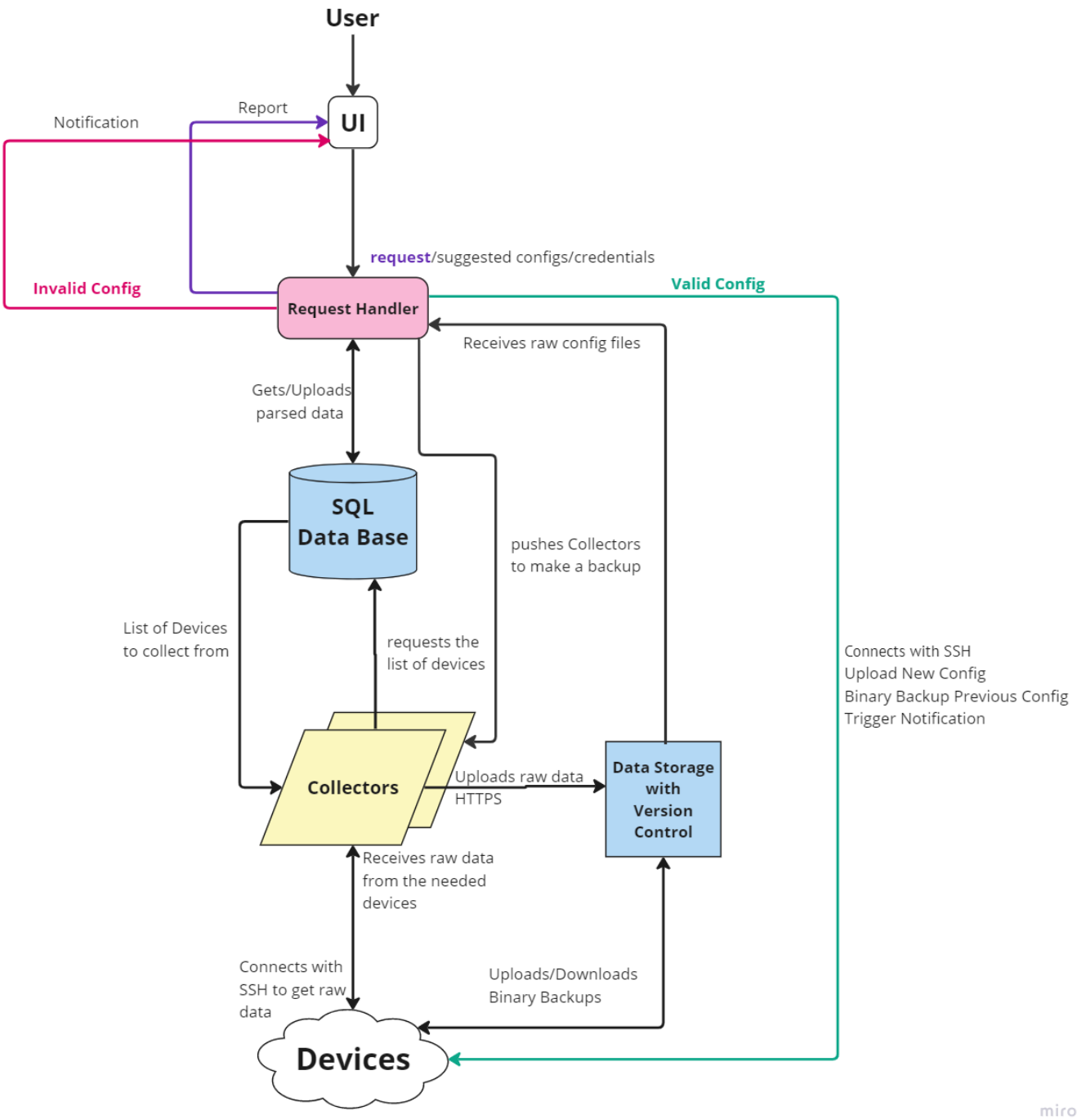


Рис.6 Структурна схема власного рішення

2.2. Опис модулів СККККМ

2.2.1. UI: користувацький веб-інтерфейс

В схемі використовується веб-інтерфейс, який дозволяє користувачам швидко отримати доступ до застосунку незалежно від свого місцезнаходження та з будь-якого пристрою. Перевагами веб-інтерфейсу також є ефективна масштабованість без додаткових ресурсів на стороні клієнта, полегшений процес обслуговування та оновлення веб-інтерфейсів (немає необхідності вносити зміни для кожного клієнта, достатньо змінити чи оновити код застосування на стороні сервера).

В якості веб-серверу, особливості та функціональність якого найбільше задовольняють вимоги до цієї СККККМ доцільно обрати Apache Server. Apache Server складається з модулів, які можна конфігурувати вручну, також цей сервер зможе оброблювати всі різні типи запитів зі сторони клієнта [18].

Apache Server не пристосований та зазвичай не використовується для систем, що знаходяться під постійним навантаженням з високим рівнем трафіку, проте для розроблюваної системи це не є проблемою, так як вона не відноситься до такого типу систем.

2.2.2. Request Handler

Request Handler – це компонент, що обробляє запити, що надходять від UI, отримуючи та маніпулюючи даними та файлами конфігурацій з DB та Data Storage.

Компонент Request Handler виконує нижче перелічені функції.

Функція «User Access Control»: керування доступом користувачів до мережевих пристроїв і забезпечення доступу для внесення змін лише авторизованим користувачам, надання детальних журналів аудиту всіх дій користувачів.

Ця функція є надважливою, так як напряму корелює з безпекою всієї системи, що для СККККМ є принциповою умовою. Доступ до пристроїв та їх конфігурацій користувачами, що не мають необхідних прав, може призвести до некоректної роботи чи припинення функціонування всієї мережі.

Контроль доступу на основі ролей (Role-Based access control - RBAC) є дуже доречним рішенням для цього компоненту, так як RBAC легко та швидко масштабується, даючи змогу керувати великою кількістю груп користувачів. Також завдяки RBAC можна дуже детально визначати повноваження та права доступу користувачів, що суттєво зменшує ризик витоку даних чи несанкціонованого доступу до інформації [19]. RBAC дозволяє створювати специфічні ролі з чітко визначеними задачами та дозволами, що є перевагою для ефективності та прозорості структури розподілення обов’язків між мережевими адміністраторами в компанії.

Функція «Analyze Configuration`s Validity»: функція запобігання завантаженню некоректних конфігурацій на пристрої.

СККККМ повинна мати функціонал перевірки конфігурації на правильність, що забезпечить зниження ризику виникнення проблем у роботі мережі через помилку адміністратора. В момент первісного конфігурування пристрою конфігурація має пройти повну перевірку на коректність, проте при подальших оновленнях немає потреби перевіряти всю конфігурацію заново: достатнім буде проаналізувати тільки внесені зміни та ті частини конфігурації, на які ці зміні можуть потенційно вплинути.

Наприклад: початково конфігурація інтерфейсу GigabitEthernet0/1 пристрою Switch1 мала вигляд:

«
switchport mode access
switchport access vlan 10
»

Оновлення конфігурації змінює режим access на trunk (що визначає, що порт може передавати трафік від декількох VLAN), що потребує в свою чергу перевірити чи було прибрано «switchport access vlan» атрибут, або чи було визначено, які VLAN будуть дозволені на trunk порті додаванням команди «switchport trunk allowed vlan» до конфігурації.

Функція «Backup»: відповідає за запускання процесу створення резервних копій конфігурацій пристрою на вимогу, шляхом надсилання інструкції з командою «почати роботу» до колекторів.

При завантаженні попередньо перевіреної на коректність конфігурації на пристрій функціонал надсилатиме Колекторам команду «почати збір поточних конфігурацій» та оновлюватиме відповідне поле в таблиці пристрою, яке визначає, чи треба колектору зібрати конфігурацію під час роботи на вимогу Request Handler.

Ця функція буде використовуватися кожного разу при зміні конфігурації пристроїв. В результаті створюватиметься резервна копія працюючої поточної конфігурації, і тільки після цього буде здійснено завантаження нової конфігурації. Це необхідно для того, щоб мінімізувати негативний чи непередбачуваний вплив нової конфігурації на роботу мережі, який може мати місце навіть незважаючи на те, що така нова конфігурація пройшла перевірку на валідність.

Варто зазначити, що процес створення резервних копій у змодельованій системі буде відбуватися регулярно, але буде реалізований функціонуванням колекторів без необхідності компоненту Request Handler брати у цьому участь.

Функція «Uploading New Config»: виконується у разі додавання нових конфігурацій на пристрої або при потребі оновлення вже існуючих.

Якщо після проведення аналізу конфігурації на валідність детерміновано, що вона є правильною, то відбувається підключення до пристрою із застосуванням протоколу SSH. Нова конфігурація завантажується на пристрій та запускаються функції створення резервної копії попередньої конфігурації та нотифікації користувачів щодо зміни конфігурації в мережі. Якщо ж конфігурація не пройшла перевірку на правильність – функціонал надсилає звіт про це на UI.

Серед всіх існуючих мережевих протоколів було обрано саме SSH, так як на даний момент він є одним із найбезпечніших для віддаленого доступу через використання шифрування даних з public key [20]. Такий варіант шифрування є більш безпечним, ніж використання паролів, та гарантує доступ до віддалених пристроїв тільки для авторизованих користувачів.

Функція «Notification of Users»: зважаючи на те, що мережа підприємства є динамічною структурою та у ній постійно відбуваються зміни, СККМ необхідно мати функціонал, що відповідатиме за сповіщення авторизованих користувачів системи про здійснені зміни до конфігурацій, проблеми роботи мережі, необхідність оновити конфігурацію.

Нотифікування має надсилати повідомлення, зважаючи на ролі та права доступу користувачів, аби усунути надсилання інформації, що є нерелевантною для певного кола працівників, чи має бути опрацьованою тільки працівниками з певними правами та задачами.

Функція «Compare Configs»: для полегшення процесу аналізу історії змін та оновлень конфігурацій пристроїв необхідно мати функціонал для порівняння поточної конфігурації з попередніми резервними копіями, а також для порівняння всіх наявних резервних копій між собою.

Функція «Inventory Management»: ця функція повинна забезпечити відстеження всіх пристроїв в мережі, включаючи основні параметри їх конфігурації, деталі апаратного забезпечення та фізичного розташування, систематизувати інформацію про інвентар та полегшити процес менеджменту і підтримки мережі у ефективно-робочому стані.

2.2.3. SQL Data Base

SQL Data Base зберігає інформацію про користувачів системи, пристрої та події внесення змін до конфігурацій (час, виконавець, пристрій і т.і.), що використовується та оброблюється Request Handler та контролерами.

На рис. 7 зображена ER-діаграма, яка відображає всі сутності, їх атрибути та зв’язки між ними:

- сутності зображено у вигляді таблиць із списками атрибутів (обов’язкові атрибути позначені символом «*», а не обов’язкові – «о»),
- відношення сутностей одної до іншої позначено стрілками (craw`s foot notation) [21],
- якщо атрибут є primary key або foreign key – це зазначається у дужках.

Для полегшення роботи з інформацією та підвищення рівня безпеки у системі, деталі користувача та його облікові дані було віднесено до окремих сутностей User_Details та Credentials відповідно.

Для авторизації та автентифікації користувача сутність Credentials має два обов’язкових атрибути: ім’я користувача для входу в систему та хешований пароль.

У User_Details зберігаються дані більш загального типу: пошта користувача, дата, коли користувача було додано до системи, його ім’я. Зазначені вище атрибути є обов’язковими. Також User_Details зберігає два необов’язкових атрибути - гендер та вік.

User_Details та Credentials мають обов’язкове з двох сторін відношення 1 до 1 із сутністю User, адже кожний користувач, якого було додано до системи, зобов’язаний мати тільки одні унікальні дані для входу та загальну інформацію про себе.

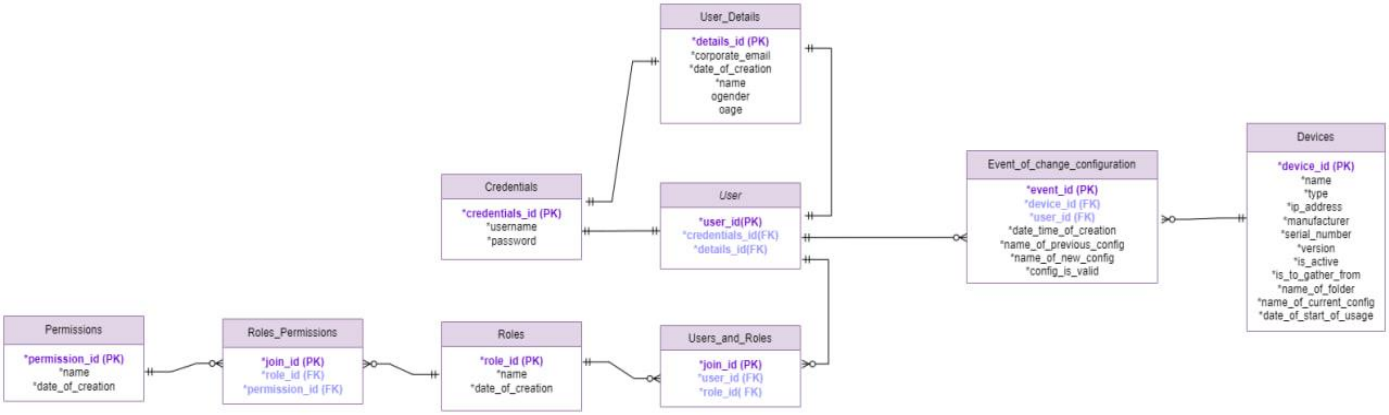


Рис. 7 ER-діаграма SQL Data Base

Сутності Roles та Permissions мають два обов’язкових атрибути - ім’я та дату створення. Roles зберігає в собі інформацію про всі наявні ролі, які можуть належати користувачам, Permissions – дані про всі можливі дозволи на проведення операцій та дій у системі.

Сутність Roles має необов’язкове відношення «багато до багатьох» із сутністю Permissions.

Це необхідно, так як декілька ролей можуть мати один і той же дозвіл на певну операцію в системі, і в той же час одна роль може мати багато дозволів.

Для реалізації зв’язку «багато до багатьох» створено додаткову зв’язуючу таблицю Roles_Permissions, що має власний ідентифікатор та ідентифікатори сутностей Permissions та Roles як foreign keys, знаходячись у відношеннях «один до багатьох» із сутностями, які вона пов’язує.

Сутності User та Roles мають необов’язковий зв’язок «багато до багатьох», так як один користувач може мати багато ролей, і одна і та ж роль, може бути призначеною різним користувачам. З цієї причини, за аналогією з реалізацією зв’язку ролей та дозволів, було створено проміжну таблицю Users_and_Roles.

Сутність Devices зберігає інформацію про кожний наявний в мережі пристрій:

- його ім’я, тип (маршрутизатор, кінцевий пристрій, т.і.), айпі-адресу;
- виробника, серійний номер, версію;
- флаг is_active, який є показником, чи на даний момент пристрій є активним та робочим;
- флаг is_to_gather_from, який є true, у разі, якщо конфігурація даного пристрою оновлюється та необхідно зібрати дані та зробити резервну копію поточної конфігурації;
- унікальне ім’я папки, в якій зберігається вся історія конфігурації цього пристрою;
- унікальне ім’я поточної конфігурації;
- дата та час, коли цей пристрій було додано до інфраструктури мережі.

Сутність Event_of_Change_of_Configuration зберігає інформацію про всі події, коли відбувалася спроба змінити конфігурацію на якомусь пристрої. Ця сутність має наступні атрибути:

- дата та час створення; унікальну назву конфігурації, що була на пристрої до зміни;
- унікальне ім’я нової конфігурації, що завантажилась на девайс;
- флаг config_is_valid, що true якщо конфігурація була визнана коректною;
- device_id та user_id, що є foreign keys.

У разі, якщо спроба зміни конфігурації була невдалою, то імена попередньої та поточної конфігурацій будуть співпадати.

Так як кожний користувач може створювати багато подій «зміна конфігурацій» на багатьох девайсах, але конкретна подія може бути створеною тільки одним користувачем для одного пристрою, то сутність Event_of_Change_of_Configuration знаходиться у відношенні один до багатьох із сутностями Devices та User.

2.2.4. Data Storage with Version Control

Цей компонент зберігає виключно файли конфігурацій та їх резервні копії.

Кожний тип пристроїв має окремі папки, в яких для кожного представника певного типу створюються власні папки, що називаються згідно до імен пристроїв.

Всередині кожної такої папки зберігаються:

- всі резервні копії конфігурацій поточного девайсу,
- конфігурації, що при спробі бути встановленими на пристрій пройшли перевірку на правильність,
- файл з поточною конфігурацією.

Кожний файл має назву у форматі «назваПристрою_датаЧас.conf», що дозволяє унікально ідентифікувати всі конфігураційні файли, тримаючи їх у хронологічному порядку.

2.2.5. Collectors

Collectors - це компоненти, що збирають дані з різних мережевих пристроїв та передають їх до сховища даних.

Загально-прийнятою практикою є створення окремих колекторів, що відповідають за зв’язок із пристроями певних типів (комутатори, маршрутизатори, брандмауери тощо).

З певним інтервалом колектори робитимуть запит до БД та отримуватимуть список пристроїв, з яких необхідно зібрати дані і передати їх до сховища із контролем версій.

Отримавши айпіадреси пристроїв, конфігурації яких мають бути зібраними, колектори дистанційно підключатимуться до визначених пристроїв за протоколом SSH та збиратимуть поточні конфігурації, формуючи файли. Після цього колектори завантажуватимуть зібрані файли до сховища даних із контролем версій, використовуючи протокол HTTPS, що шифрує дані протягом передачі.

У разі потреби зробити резервні копії конфігурацій певних пристроїв, Request Handler надсилатиме запит до колекторів та запускатиме процес збору та передачі даних до сховища.

Крім всіх зазначених вище компонентів мережа також включає всі наявні пристрої різного типу, конфігурації та коректність роботи яких має адмініструватися системою.

2.2.6. Переваги використання хмарних платформ для СККККМ систем

Основними перевагами використання хмарних платформ для розробки системи даного типу є наступні:

- масштабованість: так як кількість пристроїв у мережі підприємства є динамічною, можливість масштабувати ресурси відповідно до потреб, яку надають хмарні платформи, є необхідною;
- високий рівень безпеки: хмарні платформи надають розширені функції безпеки, що можуть бути використаними під час розробки системи [22];
- забезпечення високої доступності: хмарні платформи використовують резервні системи, що автоматично беруть на себе задачі тих систем, що вийшли з ладу, також хмарні платформи забезпечують аварійне відновлення у разі необхідності;
- полегшений супровід: мережеві адміністратори матимуть можливість отримувати доступ до СККККМ віддалено, крім того, через відсутність фізичних пристроїв, зникає потреба витрачати час та ресурси на їх розміщення, організацію належного для них простору та супровід.

2.3. Висновки до розділу 2

У цьому розділі було проведено структурну розробку власної СККККМ системи та описано функціональності та сценарії роботи кожного її компоненту:

- обґрунтована доречність використання саме веб-інтерфейсу та обрано веб-сервер;
- визначені всі функції центрального компоненту із логікою СККККМ - Request Handler;
- структурно розроблена БД: створено ER-діаграму для відображення сутностей та зв’язків між ними;
- визначено, що для збереження файлів конфігурацій доречним буде використання сховища даних із контролем версій;
- описаний сценарій роботи Колектору.

Також в цьому розділі було дано обґрунтування доцільності використання хмарних платформ для СККККМ систем.

Розділ 3. Розробка компоненту Collector

Колектори імплементують одну з найважливіших функціональностей СККККМ – створення резервних копій конфігураційних файлів для всіх пристроїв у мережі. Запуск колекторів для кожного типу пристроїв буде відбуватись регулярно, з інтервалом, який буде визначатись користувачами системи.

Використовуючи хмарну платформу, S3 bucket може бути використаним як сховище із контролем версій, SQL Database може бути реалізованою за допомогою сервісу RDS (Relational Database Service) платформи AWS.

Компонент колектор може бути програмою, що запускається із різними аргументами утилітою Crontab, що автоматично планує та запускає завдання [23].

У випадку змодельованої системи, маємо визначити дві cronjobs:

- одна буде виконуватись щогодини та збирати конфігурації з усіх активних пристроїв,
- друга буде запускатись щохвилини для того, щоб зробити резервну копію конфігурації для ново-сконфігурованого пристрою, у разі, якщо такий є (is_to_gather_from встановлено в true).

Cronjob, що запускається кожну хвилину відповідає за сценарій, в якому Request Handler надсилає запит до Колектору негайно зробити резервну копію поточної конфігурації певного пристрою, проте так як в цій роботі компонент Request Handler не розробляється, то сценарій роботи Колектору на вимогу буде реалізований регулярним (щохвилинним) запуском програми з аргументом –new (Рис.8).

Код колектору буде розроблятись для системи, що складається з трьох типів девайсів: switches, routers, firewalls.

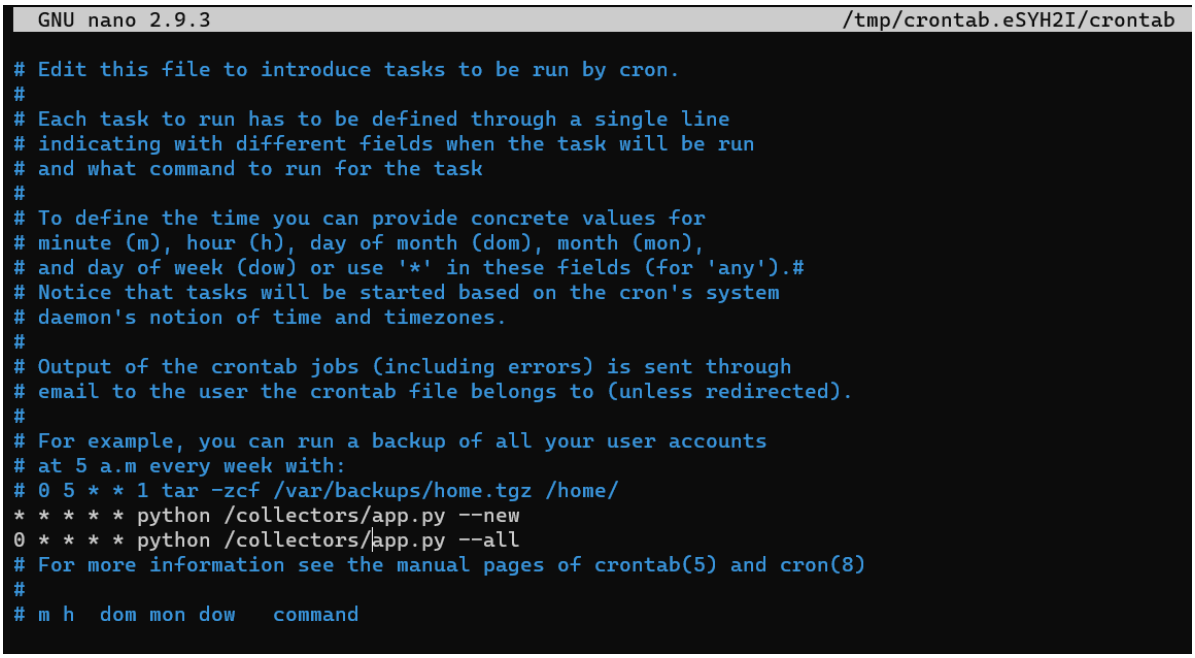


Рис.8 Приклад визначення двох cronjobs

Програма може бути запущеною з наступними аргументами:

all – конфігурації з усіх пристроїв, що є активними буде зібрано;

new – конфігурації з усіх пристроїв, що є активними та мають флаг is_to_gather_from встановленим в true, будуть зібрані;

switch – конфігурації з усіх активних пристроїв типу switch будуть зібрані;

router - конфігурації з усіх активних пристроїв типу router будуть зібрані;

firewall - конфігурації з усіх активних пристроїв типу firewall будуть зібрані.

Так як у даній роботі розробляється не ціла система, а лише один її компонент – Collector, було вирішено тестувати написану програму для реалізації функціоналу збору та збереження до сховища конфігурацій, використовуючи БД, розміщену не на хмарному сервері, а на локальному (так як розгортання БД на хмарних платформах вимагає додаткових витрат).

Для цього було використано систему керування базами даних MySQL та інструмент MySQL Workbench, що дозволяє взаємодіяти з MySQL базами даних через графічний інтерфейс.

Так як Колектори взаємодіють лише з однією сутністю з усієї змодельованої БД, то до створеної БД «dbtest» було додано тільки одну таблицю – «devices» (Рис.9), заповнену тестовими даними.

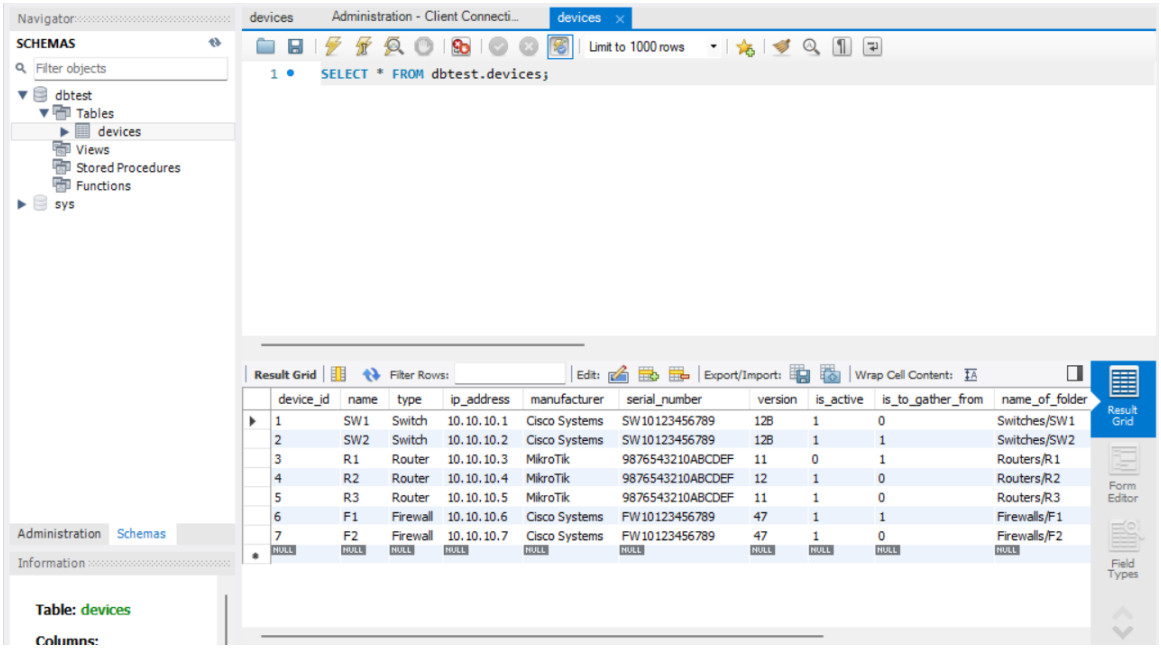


Рис.9 Створена БД у MySQL Workbench

Також у AWS було створено S3 bucket – «backupscollectors» (Рис.10)

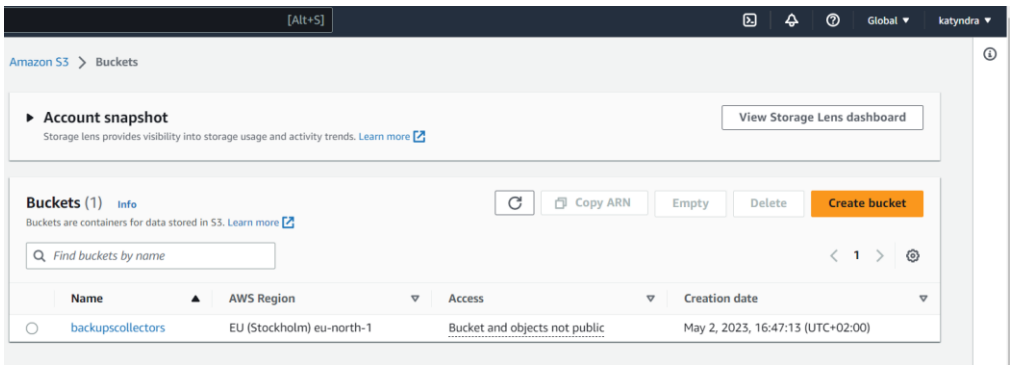


Рис.10 Створений S3 bucket

Для подальшого доступу до S3 bucket за допомогою бібліотеки boto3, було згенеровано ключ доступу з унікальним ID та Value (Рис.11).

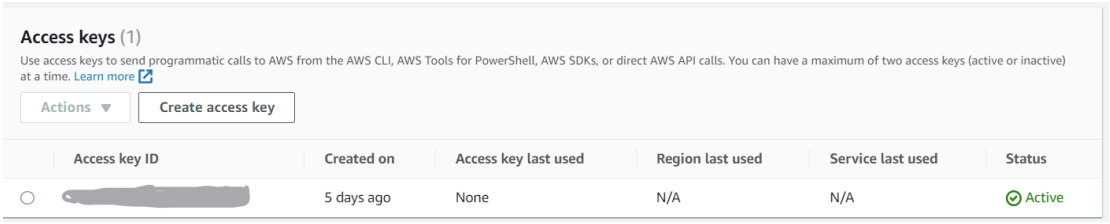


Рис.11 Створений Access Key

До директорії проекту, в якій розроблялась програма для компоненту Колектор, було додано config.ini (Рис. 12) файл, в якому зберігаються всі значення для отримання доступу до БД та s3 bucket.

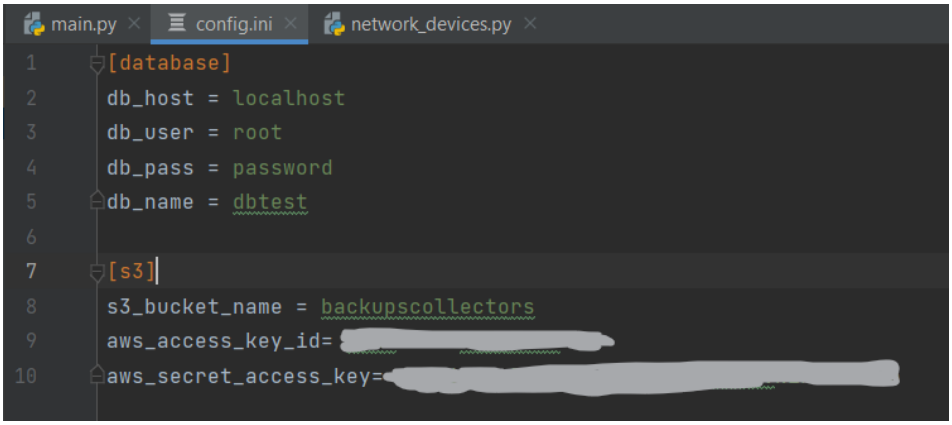


Рис.12 config.ini file

Було створено клас **Device** (Рис. 13), представники якого ініціюються наступними атрибутами:

- ім’я хоста,
- айпі-адреса девайсу,
- шлях до папки, в якій зберігаються всі конфігураційні файли даного представника класу девайсів.

Також є два атрибути – ssh та config - встановлене з девайсом ssh з'єднання та назва конфігураційного файлу, які при ініціалізації отримують значення None.

```
class Device:
    def __init__(self, hostname, ip_address, s3_object_prefix):
        self.hostname = hostname
        self.ip_address = ip_address
        self.s3_object_prefix = s3_object_prefix
        self.ssh = None
        self.config = None
```

Рис.13 __init__ метод класу Device

Device.connect() (Рис.14)

Встановлює SSH підключення до мережевого пристрою за указаною його айпі-адресою, іменем користувача та шляхом до ключа SSH, використовуючи модель paramiko [24].

Проте, так як код буде тестуватися без доступу до реальних пристроїв, до яких можна під`єднатись, на поточний момент логіка, що відповідає за з'єднання з пристроєм – закоментована. Натомість, метод виводить повідомлення про успішне під'єднання, у разі його виклику та відсутності помилок.

```
def connect(self) -> None:
    try:
        print(f"Connecting to {self.hostname} ({self.ip_address})...")
        # ssh = paramiko.SSHClient()
        # ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        # ssh.connect(self.ip_address, username=self.username, key_filename=self.ssh_key_path)
        # self.ssh = ssh
        print(f"Connected to {self.hostname} ({self.ip_address}) successfully!")
    except Exception as e:
        print(f"Error occurred while connecting to {self.hostname}: {e}")
```

Рис.14 connect метод класу Device

Device. collect_configuration() (Рис.15)

Асинхронний метод, призначений для збору конфігурації пристроїв. Повертає стрічку з конфігурацією та надає значення атрибуту config назви файлу, в якому зберігатиметься конфігурація у форматі «НазваХоста_Дата_Час».

Device. _collect_configuration() (Рис.15)

Метод є приватним та збирає конфігурацію, запускаючи та записуючи результати команди show running-config на пристроях.

Проте для тестування метод повертає стрічку з індикацією якому саме пристрою належить ця конфігурація.

```
async def collect_configuration(self) -> Optional[str]:
    try:
        loop = asyncio.get_running_loop()
        config = await loop.run_in_executor(None, self._collect_configuration)
        now = datetime.datetime.now()
        timestamp = now.strftime('%Y-%m-%d_%H-%M-%S')
        filename = f"{self.hostname}_{timestamp}.conf"
        self.config = filename
        print(f"The configuration for {self.hostname} was collected with a file name {self.config}")
        return config

    except Exception as e:
        print(f"Error occurred while collecting configuration for {self.hostname}: {e}")
        return None

def _collect_configuration(self) -> str:
    #stdin, stdout, stderr = self.ssh.exec_command('show running-config')
    config = f"I`m {self.hostname} and this is my configuration" #stdout.read().decode('utf-8')
    return config
```

Рис.15 collect_configuration _collect_configuration методи класу Device

Device. upload_configuration(s3_bucket_name, config) (Рис.16)

Асинхронний метод, що завантажує до s3 bucket файл з конфігурацією.

Використовує configparser, аби отримати значення ключів для доступу до AWS s3 bucket та бібліотеку boto3 [25] для з'єднання з сховищем та завантаженням сформованого файлу згідно визначеного шляху.

```
async def upload_configuration(self, s3_bucket_name, config) -> None:
    try:
        if not self.config:
            raise Exception("Configuration file is not found. Please collect configuration first.")

        c = configparser.ConfigParser()
        c.read('config.ini')
        aws_access_key_id = c['s3']['aws_access_key_id']
        aws_secret_access_key = c['s3']['aws_secret_access_key']

        s3 = boto3.resource('s3', aws_access_key_id=aws_access_key_id, aws_secret_access_key=aws_secret_access_key)
        s3_object_key = f"{self.s3_object_prefix}/{self.config}"
        s3.Bucket(s3_bucket_name).put_object(Key=s3_object_key, Body=config)
        print(f"Successfully uploaded configuration to S3 bucket {s3_bucket_name}")
        self.config = s3_object_key
    except Exception as e:
        print(f"Error occurred while uploading configuration for {self.hostname}: {e}")
```

Рис.16 upload_configuration метод класу Device

Два попередньо-описані методи було зроблено асинхронними, оскільки вони включають в себе операції вводу/виводу, які потенційно можуть блокувати та гальмувати цикл подій, що є критичним при роботі з мережами великого розміру. Зробивши ці методи асинхронними, з'являється можливість, чекаючи завершення одних операцій, виконувати паралельно інші, запобігаючи блокуванню програми та покращуючи ефективність.

Device.disconnect() (Рис.17)

Метод перевіряє, чи є встановленим SSH з’єднання з девайсом, і у разі його наявності, закриває його.

Так як при тестуванні коду з’єднання не буде встановлено – додано додатковий вивід, що виведе повідомлення при виклику методу.

```
def disconnect(self) -> None:
    try:
        print(f"{self.hostname} successfully disconnected")
        if self.ssh:
            self.ssh.close()
            self.ssh = None
            print(f"{self.hostname} successfully disconnected")
    except Exception as e:
        print(f"Error occurred while disconnecting from {self.hostname}: {e}")
```

Рис.17 disconnect метод класу Device

Router(Device), Switch(Device), Firewall(Device) (Рис.18)

Репрезентують три типи девайсів, які потенційно можуть бути наявними в мережі.

Три класи, які наслідують клас Device, наслідуючи всі його атрибути та методи та також маючи додатковий атрибут, що визначає їх тип.

```python
class Router(Device):
    def __init__(self, hostname, ip_address, username):
        super().__init__(hostname, ip_address, username)
        self.device_type = "Router"


class Switch(Device):
    def __init__(self, hostname, ip_address, username):
        super().__init__(hostname, ip_address, username)
        self.device_type = "Switch"


class Firewall(Device):
    def __init__(self, hostname, ip_address, username):
        super().__init__(hostname, ip_address, username)
        self.device_type = "Firewall"
```

Рис.18 Classes Router, Switch and Firewall

Функція connect_to_database() (Рис.19) - встановлює зєднання з БД.

Використовує configparser [26], для доступу до налаштувань, які необхідні для підключення до БД та бібліотеку pymysql [27] для встановлення цього підключення.

```python
def connect_to_database() -> Union[pymysql.connections.Connection, None]:
    try:
        config_db = configparser.ConfigParser()
        config_db.read('config.ini')
        db_host = config_db['database']['db_host']
        db_user = config_db['database']['db_user']
        db_pass = config_db['database']['db_pass']
        db_name = config_db['database']['db_name']

        connection = pymysql.connect(
            host=db_host,
            user=db_user,
            password=db_pass,
            database=db_name)
        return connection
    except Exception as e:
        print(f"Error occurred while connecting to database: {e}")
        return None
```

Рис.19 функція connect_to_database

Асинхронна функція process_device() (Рис.20)

Викликає всі методи, що необхідні для виконання функціоналу компоненту Колектор для конкретного представника класу Device, який їй було передано.

```
async def process_device(device, s3_bucket_name) -> None:
    device.connect()
    config = await device.collect_configuration()
    await device.upload_configuration(s3_bucket_name, config)
    device.disconnect()
```

Рис.20 функція process_device

Асинхронна функція main(devices, s3_bucket_name) (Рис.21)

Отримує на вхід список з об’єктами класу Device та ім’я s3 bucket

Створює список для зберігання завдань, аби зберігати process_device coroutine завдання для кожного об’єкту. Для кожного об’єкту створює завдання, додаючи їх до списку, а потім, використовуючи asyncio.gather(), одночасно запускає всі завдання та чекає їх завершення [28].

```
async def main(devices, s3_bucket_name) -> None:
    tasks = []
    for device in devices:
        tasks.append(asyncio.create_task(process_device(device, s3_bucket_name)))
    await asyncio.gather(*tasks)
```

Рис.21 функція main

Парс аргументів, з якими може бути запущена програма [29] (Рис.22).

```
parser = argparse.ArgumentParser(description='Device Configuration Collection Tool')

parser.add_argument('--all', action='store_true', help='Regular collection of all configurations')
parser.add_argument('--new', action='store_true', help='Collection for newly configured devices')
parser.add_argument('--switch', action='store_true', help='Collection from switches')
parser.add_argument('--router', action='store_true', help='Collection for routers')
parser.add_argument('--firewall', action='store_true', help='Collection for firewalls')
args = parser.parse_args()
```

Рис.22 parsing arguments

Створення з’єднання з БД та виконання різних запитів до неї (Рис.23), в залежності від того, з яким аргументом було запущено програму. У всіх випадках вибирається однакова інформація (ім’я, айпі-адреса, назва папки та тип девайсу), проте в залежності від аргументу змінюються вимоги до типу девайсів чи значення флагів.

```
conn = connect_to_database()
if conn is None:
    exit()

try:
    cur = conn.cursor()

    if args.switch:
        cur.execute("SELECT name, ip_address, name_of_folder, type FROM Devices WHERE type='switch' AND "
                    "is_active=True")
    elif args.router:
        cur.execute("SELECT name, ip_address, name_of_folder, type FROM Devices WHERE type='router' AND "
                    "is_active=True")
    elif args.firewall:
        cur.execute("SELECT name, ip_address, name_of_folder, type FROM Devices WHERE type='firewall' AND "
                    "is_active=True")
    elif args.all:
        cur.execute("SELECT name, ip_address, name_of_folder, type FROM Devices WHERE is_active=True")
    elif args.new:
        cur.execute("SELECT name, ip_address, name_of_folder, type FROM Devices WHERE is_active=True AND "
                    "is_to_gather_from=True")
    else:
        print('Invalid usage, please, use -h or --help to see available options.')
```

Рис.23 запити до БД

Послідовно обробляючи дані, отримані з БД, програма створюватиме об’єкти класу Router, Switch або Firewall, додаючи їх до списку девайсів (Рис.24).

```
        rows = cur.fetchall()
        # creating of objects and appending them to the list
        for row in rows:
            if row[3] == 'Router':
                devices.append(network_devices.Router(row[0], row[1], row[2]))
            elif row[3] == 'Switch':
                devices.append(network_devices.Switch(row[0], row[1], row[2]))
            elif row[3] == 'Firewall':
                devices.append(network_devices.Firewall(row[0], row[1], row[2]))

        cur.close()
        conn.close()
    except Exception as e:
        print(f"Error occurred while receiving the list of devices {e}")
        conn.close()
        exit()
```

Рис.24 формування списку девайсів

Після отримання списку девайсів – програма викликає функцію main, що асинхронно обробить кожен девайс (Рис.25).

```
try:
    asyncio.run(main(devices, s3_bucket_name))
except Exception as e:
    print(f"Error occurred while executing operations on the devices: {e}")
    exit()
```

Рис.25 виклик функції main

Нижче наведені приклади запуску програми з різними аргументами та структура, яка утворюється у S3 bucket після завантаження файлів.

```
C:\Users\ponom\PycharmProjects\pythonProject6>C:\Users\ponom\AppData\Local\Programs\Python\Python38\python.exe main.py --all
Connecting to SW1 (10.10.10.1)...
Connected to SW1 (10.10.10.1) successfully!
Connecting to SW2 (10.10.10.2)...
Connected to SW2 (10.10.10.2) successfully!
Connecting to R2 (10.10.10.4)...
Connected to R2 (10.10.10.4) successfully!
Connecting to R3 (10.10.10.5)...
Connected to R3 (10.10.10.5) successfully!
Connecting to F1 (10.10.10.6)...
Connected to F1 (10.10.10.6) successfully!
Connecting to F2 (10.10.10.7)...
Connected to F2 (10.10.10.7) successfully!
The configuration for SW1 was collected with a file name SW1_2023-05-07_17-56-02.conf
Successfully uploaded configuration to S3 bucket backupscollectors
SW1 successfully disconnected
The configuration for SW2 was collected with a file name SW2_2023-05-07_17-56-03.conf
Successfully uploaded configuration to S3 bucket backupscollectors
SW2 successfully disconnected
The configuration for R2 was collected with a file name R2_2023-05-07_17-56-03.conf
Successfully uploaded configuration to S3 bucket backupscollectors
R2 successfully disconnected
The configuration for R3 was collected with a file name R3_2023-05-07_17-56-04.conf
Successfully uploaded configuration to S3 bucket backupscollectors
R3 successfully disconnected
The configuration for F1 was collected with a file name F1_2023-05-07_17-56-04.conf
Successfully uploaded configuration to S3 bucket backupscollectors
F1 successfully disconnected
The configuration for F2 was collected with a file name F2_2023-05-07_17-56-05.conf
Successfully uploaded configuration to S3 bucket backupscollectors
F2 successfully disconnected
```

Рис.26 результат запуску з аргументом --all

```
C:\Users\ponom\PycharmProjects\pythonProject6>C:\Users\ponom\AppData\Local\Programs\Python\Python38\python.exe main.py --new
Connecting to SW2 (10.10.10.2)...
Connected to SW2 (10.10.10.2) successfully!
Connecting to F1 (10.10.10.6)...
Connected to F1 (10.10.10.6) successfully!
The configuration for SW2 was collected with a file name SW2_2023-05-07_20-46-30.conf
Successfully uploaded configuration to S3 bucket backupscollectors
SW2 successfully disconnected
The configuration for F1 was collected with a file name F1_2023-05-07_20-46-31.conf
Successfully uploaded configuration to S3 bucket backupscollectors
F1 successfully disconnected
```

Рис.27 результат запуску з аргументом –new

```
C:\Users\ponom\PycharmProjects\pythonProject6>C:\Users\ponom\AppData\Local\Programs\Python\Python38\python.exe main.py --router
Connecting to R2 (10.10.10.4)...
Connected to R2 (10.10.10.4) successfully!
Connecting to R3 (10.10.10.5)...
Connected to R3 (10.10.10.5) successfully!
The configuration for R2 was collected with a file name R2_2023-05-07_20-47-01.conf
Successfully uploaded configuration to S3 bucket backupscollectors
R2 successfully disconnected
The configuration for R3 was collected with a file name R3_2023-05-07_20-47-03.conf
Successfully uploaded configuration to S3 bucket backupscollectors
R3 successfully disconnected
```

Рис.28 результат запуску з аргументом --router

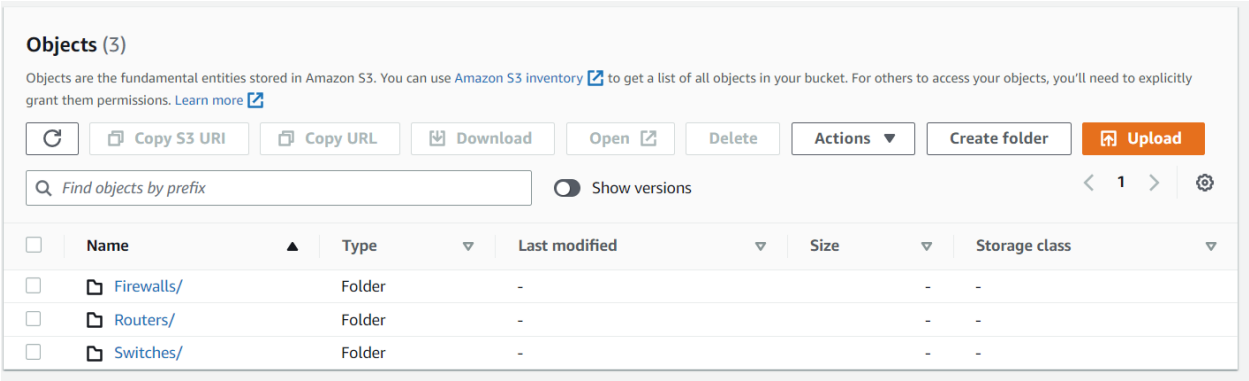


Рис.29 папки в s3 bucket

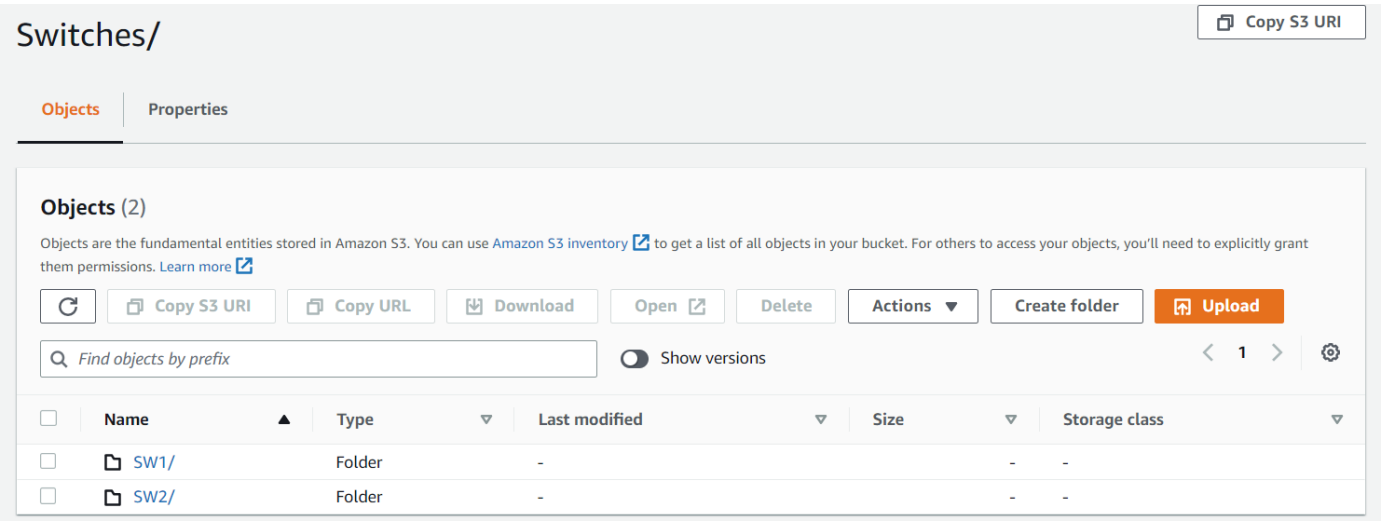


Рис.30 папки для switches

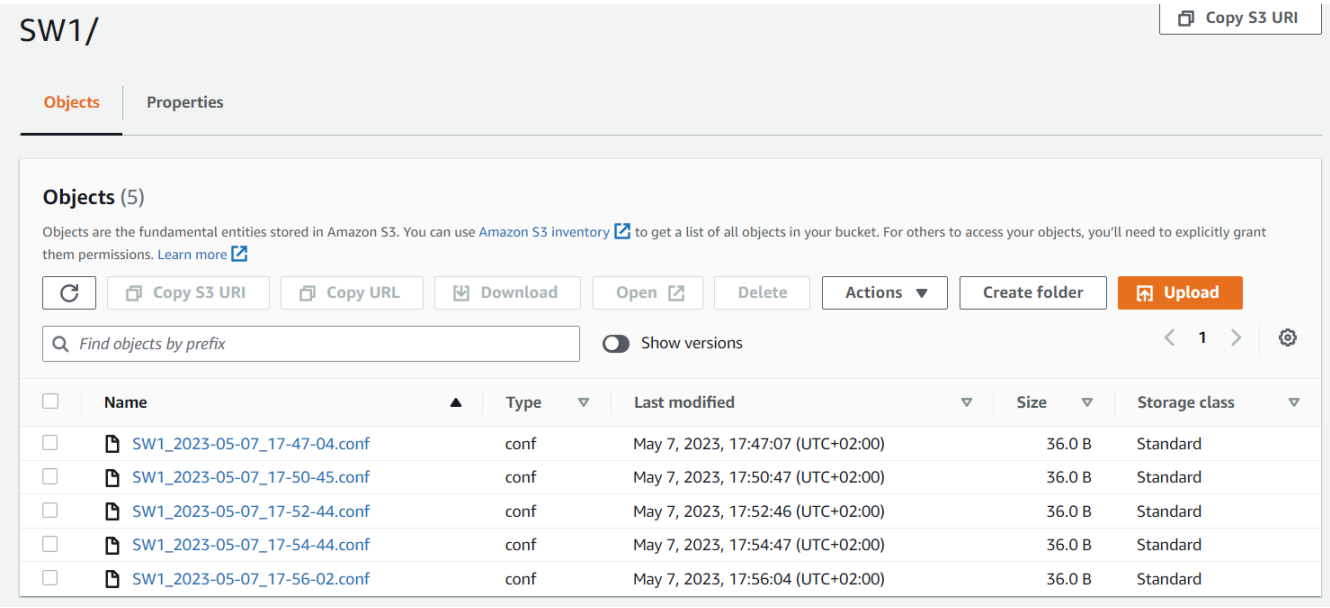


Рис.31 backups для SW1

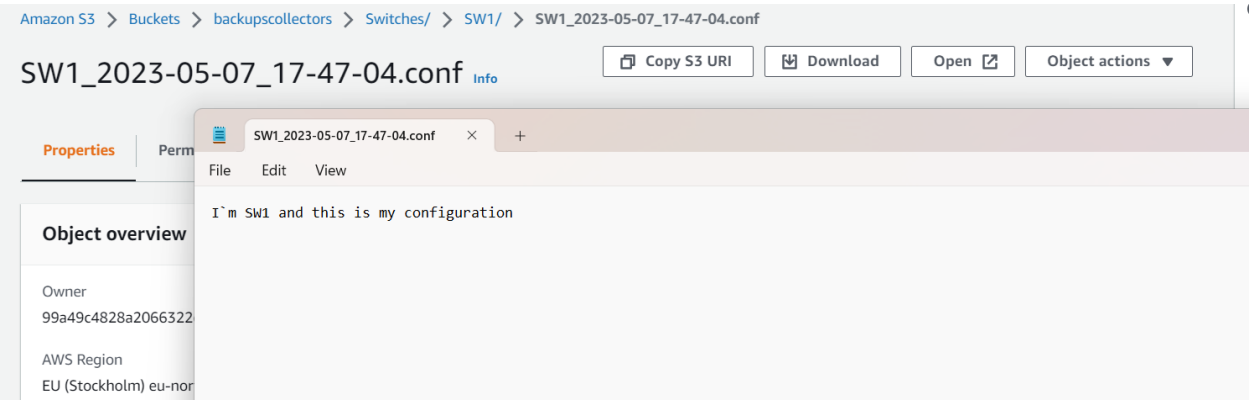


Рис.32 конкретний backup для SW1

Висновки до розділу 3

У розділі 3 описаний процес розробки компоненту «Колектор» та продемонстровано його роботу. Також було розроблено код, який реалізує підключення до потенційних девайсів.

У ході розробки коду були використані підходи ООП (об’єктно-орієнтованого програмування), виконано роботу з БД (підключення, виконання запитів, обробку даних), робота з підключенням SSH до пристроїв, робота з AWS SDK boto3 для підключення та роботи з сервісами AWS, бібліотека asyncio для асинхронного виконання завдань, робота з аргументами командного рядка. Код був протестований на коректність роботи та відсутність помилок.

Висновки

Під час написання даної роботи було виконано наступні завдання:

- визначено основні завдання СККККМ та обґрунтовано важливість використання даних систем для підвищення надійності роботи мережі;
- проведено порівняльний аналіз різних архітектурних шаблонів для використання у розробці СККККМ;
- розглянуто різні платформи для побудови СККККМ – їх переваги та недоліки у контексті задач системи;
- розглянуто існуючі продукти для реалізації компонентів СККККМ системи;
- проведено структурну розробку власної СККККМ системи та описано функціонування та особливості кожного змодельованого компоненту системи;
- розроблено та протестовано один із компонентів системи – Колектор.

Досягнуто таких результатів:

- розроблено структурну схему СККККМ, яка задовольняє основним стандартним вимогам до неї: надійність контролю, ефективний моніторинг стану пристроїв та їх конфігурацій, структуроване зберігання даних;
- проведено практичну розробку одного з компонентів структурної схеми СККККМ.

Розроблені структурна схема СККККМ та компонент можуть бути застосовані на підприємствах для централізованого управління конфігураціями використовуваних девайсів та оптимізації роботи системних інженерів.

Список використаних джерел

1. Key Importance of Network Configuration Management [Електронний ресурс] – Режим доступу: https://infraon.io/blog/key-importance-of-network-configuration-management/
2. Network configuration management [Електронний ресурс] – Режим доступу: https://www.cisco.com/en/US/technologies/tk869/tk769/technologies_white_paper0900aecd806c0d88.html
3. Microservice architecture at Medium [Електронний ресурс] – Режим доступу: https://medium.engineering/microservice-architecture-at-medium-9c33805eb74f
4. On Modular Architectures [Електронний ресурс] – Режим доступу: https://medium.com/on-software-architecture/on-modular-architectures-53ec61f88ff4
5. Microservice architecture [Електронний ресурс] – Режим доступу: https://medium.com/@IvanZmerzlyi/microservice-architecture-f8a382291ff4
6. Docker vs. Virtual Machines: Differences You Should Know [Електронний ресурс] – Режим доступу: https://cloudacademy.com/blog/docker-vs-virtual-machines-differences-you-should-know/#:~:text=VMs%20have%20the%20host%20OS,and%20increases%20the%20boot%20time.
7. Use containers to Build, Share and Run your applications [Електронний ресурс] – Режим доступу: https://www.docker.com/resources/what-container/
8. Top 20 Docker Security Best Practices: Ultimate Guide [Електронний ресурс] – Режим доступу: https://blog.aquasec.com/docker-security-best-practices
9. When to Use Docker Compose vs. Kubernetes [Електронний ресурс] – Режим доступу: https://earthly.dev/blog/dockercompose-vs-k8s/#:~:text=One%20key%20difference%20is%20that,several%20machines%2C%20virtual%20or%20real

10. Kubernetes vs Docker Compose: What's the difference? [Електронний ресурс] – Режим доступу: https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/What-is-Kubernetes-vs-Docker-Compose-How-these-DevOps-tools-compare
11. Amazon Simple Storage Service [Електронний ресурс] – Режим доступу: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
12. Documentation MinIO [Електронний ресурс] – Режим доступу: https://min.io/docs/minio/kubernetes/upstream/
13. Ansible Documentation [Електронний ресурс] – Режим доступу: https://docs.ansible.com/ansible/latest/index.html
14. Ansible vs. Puppet: What you need to know [Електронний ресурс] – Режим доступу: https://www.redhat.com/en/topics/automation/ansible-vs-puppet
15. Overview of Puppet`s architecture [Електронний ресурс] – Режим доступу: https://www.puppet.com/docs/puppet/5.5/architecture.html
16. RANCID - Really Awesome New Cisco confIg Differ [Електронний ресурс] – Режим доступу: https://shrubbery.net/rancid/
17. Oxidized [Електронний ресурс] – Режим доступу: https://github.com/ytti/oxidized
18. NGINX vs Apache: Head to Head Comparison [Електронний ресурс] – Режим доступу: https://hackr.io/blog/nginx-vs-apache
19. Role-Based Access Control (RBAC) [Електронний ресурс] – Режим доступу: https://www.imperva.com/learn/data-security/role-based-access-control-rbac/
20. SSH vs SFTP Guide [Електронний ресурс] – Режим доступу: https://www.comparitech.com/net-admin/ssh-vs-sftp/
21. Crow`s Foot Notation [Електронний ресурс] – Режим доступу: https://vertabelo.com/blog/crow-s-foot-notation/
22. Cloud Computing Advantages in the Public Sector [Електронний ресурс] – Режим доступу: https://www.cisco.com/c/dam/en_us/solutions/industries/docs/c11-687784_cloud_omputing_wp.pdf

23. Crontabguru [Електронний ресурс] – Режим доступу: https://crontab.guru/
24. Paramiko documentation [Електронний ресурс] – Режим доступу: https://docs.paramiko.org/en/stable/
25. Boto3 documentation [Електронний ресурс] – Режим доступу: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/s3-example-creating-buckets.html
26. Configparser documentation [Електронний ресурс] – Режим доступу: https://docs.python.org/3/library/configparser.html
27. pyMySql documentation [Електронний ресурс] – Режим доступу: https://pymysql.readthedocs.io/en/latest/
28. Asyncio documentation [Електронний ресурс] – Режим доступу: https://docs.python.org/3/library/asyncio.html
29. Argparse socumentation [Електронний ресурс] – Режим доступу: https://docs.python.org/3/library/argparse.html

Додаток А

```
import asyncio

import configparser

from typing import Optional


import paramiko

import datetime

import boto3


"""

Class Device

Represents a device to which the connection should occur and the configuration should be collected and uploaded to s3


Attributes:

hostname

ip_address

s3_object_prefix: the path to the folder, where all configurations are stored

ssh: An SSHClient object representing the SSH connection to the device

config: The name of the S3 object where the configuration for this device is stored
```

Methods:

def __init__(self, hostname, ip_address, s3_object_prefix) -> None

def connect(self) -> None:

async def collect_configuration(self) -> Optional[str]:

def _collect_configuration(self) -> str:

async def upload_configuration(self, s3_bucket_name, config):

def disconnect(self):

Note: The SSH connection and configuration name are not initialized in the constructor, but receive a value

in other methods of the class.

"""

class Device:

"""

Initializes a new instance of the Device class with the given hostname, IP address, and S3 object prefix

"""

def __init__(self, hostname, ip_address, s3_object_prefix) -> None:

```
        self.hostname = hostname
        self.ip_address = ip_address
        self.s3_object_prefix = s3_object_prefix
        self.ssh = None
        self.config = None

    """
    Description: is used to establish an SSH connection to a network device with the specified IP address,
    username, and SSH key path, using the paramiko module to create an SSH connection.
    Parameters: None
    Returns: None
    """

    def connect(self) -> None:
        try:
            print(f"Connecting to {self.hostname} ({self.ip_address})...")
            # ssh = paramiko.SSHClient()
            # ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
            # ssh.connect(self.ip_address, username=self.username, key_filename=self.ssh_key_path)
```

```
            # self.ssh = ssh
            print(f"Connected to {self.hostname} ({self.ip_address}) successfully!")
        except Exception as e:
            print(f"Error occurred while connecting to {self.hostname}: {e}")


    """
    Description: Collects the configuration of the device asynchronously and saves it to a file with a timestamp and the
                                                                                    device name.
    Parameters: None
    Returns: The configuration as a string or None in case of an error.
    Raises: None
    """

    async def collect_configuration(self) -> Optional[str]:
        try:
            loop = asyncio.get_running_loop()
            config = await loop.run_in_executor(None, self._collect_configuration)
            now = datetime.datetime.now()
            timestamp = now.strftime('%Y-%m-%d_%H-%M-%S')
            filename = f"{self.hostname}_{timestamp}.conf"
```

```
            self.config = filename
            print(f"The configuration for {self.hostname} was collected with a file name {self.config}")
            return config

        except Exception as e:
            print(f"Error occurred while collecting configuration for {self.hostname}: {e}")
            return None

    """
    Description: Collects the configuration of the device using SSH and returns it as a string.
    Parameters: None
    Returns: The configuration as a string.
    Raises: None
    """

    def _collect_configuration(self) -> str:
        # stdin, stdout, stderr = self.ssh.exec_command('show running-config')
        config = f"I`m {self.hostname} and this is my configuration"  # stdout.read().decode('utf-8')
        return config
```

"""

Uploads the configuration file of a device to an S3 bucket using the boto3 library. Takes two parameters:

s3_bucket_name: A string, representing the name of the S3 bucket to upload the configuration file

config: A string representing the configuration file contents

Returns: None

Exceptions:

If the configuration file is not found, it raises an Exception with specified message

If there is any other exception while uploading the configuration file, prints an error message with the

hostname and the exception

"""

```
async def upload_configuration(self, s3_bucket_name, config) -> None:
    try:
        if not self.config:
            raise Exception("Configuration file is not found. Please collect configuration first.")
```

```
            c = configparser.ConfigParser()
            c.read('config.ini')
            aws_access_key_id = c['s3']['aws_access_key_id']
            aws_secret_access_key = c['s3']['aws_secret_access_key']

            s3 = boto3.resource('s3', aws_access_key_id=aws_access_key_id,
aws_secret_access_key=aws_secret_access_key)
            s3_object_key = f"{self.s3_object_prefix}/{self.config}"
            s3.Bucket(s3_bucket_name).put_object(Key=s3_object_key, Body=config)
            print(f"Successfully uploaded configuration to S3 bucket {s3_bucket_name}")
            self.config = s3_object_key
        except Exception as e:
            print(f"Error occurred while uploading configuration for {self.hostname}: {e}")

    """
    Attempts to close the SSH connection to the device
    Parameters:None
    Returns:None
    Raises: None
    """
```

```
    def disconnect(self) -> None:
        try:
            print(f"{self.hostname} successfully disconnected")
            if self.ssh:
                self.ssh.close()
                self.ssh = None
                print(f"{self.hostname} successfully disconnected")
        except Exception as e:
            print(f"Error occurred while disconnecting from {self.hostname}: {e}")



"""
Classes Router(Device), Switch(Device) and Firewall(Device) inherit Device class and all its methods and attributes
Also have additional attribute device_type
"""



class Router(Device):
    def __init__(self, hostname, ip_address, username):
```

```
        super().__init__(hostname, ip_address, username)
        self.device_type = "Router"


class Switch(Device):
    def __init__(self, hostname, ip_address, username):
        super().__init__(hostname, ip_address, username)
        self.device_type = "Switch"


class Firewall(Device):
    def __init__(self, hostname, ip_address, username):
        super().__init__(hostname, ip_address, username)
        self.device_type = "Firewall"
```

Додаток В

```
from typing import Union

import network_devices
import argparse
import pymysql
import configparser
import asyncio

"""
Description: Establishes a connection to a MySQL database using the configuration details specified in a config.ini file
Uses the pymysql library to establish a connection to the MySQL database with the specified configuration details
If any error occurs while establishing the connection, it prints an error message and returns None

Parameters: None
Return Value: Returns a connection object that can be used to interact with the MySQL database or None
"""


def connect_to_database() -> Union[pymysql.connections.Connection, None]:
    try:
        config_db = configparser.ConfigParser()
        config_db.read('config.ini')
```

```
        db_host = config_db['database']['db_host']
        db_user = config_db['database']['db_user']
        db_pass = config_db['database']['db_pass']
        db_name = config_db['database']['db_name']

        connection = pymysql.connect(
            host=db_host,
            user=db_user,
            password=db_pass,
            database=db_name)
        return connection
    except Exception as e:
        print(f"Error occurred while connecting to database: {e}")
        return None
```

"""

Asynchronously processes each device: connects to it, collects configuration, uploads it to s3 and disconnects

Arguments:

device: an instance of Device class representing the network device to be processed

s3_bucket_name: a string representing the name of the S3 bucket where the configuration will be uploaded

Returns: None

Raises: None

"""

```
async def process_device(device, s3_bucket_name) -> None:
    device.connect()
    config = await device.collect_configuration()
    await device.upload_configuration(s3_bucket_name, config)
    device.disconnect()
```

"""

Creates an empty list of tasks to store the process_device coroutine tasks for each Device object.

Loops through each Device object in the devices list and creates a task for the process_device coroutine for each device

(using asyncio.create_task()). Appends these tasks to the tasks list.

Finally, uses asyncio.gather() to run all the tasks concurrently and waits for them to complete

Arguments

-list of Device objects

-an S3 bucket name

Returns: None

Raises: None

"""

```
async def main(devices, s3_bucket_name) -> None:
```

```
    tasks = []
    for device in devices:
        tasks.append(asyncio.create_task(process_device(device, s3_bucket_name)))
    await asyncio.gather(*tasks)


if __name__ == '__main__':
    """
    Parsing all arguments
    Usage
    python main.py --all (cronjob runs each hour)
    python main.py --new (cronjob runs each minute)
    python main.py --switch (gathers configuration for all active switches)
    python main.py --router (gathers configuration for all active routers)
    python main.py --firewall (gathers configuration for all active firewalls)

    """
    parser = argparse.ArgumentParser(description='Device Configuration Collection Tool')

    parser.add_argument('--all', action='store_true', help='Regular collection of all configurations')
    parser.add_argument('--new', action='store_true', help='Collection for newly configured devices')
    parser.add_argument('--switch', action='store_true', help='Collection from switches')
    parser.add_argument('--router', action='store_true', help='Collection for routers')
    parser.add_argument('--firewall', action='store_true', help='Collection for firewalls')
```

```
    args = parser.parse_args()

    # list with devices and parser to get name of the bucket from the config file
    devices = []
    config = configparser.ConfigParser()
    config.read('config.ini')
    s3_bucket_name = config['s3']['s3_bucket_name']

    """
    Creation of the connection to the DB and execution of queries in correlation to the specified arguments
    """
    conn = connect_to_database()
    if conn is None:
        exit()

    try:
        cur = conn.cursor()

        if args.switch:
            cur.execute("SELECT name, ip_address, name_of_folder, type FROM Devices WHERE type='switch' AND "
                        "is_active=True")
        elif args.router:
            cur.execute("SELECT name, ip_address, name_of_folder, type FROM Devices WHERE type='router' AND "
                        "is_active=True")
        elif args.firewall:
```

```
            cur.execute("SELECT name, ip_address, name_of_folder, type FROM
Devices WHERE type='firewall' AND "
                        "is_active=True")
        elif args.all:
            cur.execute("SELECT name, ip_address, name_of_folder, type FROM
Devices WHERE is_active=True")
        elif args.new:
            cur.execute("SELECT name, ip_address, name_of_folder, type FROM
Devices WHERE is_active=True AND "
                        "is_to_gather_from=True")
        else:
            print('Invalid usage, please, use -h or --help to see available options.')

        rows = cur.fetchall()
        # creating of objects and appending them to the list
        for row in rows:
            if row[3] == 'Router':
                devices.append(network_devices.Router(row[0], row[1], row[2]))
            elif row[3] == 'Switch':
                devices.append(network_devices.Switch(row[0], row[1], row[2]))
            elif row[3] == 'Firewall':
                devices.append(network_devices.Firewall(row[0], row[1], row[2]))
        # close the connection
        cur.close()
        conn.close()
    except Exception as e:
        print(f"Error occurred while receiving the list of devices {e}")
        conn.close()
```

```
        exit()
    # takes a coroutine as an argument, runs the coroutine, and then shuts down the event loop once the coroutine
    # completes
    try:
        asyncio.run(main(devices, s3_bucket_name))
    except Exception as e:
        print(f"Error occurred while executing operations on the devices: {e}")
        exit()
```