Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська дипломна робота

на тему: «Розробка програмного забезпечення для управління конфігураціями при розгорненні та масштабуванні електронних ресурсів»

Виконав: студент IV курсу
групи 3ПІ-18б
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Миргородський А.В.
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ Романюк О.В.
(прізвище та ініціали)
Рецензент: к.т.н., доц. каф. КН Колодний В.В.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедри _______ О.Н. Романюк
«13» червня 2022 р.

ВНТУ – 2022

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.

____

25 березня 2022 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Миргородському Андрію Вікторовичу

1. Тема роботи – «Розробка програмного забезпечення для управління конфігураціями при розгорненні та масштабуванні електронних ресурсів».

Керівник роботи: Романюк Оксана Володимирівна, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від 24 березня 2022 року №66.

2. Строк подання студентом роботи 13 червня 2022 року.

3. Вихідні дані до роботи: модель розробки – ітеративна; метод передачі повідомлень між серверами – виклик віддалених процедур (RPC); вхідні дані – текстові файли в форматі TOML з описом бажаного стану системи та налаштуваннями для підключення і роботи з серверами; вихідні дані – логи виконання інструкцій на віддалених серверах; середовище розробки – JetBrains GoLand; мова програмування – Go.

4. Зміст розрахунково–пояснювальної записки: вступ; обґрунтування вибору методу розробки та постановка задачі дослідження; розробка структури та алгоритмів роботи програмного продукту; розробка програмних компонент для додатку; тестування програми; висновки; список використаних джерел; додатки.

5. Перелік графічного матеріалу: титульний слайд; актуальність теми; мета, об'єкт та предмет дослідження; задачі дослідження; новизна і практична цінність одержаних результатів; порівняльний аналіз аналогів; архітектура додатку; блок-схема алгоритму формування змінних хостів; блок-схема алгоритму запуску конфігураційних скриптів; структура графічного інтерфейсу

головного вікна додатку; структура графічного інтерфейсу вкладки «Інвентар»; структура графічного інтерфейсу вкладки «Стан»; структура графічного інтерфейсу вкладки «Виконання»; тестування програми; апробація матеріалів бакалаврської дипломної роботи; фінальний слайд.

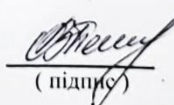
6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Романюк О.В., к.т.н., доцент кафедри ПЗ	25.03.22	10.06.22

7. Дата видачі завдання 25 березня 2022 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання і вибір методу вирішення поставленої задачі дослідження	26.03.2022 – 02.04.2022	Вик.
2	Розробка архітектури програмного додатку	03.04.2022 – 09.04.2022	Вик.
3	Розробка алгоритму формування змінних хостів	10.04.2022 – 17.04.2022	Вик.
4	Розробка алгоритму запуску конфігураційних скриптів	18.04.2022 – 28.04.2022	Вик.
5	Аналіз і вибір мови програмування та середовища розробки	29.04.2022 – 05.05.2022	Вик.
6	Програмна реалізація додатку	06.05.2022 – 20.05.2022	Вик.
7	Тестування програмного забезпечення	21.05.2022 – 25.05.2022	Вик.
8	Оформлення матеріалів до захисту БДР	26.05.2022 – 10.06.2022	Вик.

Студент  **Миргородський А.В.**
(підпис) (прізвище та ініціали)

Керівник бакалаврської дипломної роботи ________ **Романюк О. В.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота складається з 115 сторінок формату А4, на яких є 45 рисунків, 4 таблиці, список використаних джерел містить 34 найменування.

У бакалаврській дипломній роботі проведено детальний аналіз стану програмного забезпечення для управління конфігураціями при розгорненні та масштабуванні електронних ресурсів. Встановлено об’єкт, предмет, завдання та методи дослідження. Сформульовано мету дослідження – підвищення ефективності та гнучкості процесу управління конфігураціями при розгорненні та масштабуванні електронних ресурсів. Для реалізації мети було розроблено алгоритми роботи, інтерфейс та програмну реалізацію додатку для управління конфігураціями.

Запропоновано алгоритм формування змінних серверів, який враховує архітектурні особливості продукту та підходить для створення комплексних конфігурацій електронних ресурсів. Розроблено алгоритм запуску конфігураційних скриптів, який дозволяє гнучко налаштовувати процес виконання та динамічно розширювати його контекст.

Програмний додаток розроблено з використанням мови програмування Go, бібліотеки Yaegi, фреймворку gRPC та середовища розробки JetBrains GoLand. В результаті виконання бакалаврської дипломної роботи розроблено програмний засіб, працездатність і правильність роботи якого перевірено, підготовлена інструкція користувача.

Отримані в бакалаврській дипломній роботі результати можна використати для побудови високоефективної системи автоматизованого розгорнення і масштабування електронних ресурсів.

Ключові слова: керування конфігураціями, розгорнення електронних ресурсів, масштабування електронних ресурсів.

ABSTRACT

The bachelor’s thesis consists of 115 pages of A4 format, which has 45 figures, 4 tables, the list of references contains 34 titles.

A detailed analysis of the state of the software for configuration management in the deployment and scaling of electronic resources is carried out in the bachelor's thesis. The object, subject, tasks and methods of research were established. The purpose of the study is to increase the efficiency and flexibility of the configuration management process when deploying and scaling electronic resources. To implement the goal were developed algorithms, interface and software implementation of the configuration management application.

Algorithm for the formation of servers’ variables, considering the architectural features of the product and suitable for the creation of complex configurations of electronic resources, was proposed. Algorithm of running configuration scripts, which allows to flexibly configure the process of execution and dynamically expand its context, is developed.

The software application was developed using Go programming language, Yaegi library, gRPC framework and JetBrains GoLand development environment. As a result of the bachelor's thesis a software tool was developed, the efficiency and correctness of which was tested, and the user's manual was prepared.

The results obtained in the bachelor's thesis can be used to build a highly efficient system of automated deployment and scaling of electronic resources.

Key words: configuration management, deployment of electronic resources, scaling of electronic resources.

ЗМІСТ

ВСТУП........ 8
1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ........ 12
1.1 Аналіз технологій для керування конфігураціями........ 12
1.2 Порівняльний аналіз аналогів........ 14
1.3 Аналіз методів розв'язання поставленої задачі........ 19
1.4 Постановка задач розробки програмного забезпечення для управління конфігураціями........ 22
1.5 Висновки........ 23
2 РОЗРОБКА АРХІТЕКТУРИ ТА АЛГОРИТМІВ РОБОТИ ПРОГРАМНОГО ПРОДУКТУ........ 24
2.1 Розробка архітектури програмного продукту........ 24
2.2 Розробка алгоритму формування змінних хостів........ 27
2.3 Розробка алгоритму запуску конфігураційних скриптів........ 30
2.4 Розробка структури графічного інтерфейсу користувача........ 34
2.5 Висновки........ 38
3 РОЗРОБКА ПРОГРАМНИХ КОМПОНЕНТ ДОДАТКУ........ 39
3.1 Варіантний аналіз і обґрунтування вибору мови програмування........ 39
3.2 Вибір середовища розробки........ 43
3.3 Програмна реалізація додатку........ 47
3.4 Висновки........ 59
4 ТЕСТУВАННЯ ПРОГРАМИ........ 60
4.1 Аналіз методів тестування програмного забезпечення........ 60
4.2 Тестування розробленого програмного продукту........ 62
4.3 Розробка інструкції користувача........ 66
4.4 Системні вимоги........ 70
4.5 Висновки........ 71
ВИСНОВКИ........ 72

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 73
ДОДАТКИ 76
Додаток А. Технічне завдання 77
Додаток Б. Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень 81
Додаток В. Лістинг програми 83
Додаток Г. Графічна частина 107

ВСТУП

Обґрунтування вибору теми дослідження. Розрахункові потужності, що необхідні для сучасного програмного забезпечення, стають дедалі більшими [1]. Особливо це стосується серверного програмного забезпечення, яке може складатися з великої кількості високонавантажених компонентів: баз даних, що дублюються на декількох машинах заради високої доступності та гарантій безпроблемної роботи, різноманітних серверів для обслуговування внутрішнього API і мікросервісів, застосунків для брокерів повідомлень або управління обліковими даними, а також багато інших.

Керування такою кількістю ПЗ на серверах з різними конфігураціями може швидко стати складною задачею, особливо в випадку роботи в команді й швидкого розвитку проекту. Крім того, зазвичай, менеджмент інфраструктури виходить далеко за межі лише одного основного середовища (production environment), адже необхідно також керувати внутрішніми системами для розробників (development environment) і тестувальників (integration/demo environment). Добре налаштовані CI/CD-процеси також вимагають додаткової інфраструктури для автоматизованої збірки, тестування й розповсюдження компонентів продукту [2].

Якщо втрата однієї або декількох машин з CI/CD-систем або середовища для розробників не є критичною ситуацією, яка буде потребувати лише додаткового часу на відновлення процесів, то інша частина інфраструктури не може собі дозволити довгі періоди непрацездатності. Це може вилитися в фінансові та репутаційні втрати [3], які будуть зростати з часом поки поточний інженер на чергуванні буде вручну відновлювати втрачений сервер із резервних копій або з нуля створювати необхідні програмні ресурси.

Для вирішення задач даного характеру існує окремий прошарок програмного забезпечення, який спрощує й автоматизує менеджмент інфраструктури. До цієї категорії можна віднести різноманітні інструменти для IaC (Infrastructure as Code) або забезпечення для керування конфігураціями.

Додатки для управління конфігураціями дозволяють швидко досягти бажаного стану системи: наприклад, встановити необхідне ПЗ і згенерувати коректні файли з налаштуваннями. Крім того, такий процес, зазвичай, просто автоматизується шляхом інтеграцій з існуючими CI/CD-підходами. Разом з іншими популярними DevOps-інструментами вони можуть стати надійним ядром підтримки електронних ресурсів та їх інфраструктури [4].

Наразі існує велика кількість додатків для управління конфігураціями, але їх функціональні та архітектурні недоліки можуть стати значними обмеженнями при певних сценаріях використання. Наприклад, відсутність GUI в базовій редакції ПЗ може стати форс-мажором для недосвідчених у роботі з CLI користувачів, обмежена підтримка ОС Windows може ускладнити розробку конфігурацій для певних пристроїв. Використання додаткових компонентів, як-от бібліотек для Python в продукті Ansible, збільшує час первинної підготовки систем до роботи з ПЗ для управління конфігураціями, а опис інструкцій у структурованому вигляді в файлах формату TOML або JSON може стати значним обмеженням за потреби комплексного налаштування систем [5].

Тому задача підвищення ефективності управління конфігураціями є актуальною, бо існуючі реалізації мають певні недоліки. Це передбачає розробку нових методів і алгоритмів роботи.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою дослідження є підвищення ефективності та гнучкості процесу управління конфігураціями при розгорненні та масштабуванні електронних ресурсів шляхом удосконалення алгоритмів формування змінних хостів і запуску конфігураційних скриптів.

Відповідно до поставленої мети в бакалаврській дипломній роботі потрібно вирішити такі **завдання**:

– проаналізувати стан технологій для управління конфігураціями;

– визначити високорівневу структуру та архітектуру програмного продукту;

– розробити алгоритм формування змінних хостів для їх використання при застосуванні конфігурацій;

– розробити алгоритм запуску конфігураційний скриптів;

– розробити графічний інтерфейс користувача для створюваного продукту;

– розробити програмний додаток для керування конфігураціями на основі створеної архітектури та алгоритмів;

– розробити модульні тести для автоматизованого тестування програмного додатку;

– провести ручне тестування програмного продукту з використанням комплексних сценаріїв виконання;

– розробити інструкцію користувача.

Об’єкт дослідження – процеси управління конфігураціями при розгорненні та масштабуванні електронних ресурсів.

Предмет дослідження – методи та засоби розробки програмного забезпечення для управління конфігураціями.

Методи дослідження. У процесі дослідження використовувались: комплексний аналіз з метою визначення недоліків існуючих технічних рішень задачі та подальший синтез отриманих даних для формування нових функціональних характеристик і вимог; теорія алгоритмів для розробки та вдосконалення алгоритмів ПЗ; комп’ютерне моделювання для аналізу та перевірки отриманих теоретичних положень.

Новизна отриманих результатів.

1. Удосконалено алгоритм формування змінних хостів, у якому, на відміну від відомих алгоритмів, враховано комплексну логіку пріоритизації та перезапису даних, скореговано поведінку відповідно до зроблених архітектурних рішень та контексту виконання, що дозволило створювати та виконувати більш комплексні конфігурації систем.

2. Удосконалено алгоритм запуску конфігураційних скриптів, який, на відміну від аналогічних алгоритмів, дозволяє використовувати інтерпретовані скрипти й модулі з можливістю динамічно змінювати контекст виконання, що дозволило підвищити гнучкість процесу керування конфігураціями.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих в бакалаврській дипломній роботі теоретичних положень запропоновано алгоритми та розроблено програмні засоби для підвищення ефективності та гнучкості процесу управління конфігураціями.

Особистий внесок здобувача. Усі наукові результати, що викладені у бакалаврській дипломній роботі, отримано автором самостійно. У наукових працях, опублікованих у співавторстві, автору належать такі результати: використання засобів для керування конфігураціями при розгорненні та масштабуванні електронних ресурсів [6], використання фреймворку gRPC для розробки гнучких клієнт-серверних додатків [7], розробка алгоритму запуску скриптів при управлінні конфігураціями [8].

Апробація матеріалів бакалаврської дипломної роботи. Результати роботи доповідалися на:

– Міжнародній науково-практичній інтернет-конференції «Електронні інформаційні ресурси: створення, використання, доступ» (2021 р., м. Вінниця);

– LI Науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії (2022 р., м. Вінниця);

– XXII Всеукраїнській науково-технічній конференції молодих вчених, аспірантів та студентів «Стан, досягнення та перспективи інформаційних систем і технологій» (2022 р., м. Одеса).

Публікації. За тематикою дослідження опубліковано 3 наукових праці у збірниках матеріалів конференцій.

1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Аналіз технологій для керування конфігураціями

Створення й підтримка сучасних електронних ресурсів потребує гнучкої інфраструктури, яка буде здатна швидко надати все необхідне для розгорнення програмного забезпечення. Велика кількість користувачів з різних частин світу можуть створити значне навантаження на апаратне й програмне забезпечення, через що архітектура й підходи до підтримки великих розподілених систем значно змінились. Вертикальне масштабування серверів та запуск ПЗ в єдиному екземплярі вже є ненадійним способом розгортання ресурсів, значно поступаючись таким принципам, як горизонтальне масштабування, кластеризація та висока доступність. Збільшена кількість серверів та більш складна конфігурація сучасних систем вимагає автоматизації частини процесів, в чому можуть допомогти додатки для керування конфігураціями.

Програмне забезпечення для керування конфігураціями слугує для підтримки електронних ресурсів у певному бажаному стані. Цей бажаний стан може включати в себе конкретні параметри операційної системи, встановлені додатки та їх конфігурацію. Необхідність в таких інструментах пов’язана відразу з декількома причинами.

По-перше, популярним є підхід зберігання всієї конфігурації інфраструктури на випадок апаратних збоїв або непрацездатності певного датацентру. Більшість інструментів для керування конфігураціями підтримує принцип «Інфраструктура як код» (IaC, Infrastructure as Code), за яким можна описати бажані параметри в звичайному файлі, який може зберігатися в системі контролю версій для відслідковування змін [9]. Тоді у випадку виникнення проблем з наявними серверами є можливість створити аналогічну систему на іншому апаратному забезпеченні. Також цей підхід може використовуватися в ситуації, коли необхідно оперувати декількома сервера з аналогічними конфігураціями. В такому випадку керування конфігураціями дозволяє

автоматизувати процес та перевикористовувати вже наявні конфігурації. Прикладом такого сценарію є горизонтальне масштабування навантаження. Розглянемо більш детально принципи масштабування.

Основні два типи масштабування апаратного забезпечення – це вертикальне й горизонтальне масштабування. При вертикальному змінюються параметри вже наявного сервера в більшу чи меншу сторону. Наприклад, програмне забезпечення для індексування записів або комплексних розрахунків почало показувати гіршу продуктивність через збільшення кількості клієнтів. В такому випадку вертикальне масштабування передбачає збільшення процесорних ресурсів або ОЗУ для поточного сервера. Приклад горизонтального масштабування – це створення нових додаткових серверів, які розділять загальне навантаження між собою. Проте, такий підхід вимагає відповідної підтримки всередині використовуваного програмного забезпечення.

Горизонтальне масштабування дозволяє не тільки зекономити за допомогою більш дешевих та малопотужних серверів, а ще й гнучко оптимізувати всю інфраструктуру – при зменшенні навантаження зайві сервери можуть бути використані для інших задач, що також називають автоматичним масштабуванням [10]. Сучасні технології віртуалізації, що використовуються в датацентрах, значно спрощують більшість операцій по виділенню нових ресурсів та створенню нових серверів для окремих задач, а керування конфігураціями спрощує першочергове налаштування, через що такі інструменти є особливо актуальними на даний момент.

Важливим аспектом програмного забезпечення для керування конфігураціями є автоматизація та пришвидшення виконання задач. Додатки, що підтримують кластеризацію та високу доступність, зазвичай мають специфічні способи налаштування, які пов’язані з підвищеними вимогами до безпеки. Наприклад, до цих задач може входити додаткове налаштування TLS-сертифікатів, маніпуляції з KeyStore-сховищами, складне генерування конфігураційних файлів з урахуванням параметрів декількох серверів. Ручне налаштування такого ПЗ може займати багато часу, а аналогічні дії в випадку

екстреної ситуації (наприклад, необхідність замінити непрацездатні сервіси новими) не є допустимими, адже кожна хвилина простою високонавантаженої системи може нести в собі втрати клієнтів та прибутку. Інструменти керування конфігураціями дозволяють зберігати всі необхідні для налаштування дані, проводити комплексні операції з генерації ресурсів, а також надають можливість попередньо розробити й протестувати сценарії розгортання програмного забезпечення або цілих систем на його основі.

Таким чином, програмне забезпечення для керування конфігураціями – це потужний інструмент для підтримки працездатності електронних ресурсів та керування цифровою інфраструктурою. Керування конфігураціями дозволяє використовувати підхід «Інфраструктура як код» та автоматизувати значну частину задач, що може спростити роботу DevOps-інженерів або системних адміністраторів, які підтримують великі цифрові продукти.

1.2 Порівняльний аналіз аналогів

Додатки для керування конфігураціями – це специфічне програмне забезпечення, основними користувачами якого є інші IT-спеціалісти, що мають потребу в керуванні та підтримці електронної інфраструктури. Наприклад, це можуть бути DevOps-інженери, системні адміністратори або SRE-інженери.

Популярними інструментами для керування конфігураціями є:

– Ansible;

– Chef;

– Puppet;

– SaltStack.

Розглянемо більш детально індивідуальні особливості кожного програмного рішення.

Ansible – один з найбільш популярних інструментів для керування конфігураціями від Red Hat, який написаний на Python та використовує формат YAML для опису конфігурацій. До переваг можна віднести простоту вивчення, відносно просту структуру опису конфігурацій та велику бібліотеку вже

існуючих рішень [11]. До недоліків відносяться погана підтримка ОС Windows та відсутність GUI в офіційній базовій редакцій. Просунутий функціонал та веб-інтерфейс доступні лише в складі платного набору інструментів Red Hat Ansible Automation Platform. Приклад даного Web-UI наведено на рисунку 1.1.

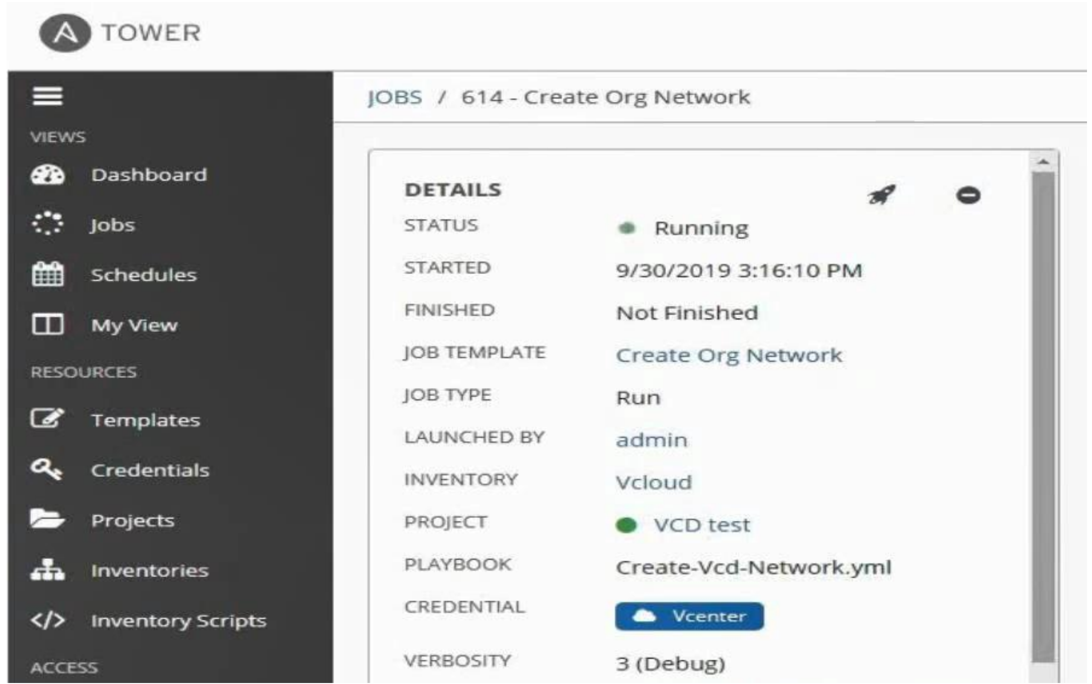


Рисунок 1.1 – Web-UI для Ansible Automation Platform

Chef – інструмент для автоматизації та керування інфраструктурою, що створений компанією Progress. Chef створено за допомогою Ruby, а для опису конфігурацій використовується власна DSL (Domain-Specific Language), яка також побудована на основі Ruby [12]. До переваг можна віднести наявність великої кількості додаткових модулів та високу гнучкість при описі конфігурацій. Основними недоліками є складність вивчення та малоінформативна документація.

Puppet – інший популярний інструмент, який спочатку був рішенням з відкритим кодом, а зараз підтримується однойменною компанією Puppet. Для розробки ПЗ аналогічно до Chef було використано Ruby, а конфігурації описуються на JSON-подібній DSL. До переваг можна віднести підтримку великої кількості різноманітних ОС та простоту інсталяції. Недоліками є велика

комплексна кодова база та декларативний стиль опису конфігурацій, що може ускладнити розробку складних скриптів для автоматизації [13].

Для використання Chef в графічному режимі можна скористатися додатком Chef Manage, який зображено на рисунку 1.2.

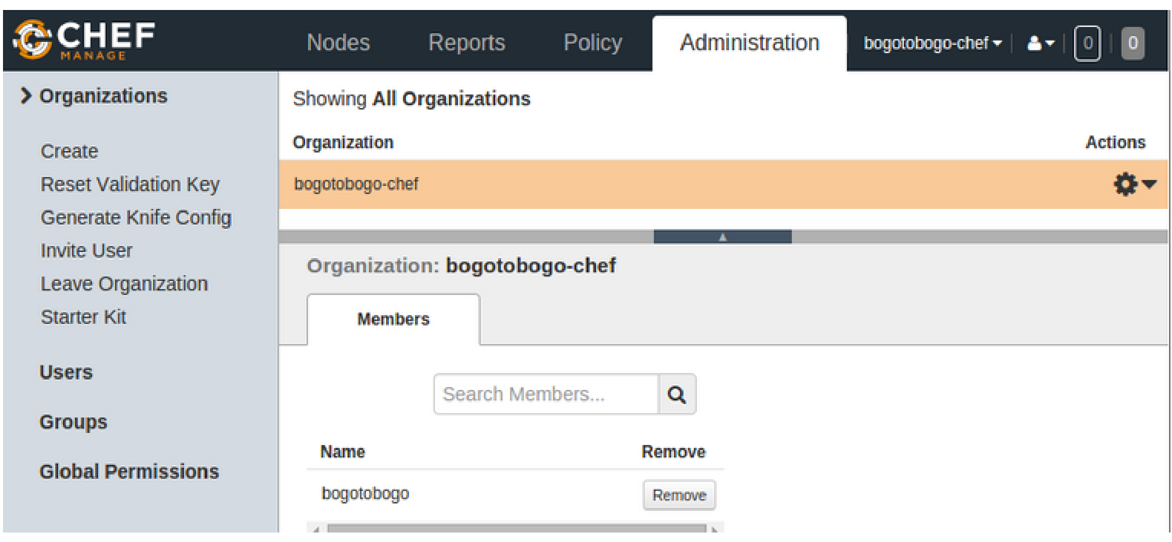


Рисунок 1.2 – Інтерфейс додатку Chef Manage

Для керування Puppet в графічному режимі існує Puppet Application Manager, який зображено на рисунку 1.3.

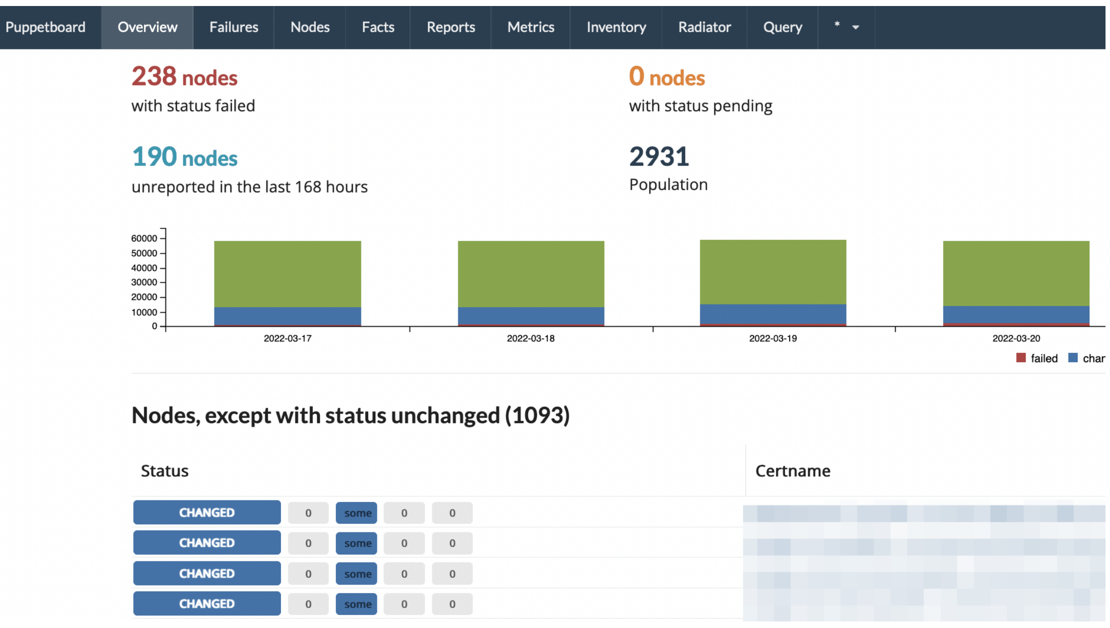


Рисунок 1.3 – Інтерфейс додатку Puppet Application Manager

SaltStack – додаток для керування конфігураціями з відкритим вихідним кодом, першочерговим розробником якого є Томас С. Гатч. Цей інструмент також розроблено з використанням Python, а конфігурації описуються за допомогою YAML. Значною перевагою є нестандартний спосіб зв'язку з використанням транспортної системи RAET, що дозволяє керувати системами з декількома тисячами серверами [14]. Недоліками є складність інсталяції та погана підтримка non-UNIX ОС.

Enterprise-версія SaltStack має графічний інтерфейс, який зображений на рисунку 1.4.

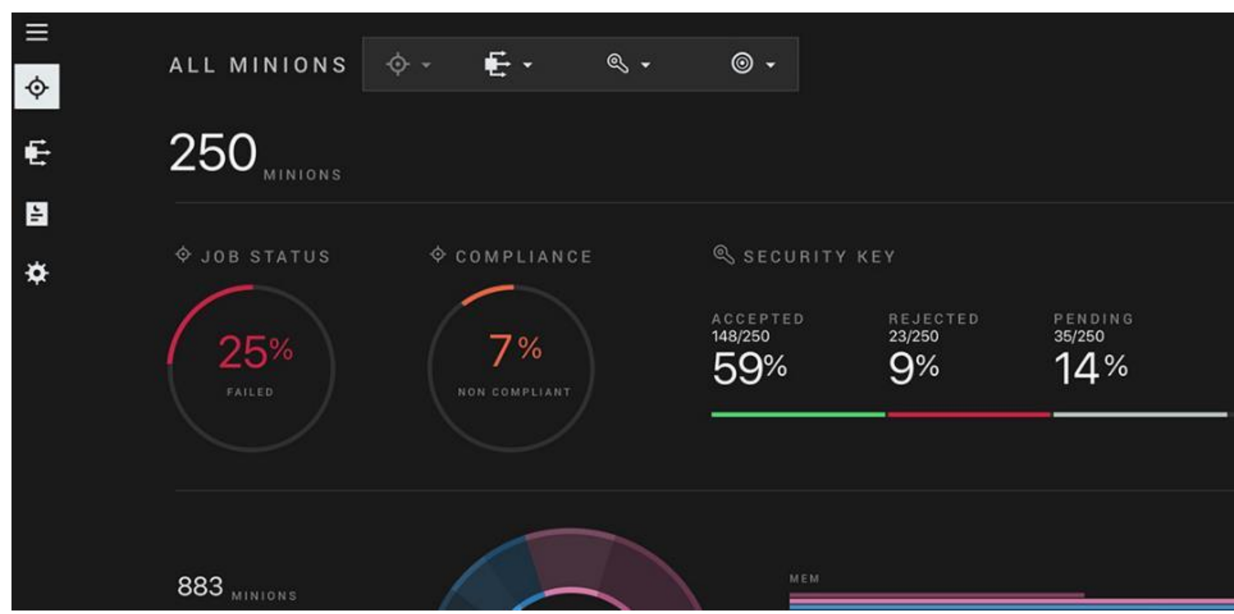


Рисунок 1.4 – Інтерфейс додатку SaltStack Enterprise

Аналіз аналогів дозволив визначити переваги й недоліки додатків в порівнянні з власним програмним продуктом «Detrint». Результати дослідження наведено в таблиці 1.1. Розглянемо більш детально окремі критерії оцінювання.

Більшість інструментів для керування конфігураціями використовують інтерфейс командного рядка (CLI) для взаємодії з користувачем. Основна причина для цього – це простота інтеграції додатків з різноманітними CI/CD-системами. Інші варіанти інтеграцій або наявність GUI зазвичай присутні лише в платних Enterprise-редакціях.

Таблиця 1.1 – Порівняльні характеристики програмних продуктів

Критерій	Ansible	Chef	Puppet	SaltStack	Detrint
CLI в базовій редакції	1	1	1	1	1
GUI в базовій редакції	0	0	0	0	1
Підтримка ОС Windows в якості контролера	0	0	0	0	1
Відсутність потреби встановлення агентів на керовані сервери	0,5	0	0	1	0
Підтримка конфігурацій з комплексною синхронізацією	0,5	0,5	0	0	1
Підсумок	2	1,5	1	2	4

Програмне забезпечення для керування конфігураціями зазвичай орієнтоване на використання з Linux та іншими UNIX-подібними операційними системами [6]. Частина продуктів, включаючи всі проаналізовані аналоги, підтримують можливість виконувати модулі та інструкції по конфігуруванню на машинах-хостах з Windows, але використання даної операційної системи в якості контролера неможливе.

Наступна різниця в додатках полягає в архітектурі зв’язку з керованими хостами. Найбільш популярними є Push та Pull архітектури. Push (Client Only) архітектура дозволяє встановити ПЗ для керування конфігураціями лише на контролер і почати працювати, а Pull (Client/Server) – вимагає встановлення додатків також і на керовані сервери. Підхід Client Only зазвичай використовує протоколи ОС для керування станом, тому може бути зручним у використанні, але не таким ефективним у швидкодії [6]. За даним критерієм Ansible отримує лише 0,5 бала через необхідність додаткового встановлення інтерпретатора Python для коректної роботи частини модулів.

Останнім критерієм є підтримка комплексних конфігурацій з використанням складних умов для синхронізації. Встановлення певного ПЗ або його налаштування може вимагати синхронізації кроків виконання між

керованими машинами. Наприклад, необхідно згенерувати файл на одному сервері та скопіювати його на інші, при цьому всі сервери повинні очікувати завершення цих операцій. Ansible та Chef отримують по 0,5 бала за даний критерій, адже через процедурний стиль опису конфігурацій вони дозволяють моделювати складні ситуації, але не мають вбудованих інструментів синхронізації для одночасного виконання різнорідних наборів конфігурацій.

Отже, з урахуванням отриманих результатів можна зробити висновок, що розробка власного програмного продукту «Detrint» є доцільною та дозволить отримати корисний функціонал, якого немає в аналогах.

1.3 Аналіз методів розв'язання поставленої задачі

Більшість додатків для керування конфігураціями мають схожу структуру та архітектурні рішення для реалізації розповсюдженого функціоналу. Проаналізуємо більш детально методи розв'язання задач в існуючих рішеннях.

Зазвичай в ПЗ для управління конфігураціями виділяють наступні складові:

– список серверів та додаткові дані для них;

– програмні модулі для вирішення атомарних задач;

– інструкції для конфігурації, які використовують модулі;

– опис конфігурації для серверів.

Розглянемо більш детально кожну складову.

Для роботи з серверами недостатньо знати лише IP-адресу або FQDN, в більшості випадків необхідно надати дані для встановлення захищеного підключення, описати додаткові умови та надати вхідні данні для конфігурацій, які залежать від параметрів конкретних хостів. Наприклад, в Ansible таку комбінацію списку серверів та змінних, що прив'язані до них, називають «Inventory».

Інвентар зазвичай описується в текстовому вигляді з використанням форматів YAML, INI, JSON та інших. Для визначення сервера зазвичай

достатньо додати його назву, адресу та облікові данні користувача для підключення (у випадку використання системних протоколів для з’єднання).

Опис кожного хоста можливо доповнити довільними змінними, які будуть включати додаткові вхідні дані для конфігурацій. Наприклад, необхідно вказати для конкретного сервера кількість доступної ОЗП для ПЗ або ж описати додаткові мережеві параметри, які будуть враховуватися інструкціями для конфігурації. Дані змінні окремих хостів називаються «host variables».

При автоматизації типових задач для великої кількості серверів зручніше описувати бажаний стан для всієї групи хостів, а не для окремих машин. Відповідно, існуючі рішення мають функціонал для опису таких сутностей в інвентарі. Група може включати не тільки список хостів, а ще й додаткові змінні для них, які найчастіше називають «group variables» [15]. На етапі виконання ПЗ для керування конфігураціями визначається фінальний список змінних для кожного сервера. Комбінуються дані всіх груп, до яких входить хост, та його власні змінні. Зазвичай параметри конкретного сервера мають більш високий пріоритет, ніж параметри груп.

Для власного програмного продукту було вирішено реалізувати описану логіку інвентаря з використанням власних алгоритмів, а в якості формату опису й подання інформації використовувати TOML. Це мінімалістичний формат опису конфігураційних файлів, який ставить за мету бути простішим за популярний формат YAML, але при цьому надавати ту ж саму простоту роботи й сприйняття.

Наступними важливими складовими програмного забезпечення для керування конфігураціями є модулі для вирішення атомарних задач та конфігураційні інструкції, які використовують ці модулі. В контексті керування конфігураціями модулями є короткі скрипти, які мають набір вхідних та вихідних даних, а результатом їх роботи є зміна стану системи. Зазвичай модулі проектуються якомога більш універсальними заради можливості їх повторного використання. В Ansible та SaltStack вони розробляються на Python, а в Chef та

Puppet – на Ruby. Крім того, в деяких випадках можуть використовуватися інші скриптові мови, які підтримуються ОС (наприклад, PowerShell для Windows).

Модулі призначені для невеликих змін, а їх дії та внутрішній зміст зазвичай приховані від кінцевого користувача, який оперує в більшому масштабі – використовує їх в якості будівельних блоків для створення комплексних скриптів для конфігурації систем [16]. Рівень абстракції для таких скриптів сильно відрізняється в залежності до продукту: Ansible та SaltStack використовують більш високорівневий підхід, а Chef та Puppet – навпаки дають більше контролю.

Ansible визначає бажану конфігурацію в форматі YAML, де окремими блоками вказується використовуваний модуль, вхідні дані для нього, а також інші додаткові параметри виконання. Обмін даними між модулями керується користувачем і обмежений вбудованими в Ansible функціями.

Chef та Puppet використовують власну DSL на основі Ruby. Хоч і з певними обмеженнями, але це дозволяє використовувати повноцінну мову програмування при описі конфігурації, що дає значно більше можливостей та дозволяє довільними шляхами оптимізувати рутинні задачі, з якими Ansible або SaltStack справляються не так гнучко. Значний недолік такого підходу – це більш високі вимоги до навичок кінцевого користувача.

З урахуванням зазначених особливостей було вирішено використовувати повноцінну мову програмування і для створення модулів, і для опису конфігурацій. Такий підхід надає більше можливостей по підвищенню ефективності та швидкодії при розробці конфігураційних скриптів, а потреба більш високої кваліфікації зі сторони користувачів не є значним мінусом, адже продукт орієнтований на DevOps-інженерів, системних адміністраторів, SRE-інженерів та інших IT-спеціалістів. Детальний вибір мови програмування наведено в розділі 3 пояснювальної записки.

Останньою складовою частиною створюваного ПЗ є робота з описами конфігурацій для серверів. Коли вже описаний інвентар з хостами, а всі необхідні модулі та інструкції на їх основі готові, то необхідно об’єднати ці

частини разом. Такий опис бажаного стану систем включає в себе сервер або їх групу та впорядкований список інструкцій по конфігуруванню (наприклад, може бути окрема інструкція по установці певного ПЗ або ж конфігуруванню мережевого екрану в ОС). В Ansible такий опис стану називається «Playbook», який будується за допомогою YAML, а в Chef – це «Recipe» з використанням Ruby.

Для створюваного ПЗ було вирішено використовувати TOML для опису стану системи. Робота з даною функцією є досить високорівневою, тому вигідно скористатися більш простим форматом опису.

Отже, було розглянуто базові складові частини та функції програмного забезпечення для управління конфігураціями. На основі отриманої інформації було описано методи та підходи до реалізації власного продукту.

1.4 Постановка задач розробки програмного забезпечення для управління конфігураціями

На основі аналізу переваг та недоліків існуючих продуктів для керування конфігураціями та з урахуванням типових архітектурних методів і підходів було сформовано наступний список задач:

– проаналізувати стан технологій для управління конфігураціями;

– визначити високорівневу структуру та архітектуру програмного продукту;

– розробити алгоритм формування змінних хостів для їх використання при застосуванні конфігурацій;

– розробити алгоритм запуску конфігураційний скриптів;

– розробити графічний інтерфейс користувача для створюваного продукту;

– розробити програмний додаток для керування конфігураціями;

– розробити модульні тести для автоматизованого тестування програмного додатку;

– провести ручне тестування програмного продукту з використанням комплексних сценаріїв виконання;

– розробити інструкцію користувача.

Технічне завдання на розробку наведено в додатку А.

1.5 Висновки

У першому розділі було розглянуто сучасний стан сфери керування програмною інфраструктурою та автоматизації, наведено причини існування та популяризації програмного забезпечення для керування конфігураціями, описано базові сценарії використання. Було проведено аналіз існуючих аналогів, описано їх переваги та недоліки, доведено доцільність розробки власного програмного продукту. Описано та проаналізовано існуючі підходи до вирішення поставленої задачі. З урахуванням отриманих даних було сформовано список основних задач, які необхідно виконати для розробки власного програмного забезпечення в рамках виконання бакалаврської дипломної роботи.

2 РОЗРОБКА АРХІТЕКТУРИ ТА АЛГОРИТМІВ РОБОТИ ПРОГРАМНОГО ПРОДУКТУ

2.1 Розробка архітектури програмного продукту

Додаток для управління конфігурація – це комплексне програмне забезпечення, яке повинно включати в себе не просто автоматичне виконання інструкцій на віддалених серверах, а й багато додаткових архітектурних компонентів для забезпечення максимальної гнучкості функціоналу та виконання усіх поставлених задач.

Розглянемо більш детально зроблені архітектурні рішення. Для кращого розуміння компонентів продукту та загального потоку виконання було створено схему, яку наведено на рисунку 2.1.

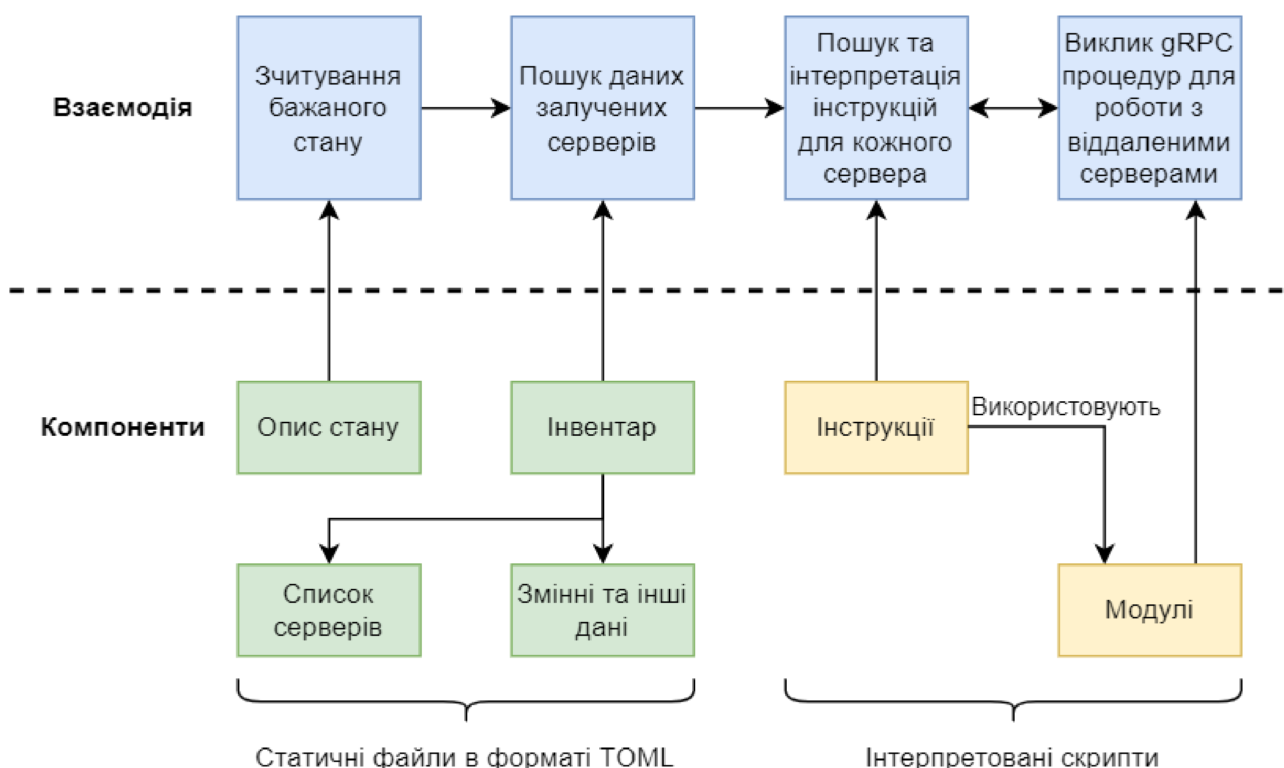


Рисунок 2.1 – Схема компонентів та їх взаємодії

Бажаний стан системи – це файл, який визначає які інструкції на яких саме серверах або їх групах необхідно виконати [17]. Аналогічний файл стану в Ansible називається плейбуком, а в Puppet – маніфестом. Як було описано в

попередньому розділі, дані для підключення до цих серверів та будь-які додаткові змінні для коректного виконання інструкцій зберігаються в наборі окремих файлів, які всі разом називаються інвентарем. Відповідно до розглянутих в підрозділі 1.3 методів розв'язання поставлених задач було вирішено використовувати формат TOML для опису бажаного стану серверів та інформації в інвентарі. Більш детально принципи роботи ПЗ з інвентарем розглянуто в підрозділі 2.2.

Після завантаження даних про бажаний стан та отримання даних про всі залучені сервери з інвентаря починається процес інтерпретації інструкцій, тобто йде фактичне виконання завдань на віддалених серверах. В Ansible такі інструкції називаються ролями, в Chef – рецептами. Інструкція або роль для управління конфігураціями – це зазвичай набір скриптів, який спрямований на досягнення конкретного результату, наприклад, встановлення та конфігурація веб-серверу [18]. Для цього використовуються модулі – готові компоненти-бібліотеки, які виконують атомарні дії.

Існує декілька варіантів роботи даного модуля ПЗ:

– повна інтерпретація інструкцій на хостах;

– часткове виконання на контролері з підключенням до віддалених серверів для атомарних змін.

Перший варіант передбачає упаковування всіх необхідних скриптів, їх пересилання на кожен сервер та виконання без прямого керування зі сторони контролера – йому передається лише проміжний статус системи та фінальний результат виконання. Другий спосіб передбачає виконання якомога більшої кількості операцій на контролері (наприклад, розрахунки на основі наданих змінних або генерація файлу конфігурації з шаблону), взаємодія з хостами відбувається лише всередині модулів для виконання елементарних операцій (наприклад, копіювання файлу або зчитування параметрів ОС). Повне виконання на віддалених серверах у першому способі може викликати додаткові складнощі при синхронізації дій між серверами, тому для створюваного програмного продукту обрано другий спосіб з частковим

виконанням на контролері. Процес інтерпретації та взаємодії між серверами детально описано в підрозділі 2.3.

Для взаємодії з керованими хостами було обрано фреймворк для віддаленого виклику процедур gRPC. Виклик віддалених процедур (RPC) – це спосіб клієнт-серверного зв’язку, що за способом використання подібний до виклику локальних функцій програми. Використання gRPC дозволяє досягти значно більшої швидкості обміну даних [7] та оптимізувати кількість запитів в порівнянні з вбудованими в ОС протоколами віддаленого управління, такими як SSH або WinRM. Перевагою фреймворку є підтримка великої кількості мов програмування, клієнт та сервер можуть бути реалізовані на різних мовах і при цьому успішно взаємодіями між собою. Вибір мови програмування для розробки програмного забезпечення описано в підрозділі 3.1.

З урахуванням описаних програмних компонентів та їх взаємодії було створено UML діаграму діяльності, яка описує загальний потік виконання додатку. Діаграма наведена на рисунку 2.2.

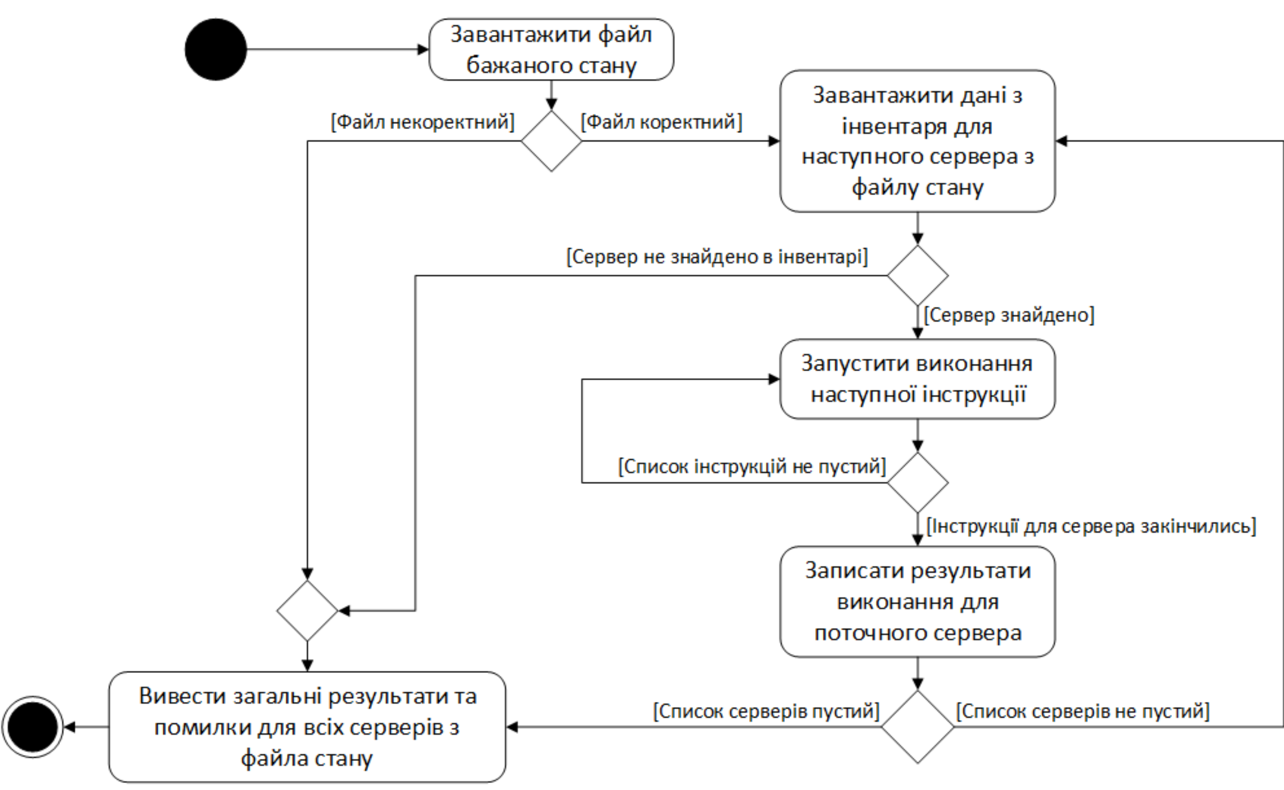


Рисунок 2.2 – Діаграма діяльності створюваного продукту

Отже, в підрозділі було описано загальну архітектуру створюваного програмного забезпечення, наведено схему компонентів та їх взаємодії. Описано загальні методи роботи додатків для управління конфігураціями та їх технічні особливості, розглянуто діаграму діяльності продукту.

2.2 Розробка алгоритму формування змінних хостів

Більшість додатків для управління конфігураціями передбачають роботу з великою кількістю різноманітних систем, на яких можуть виконуватись однакові задачі, тому можливість повторного використання інструкцій та модулів повинна бути частиною базового функціоналу. Змінні, що можуть бути визначені в інвентарі або інших конфігураційних файлах, дозволяють вирішити цю проблему повторного використання й надати достатню гнучкість при налаштуванні поведінки інструкцій.

Складність конфігурації великої кількості додатків та можливі проблеми на серверах з різним апаратним та програмним забезпеченням призвели до появи комплексної ієрархії змінних та логіки їх обробки. Наприклад, Ansible розрізняє 22 різні категорії змінних, які відсортовані в певному порядку і мають свій пріоритет – змінні вищого пріоритету перезаписують значення при конфлікті імен [19]. Загальна логіка в спрощеному форматі виглядає наступним чином:

– найменший пріоритет мають значення за замовчуванням, що визначені в ролях (інструкціях);

– різноманітні категорії змінних, що отримані з інвентаря (дані індивідуальних хостів більш пріоритетні, ніж дані груп);

– змінні, що оголошені в плейбуці (файлі бажаного стану системи);

– найвищий пріоритет мають змінні, що визначаються всередині ролей, такі як параметри модулів, блоків та інших конструкцій.

Варто відмітити, що Ansible та деякі інші продукти використовують більш високий рівень абстракції при описі конфігурацій: модулі – це скрипти на певній мові програмування, а самі ролі, які їх використовують, описують дії

в форматах YAML, JSON та інших. Ці формати добре підходять для серіалізації даних та зручні для сприйняття, але не можуть бути повноцінною заміною скриптових мов програмування. Через цю причину дані програмні продукти виокремлюють звичайні змінні та значення за замовчуванням у різні категорії з відмінними пріоритетами.

Для створюваного додатку було вирішено розробити власний алгоритм формування змінних серверів, який буде найкраще підходити для зроблених архітектурних рішень. Блок-схема алгоритму наведена на рисунку 2.3.

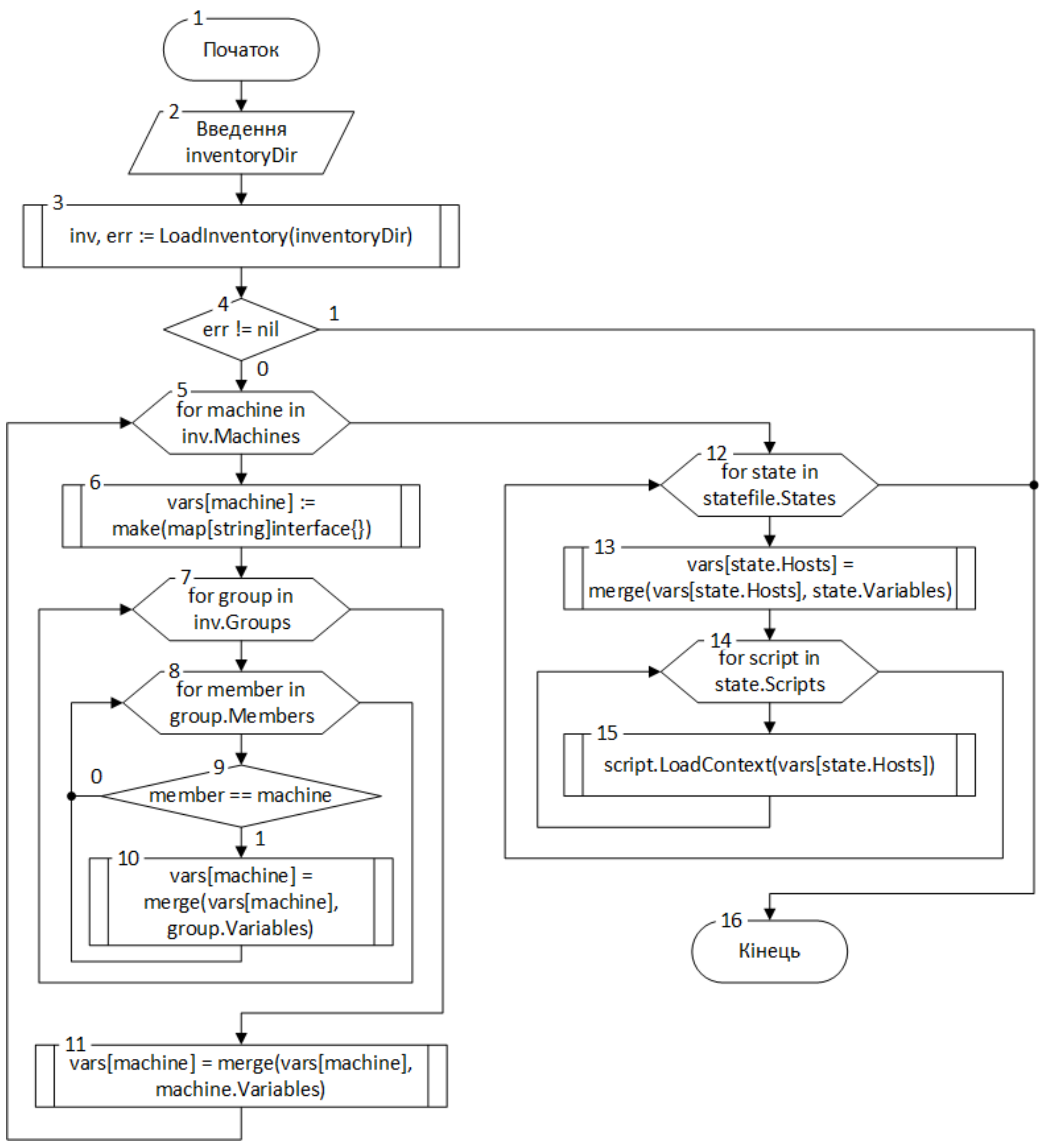


Рисунок 2.3 – Блок-схема алгоритму формування змінних хостів

Розглянемо більш детально кроки алгоритму:

Крок 1. Початок.

Крок 2. Через графічний інтерфейс користувача або параметри CLI користувач вводить шлях до директорії (inventoryDir) з файлами інвентаря.

Крок 3. Відкриваються і зчитуються усі файли в форматі TOML, що знаходяться у директорії за шляхом inventoryDir.

Крок 4. У випадку виникнення помилки зчитування інвентаря подальший процес управління конфігураціями неможливий, тому відбувається перехід до кроку 16. Інакше – продовження виконання і перехід до кроку 5.

Крок 5. Розпочинається цикл, який перебирає усі сервери, що зазначені в інвентарі. Якщо процес перебору закінчився – виконується перехід до кроку 12, інакше – перехід до кроку 6 та виконання тіла циклу.

Крок 6. В хеш-таблиці vars, яка зберігає змінні всіх серверів, створюється пустий допис для поточного хоста.

Крок 7. Починається виконання циклу, який перебирає усі групи, що описані в інвентарі. Виконання тіла циклу – перехід до кроку 8, закінчення перебору – перехід до кроку 11.

Крок 8. Розпочинається цикл з перебором членів поточної групи. Якщо виконання циклу закінчилось і всі члени перевірені – виконується перехід до кроку 7, інакше – перехід до кроку 9.

Крок 9. Виконується перевірка на співпадіння члена групи з поточним сервером. Якщо хост є частиною даної групи, то виконується перехід до кроку 10, а якщо ні – перехід до кроку 8.

Крок 10. В хеш-таблицю vars для поточного сервера копіюються усі змінні групи. Якщо декілька груп мають записи з однаковим іменем – групи, що обробляються пізніше, отримують вищий пріоритет та перезаписують значення.

Крок 11. Після перевірки й обробки всіх груп відбувається об’єднання уже наявних в структурі vars змінних з записами поточного сервера. Змінні хоста мають вищий пріоритет, ніж дані, отримані від груп.

Крок 12. Розпочинається цикл, що перебирає усі отримані дані про бажаний стан серверів. Кожен стан – це назва хоста або групи, список інструкцій для виконання та опціональний набір змінних. Виконання тіла циклу відбувається за переходом до кроку 13, а закінчення перебору – перехід до кроку 16.

Крок 13. Виконується об’єднання даних з хеш-таблиці vars зі змінними, що вказані в поточному описі стану. Тобто, записи в стані мають ще вищий пріоритет, ніж дані, що були отримані від інвентаря.

Крок 14. Виконується цикл, який перебирає усі інструкції поточного стану. Закінчення списку інструкцій викликає перехід до кроку 12, в іншому випадку відбувається перехід до кроку 15.

Крок 15. Усі змінні, що необхідні для роботи з обраним хостом, передаються в контекст інструкцій, після чого стають доступними для подальшої обробки в скриптах. Наступна логіка пріоритетів та перезапису значень всередині скриптів і модулів залежить від їх реалізації, тобто є підконтрольною кінцевим користувачам додатку.

Крок 16. Кінець.

Отже, в даному підрозділі було розглянуто проблему налаштування логіки виконання інструкцій при керуванні конфігураціями та її вирішення шляхом використання гнучкої системи змінних. Розглянуто приклади реалізації в інших популярних додатках, розроблено та описано власний алгоритм для формування змінних серверів.

2.3 Розробка алгоритму запуску конфігураційних скриптів

Взаємодія з віддаленими серверами – це найважливіша частина процесу керування конфігураціями, коли всі необхідні дані було введено та оброблено й виконання інструкцій вимагає з’єднання з хостом для виконання фактичних дій. Для кращого розуміння клієнт-серверної взаємодії необхідно більш детально розглянути загальний процес виконання скриптів усередині додатку.

Такі програмні продукти, як Ansible, Puppet та інші, використовують чітку структуру побудови інструкцій (ролей або маніфестів в термінології даних додатків) та їх виконання: роль складається з завдань, кожне завдання – це набір з однієї або декількох дій, що виконуються готовими модулями. Запуск завдань відбувається почергово, відповідно до порядку їх опису в файлі [20]. Модулі можуть слугувати для виконання широко спектру завдань: від оновлення значення змінної або створення файлу до модифікації системного реєстру або внесення нового запису в базу даних.

Такий підхід у аналогів спрощує сприйняття та високорівневу роботу з інструкціями, але ускладнює розробку комплексних конфігурацій, які потребують більш складної логіки та взаємодії між модулями. З урахуванням описаних проблем у створюваному ПЗ було вирішено використати інше архітектурне рішення [8]: і модулі, і самі інструкції знаходяться на одному рівні абстракції, тобто в обох випадках використовуються інтерпретовані скрипти. Для кращого розуміння різниці між підходами наведено схему на рисунку 2.4.

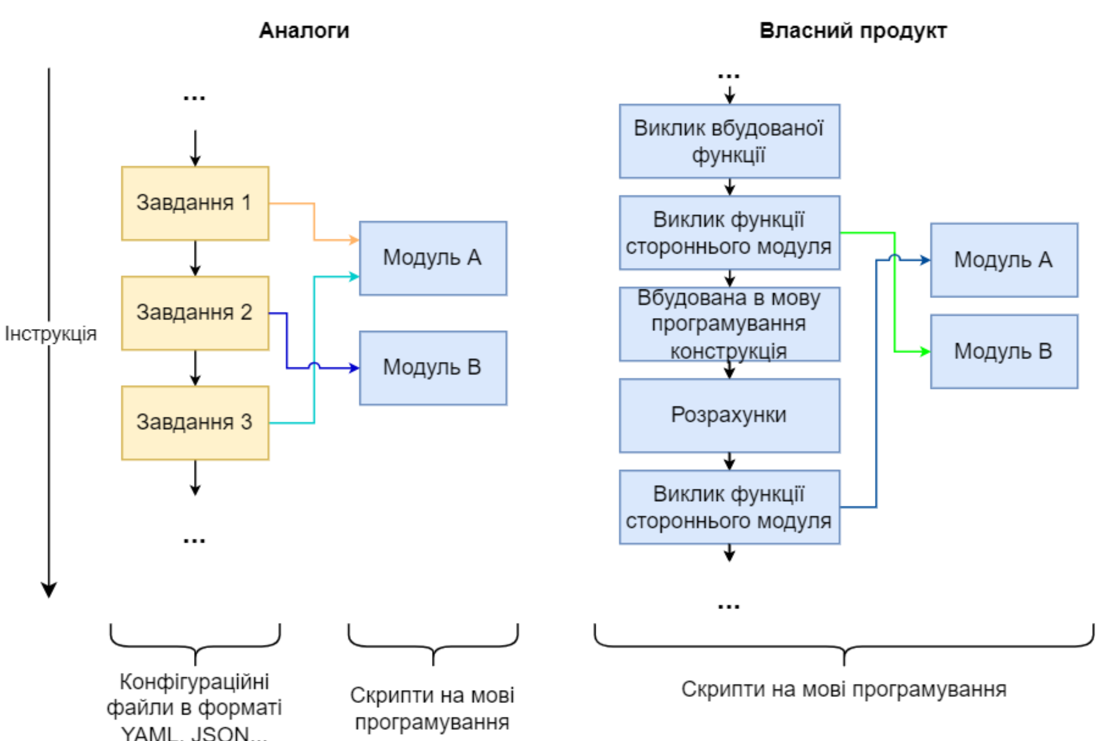


Рисунок 2.4 – Порівняння структури й процесу виконання інструкцій

Дане рішення дозволяє вирішити проблему створення комплексних конфігурацій та надає більше свободи кінцевим користувачам продукту. В такому випадку модулі представляють з себе не окремі скрипти, а аналог бібліотек для основних інструкцій.

Відповідно до описаного підходу було розроблено блок-схему алгоритму запуску конфігураційних скриптів, яку наведено на рисунку 2.5.

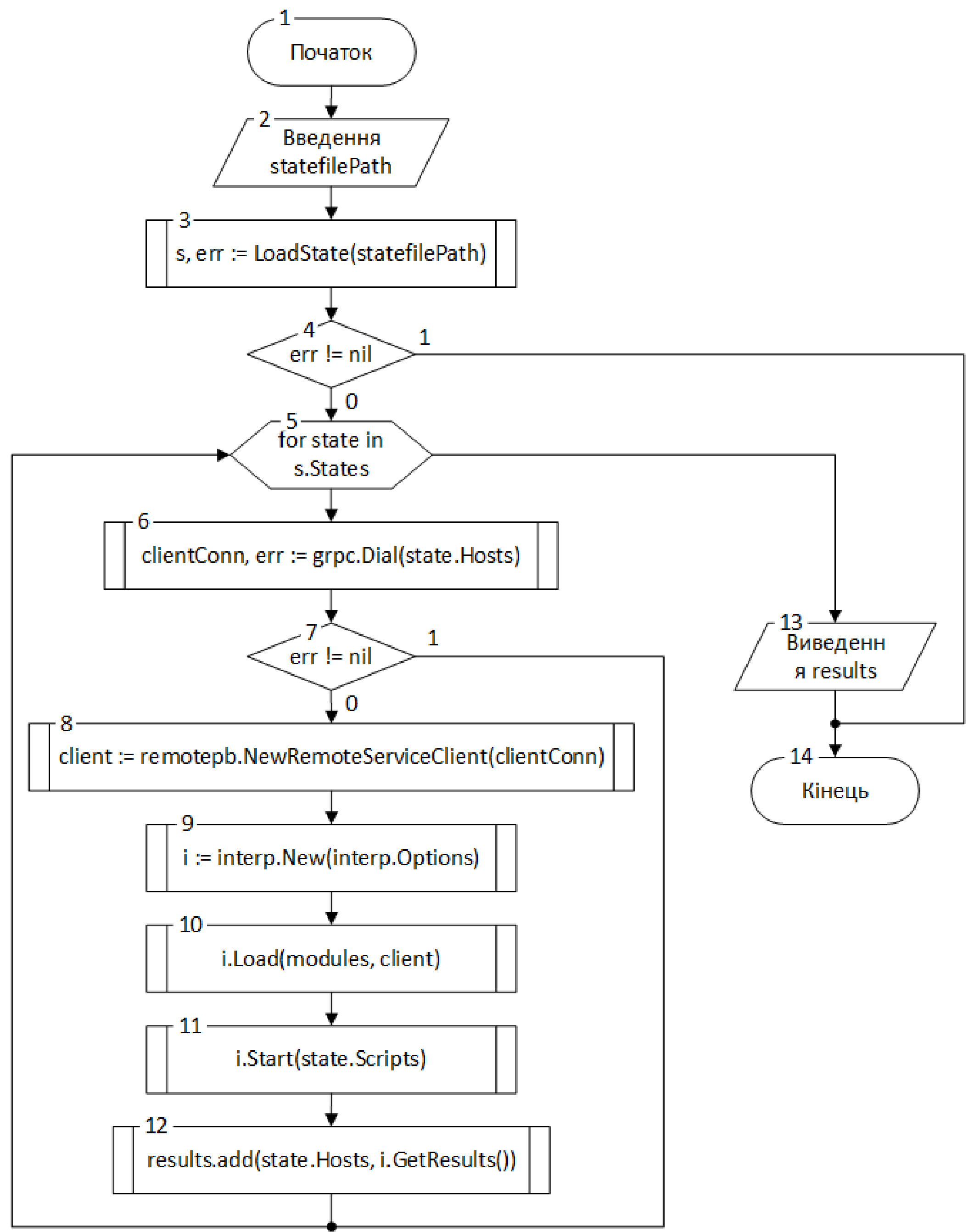


Рисунок 2.5 – Блок-схема алгоритму запуску конфігураційних скриптів

Розглянемо більш детально кроки алгоритму.

Крок 1. Початок.

Крок 2. Через графічний інтерфейс користувача або параметри CLI користувач вводить шлях до файлу, який описує бажаний стан серверів (які інструкції на яких серверах необхідно виконати), значення зберігається в змінній statefilePath.

Крок 3. Файл стану відкривається, дані з нього десеріалізуються в відповідний формат.

Крок 4. Якщо на попередньому кроці при відкритті файлу або десеріалізації даних відбулась помилка – відбувається перехід до кроку 14. Інакше – перехід до кроку 5.

Крок 5. Розпочинається цикл, який перебирає всі завантажені стани. Кожен стан – це комбінація з сервера або групи, скриптів, які треба там виконати, та додаткові змінні. Виконання тіла циклу розпочинається на кроці 6, завершення циклу викликає перехід до кроку 13.

Крок 6. Створюється з’єднання з цільовим сервером.

Крок 7. Якщо не вдалося з’єднатися з хостом – поточна ітерація циклу завершується і відбувається перехід до кроку 5. Інакше – виконання продовжується на кроці 8.

Крок 8. З використанням дескриптора з’єднання створюється gRPC-клієнт, який модулі можуть застосовувати для виконання дій на віддаленому сервері.

Крок 9. Створюється об’єкт для програмного доступу до інтерпретатора.

Крок 10. В контекст інтерпретатора завантажуються модулі та попередньо створений клієнт для доступу до сервера.

Крок 11. В інтерпретатор завантажуються та починають виконуватись скрипти.

Крок 12. Результати виконання та будь-які додаткові відомості для кожного стану зберігаються в окремий об’єкт.

Крок 13. Після закінчення циклу виводяться результати виконання.

Крок 14. Кінець алгоритму.

Таким чином, в даному підрозділі було розглянуто загальні принципи конфігурування систем за наданими користувачем інструкціями. Наведено приклади роботи аналогів, розглянуто переваги обраного технічного рішення з використанням інтерпретованих скриптів у якості інструкцій. Розроблено й детально описано блок-схему алгоритму запуску конфігураційних скриптів.

2.4 Розробка структури графічного інтерфейсу користувача

Графічний інтерфейс користувача (Graphical User Interface, GUI) – це тип інтерфейсу, який за допомогою візуальних об’єктів дозволяє користувачеві взаємодіяти з програмним забезпеченням на пристрої [21]. Ці візуальні об’єкти можуть включати в себе як графічні примітиви, так і більш комплексні елементи: наприклад, кнопки, піктограми, різноманітні меню та інші.

Комфорт роботи користувача всередині додатку забезпечується не тільки якістю розробленого функціоналу, а й зручністю та інформативністю GUI. Програмне забезпечення для управління конфігураціями вимагає комплексного інтерфейсу, який не дозволить користувачеві заплутатися в різноманітних елементах системи. Таким чином, було розроблено структурну схему головного вікна програми, яка використовує вкладки для відокремлення представлення різних функцій. Схему зображено на рисунку 2.6.

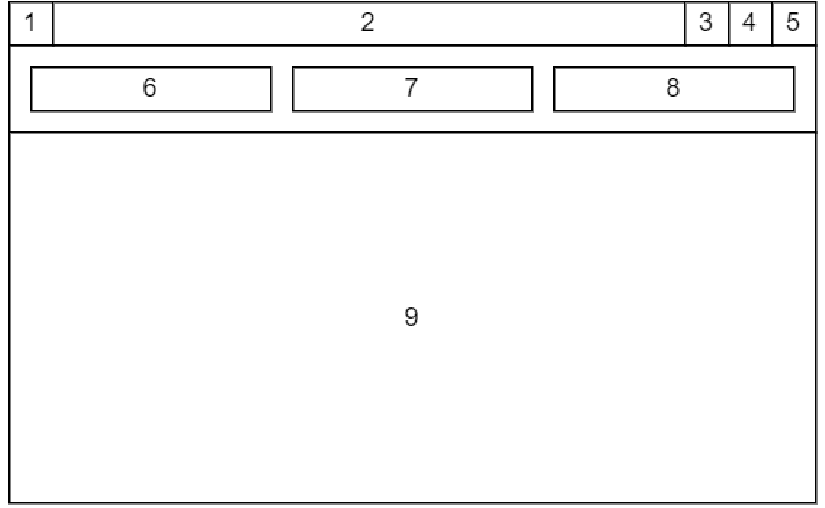


Рисунок 2.6 – Структурна схема головного вікна додатку

Основні елементи інтерфейсу головного вікна:

1) значок програми;

2) заголовок вікна з назвою програми;

3) системна кнопка згортання вікна;

4) системна кнопка розгортання вікна;

5) системна кнопка закриття програми;

6) кнопка переключення на вкладку «Інвентар»;

7) кнопка переключення на вкладку «Стан»;

8) кнопка переключення на вкладку «Виконання»;

9) вміст поточної вкладки.

Вміст центральної області вікна (9) залежить від обраної користувачем вкладки, що здійснюється за допомогою відповідних кнопок (6-8). Розглянемо більш детально схему інтерфейсу вкладки «Інвентар», що зображена на рисунку 2.7.

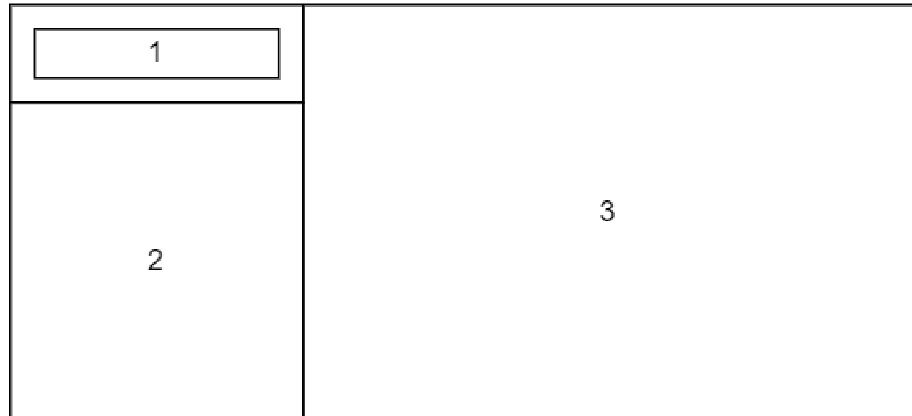


Рисунок 2.7 – Структурна схема вкладки «Інвентар»

Вкладка «Інвентар» включає в себе лише три елементи інтерфейсу:

1) кнопка відкриття файлу інвентаря;

2) меню зі списком серверів та груп;

3) область з додатковою інформацією по обраному серверу або групі.

Для перегляду вмісту інвентаря у цій вкладці користувачу необхідно спочатку за допомогою відповідної кнопки (1) завантажити TOML-файл з

даними. Якщо файл коректний і ПЗ успішно його зчитало, то список серверів та груп (2) буде оновлено. Обираючи потрібні пункти в меню користувач зможе побачити додаткову інформацію в правій частині вкладки (3).

Структурна схема вкладки «Стан» зображена на рисунку 2.8.

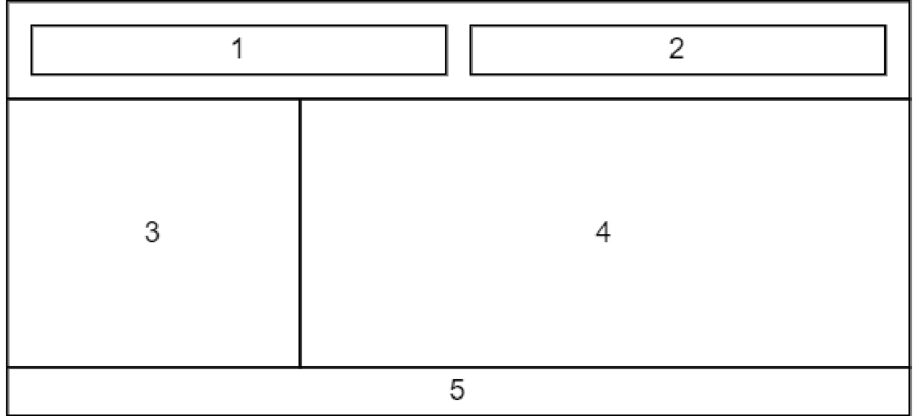


Рисунок 2.8 – Структурна схема вкладки «Стан»

Вкладка «Стан» включає в себе наступні елементи інтерфейсу:

1) кнопка відкриття файлу стану;

2) кнопка запуску виконання скриптів;

3) меню зі списком завантажених станів;

4) область з додатковою інформацією по обраному стану;

5) рядок статусу поточного файлу стану.

На цій вкладці користувачу спочатку необхідно завантажити файл з бажаним станом серверів за допомогою першої кнопки (1). При успішному зчитуванні інформації з файлу буде оновлено список станів (3), вибір будь-якого елемента списку дозволить переглянути в правій частині вікна (4) додаткову інформацію: сервер або групу, для якого будуть здійснюватися зміни, список інструкцій, додаткових змінних та інше. Рядок статусу (5) може містити додаткову інформацію про готовність до запуску виконання або ж наявні помилки (наприклад, якщо файл стану містить хост, який не визначено в інвентарі). Якщо жодних помилок немає, користувачу стає доступна кнопка запуску виконання скриптів (2).

Схема останньої вкладки «Виконання» наведена на рисунку 2.9.

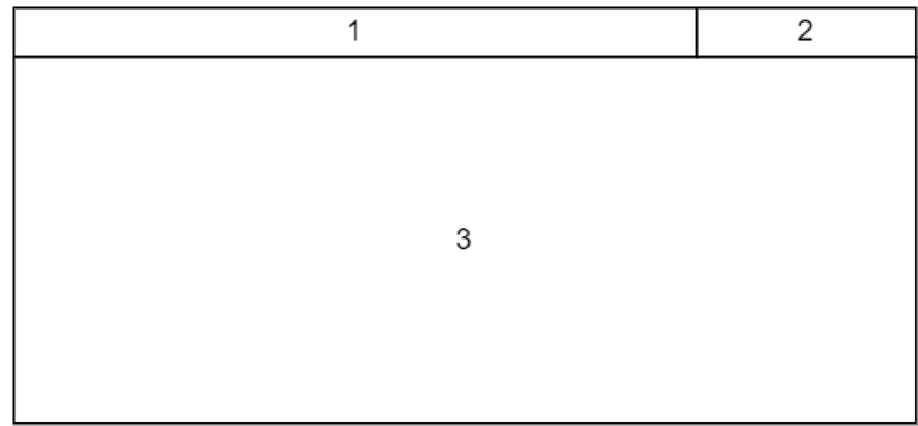


Рисунок 2.9 – Структурна схема вкладки «Виконання»

Елементи інтерфейсу, що знаходяться у цій вкладці, наступні:

1) текст-індикатор прогресу виконання;

2) загальний статус виконання;

3) логи скриптів.

Вкладка «Виконання» доступна користувачу лише після запуску виконання за допомогою відповідної кнопки на вкладці «Стан». Вона включає в себе текстову область з прогресом виконання скриптів (1) та індикатор загального статусу (2). Перший елемент відповідає за детальну інформацію з кількісними показниками (наприклад, скільки скриптів ще необхідно виконати), а другий – лише за кінцевий результат (досягнуто бажаного стану систем або ж виникли помилки). Центральна область вкладки (3) дозволяє переглянути загальні логи, що створюються скриптами та можуть допомогти при дослідженні помилок.

Отже, в даному підрозділі було розглянуто структуру графічного інтерфейсу користувача для створюваного додатку. Надано базові відомості про GUI, наведено особливості та вимоги при розробці інтерфейсу для програмного забезпечення для управління конфігураціями. Розглянуто й детально описано структурну схему головного вікна додатку, наведено

структурні схеми для окремих вкладок. Описано призначення окремих елементів інтерфейсу при використанні різних функцій програмного продукту.

2.5 Висновки

У другому розділі було детально розглянуто архітектуру створюваного програмного забезпечення, наведено схему компонентів та їх взаємодії, розроблено UML-діаграму діяльності. Розглянуто алгоритм формування змінних для хостів, описано причини використання та приклади реалізації у аналогів, наведено блок-схему розробленого алгоритму та розглянуто його окремі кроки. Також розроблено алгоритм запуску конфігураційних скриптів, наведено порівняння структури й процесу виконання інструкцій аналогів та власного продукту. Було розроблено структурні схеми інтерфейсу для головного вікна додатку та окремих вкладок, описано призначення елементів інтерфейсу при використанні різних функцій ПЗ.

3 РОЗРОБКА ПРОГРАМНИХ КОМПОНЕНТ ДОДАТКУ

3.1 Варіантний аналіз і обґрунтування вибору мови програмування

Вибір мови програмування – комплексне й складне питання, від якого залежать подальші процеси розробки, вибору інструментів та бібліотек, а в деяких випадках і засобів для тестування.

На етапі побудови архітектури створюваного продукту було зроблено рішення використовувати фреймворк gRPC для взаємодії з віддаленими серверами. Цей фреймворк не обмежує вибір мови програмування, адже офіційно підтримує більше десяти різних варіантів (в тому числі C#, C++, Java та інші) та має ряд неофіційних бібліотек для інших мов. Більш того, gRPC можна використовувати в складі будь-якої системи, яка може використовувати статичні або динамічні бібліотеки C/C++.

Таким чином, архітектура додатку не покладає обмежень на вибір мови програмування, тому необхідно провести детальний аналіз характеристик різних мов. Розглянемо більш детально C++, C#, Python та Golang.

C++ – найстаріша з обраних для аналізу мов, першу версію якої було створено ще в 1979 році. На відміну від C дозволяє використовувати ООП та всі основні концепції цієї парадигми [22]. Мова дозволяє розробляти кросплатформні додатки, основним обмеженням є лише доступність компілятора для потрібної операційної системи.

Перевагами C++ є надзвичайно висока швидкодія ПЗ, побудованого з її використанням, та можливість використовувати мову програмування на широкому спектрі пристроїв – існують компілятори навіть для різноманітних мікроконтролерів та SoC. До недоліків C++ можна віднести складні механізми роботи з пам’яттю та проблеми з інтеграцією і розширюваністю – останні стандарти мови досі не мають зручної системи для поширення і підключення модулів або бібліотек. Існують певні прототипи-напрацювання від великих компаній, які розроблять власні компілятори C++ під свої платформи, але в

цілому збірка проекта на цій мові може виявитись складною та нетривіальною для розробників з малим досвідом.

C# – мова програмування від Microsoft, перша версія якої вийшла в 2000 році. Першочергово розроблялася в якості конкурента для Java та як покращена версія C++, але вже багато років має свій власний вектор розвитку. C# є компільованою мовою програмування, але її принципи роботи відрізняються від C/C++: спочатку вихідний код програми компілюється в формат IL (Intermediate Language), який є портативним бінарним форматом, а потім IL-код перед виконанням компілюється в машинний код за допомогою CLR (Common Language Runtime) [23]. Такий підхід дозволяє досягти ще кращої та простішої в реалізації кросплатформності, хоча й вимагає деяких додаткових розрахунків під час запуску ПЗ.

Перевагами C# є спрощена високорівнева система роботи з пам’яттю та використання збірника сміття (GC), а також багата стандартна бібліотека, особливо в разі використання фреймворка .NET. Недоліки цієї мови програмування витікають з її переваг: більш дружелюбна до розробника система менеджменту пам’яті частково обмежує максимальну швидкодію розробленого ПЗ, якщо порівнювати з іншими компільованими мовами програмування. Також використання більшості можливостей .NET можливе лише на ОС Windows, при розробці додатків для інших систем необхідно використовувати .NET Core або альтернативні бібліотеки, що обмежує портативність створюваного ПЗ.

Python – високорівнева інтерпретована мова програмування, яка має довгу історію від 1991 року. Основний фокус Python полягає в спрощенні та пришвидшенні розробки: синтаксис мови сильно відрізняється від C++ або Java, а більш високий рівень абстракції великої частини бібліотек дозволяє швидко розробляти ПЗ. Python вимагає встановлення інтерпретатора для виконання програм і за довгий час існування мови їх було розроблено досить багато, тому додатки можна розробляти навіть для певних підтримуваних мікроконтролерів та портативних пристроїв, хоча й з певними обмеженнями.

Перевагами Python є обширна стандартна бібліотека та простота інтеграції з іншими мовами програмування – є можливість робити виклики до бібліотек, що були створені з використанням C/C++, Java та інших мов [24]. В якості основних недоліків можна назвати незадовільну швидкодію Python, що викликано необхідністю використовувати інтерпретатор, а також обмеженість сфер використання. Наприклад, цю мову програмування досить рідко можна зустріти при розробці мобільних додатків.

Go – компільована мова програмування, яка була розроблена в 2007 році співробітниками Google, а потім у 2009 році стала проектом з відкритим вихідним кодом. Golang – інша назва цієї мови програмування, що часто використовується для уникнення плутанини. Go намагається враховувати переваги й недоліки своїх конкурентів, тому має унікальну суміш характеристик: це компільована мова, яка лише трохи повільніше за C/C++ і в деяких випадках навіть наздоганяє їх, але при цьому дає можливість швидко розробляти ПЗ за допомогою спрощеного синтаксису та має високорівневу систему роботи з пам’яттю, яка використовує GC.

До переваг Golang можна віднести хорошу швидкодію, просту систему роботи з потоками виконання, кросплатформність і обширну стандартну бібліотеку, яка має готові рішення для тестування, роботи з мережею, зображеннями та іншим [25]. Основними недоліками є нестандартна реалізація парадигми ООП та синтаксис мови. Деякі концепції та шаблони програмування сильно відрізняються в порівнянні з Java або C#, а значні відмінності в синтаксисі вимагають додаткового часу для адаптації перед початком розробки.

З урахуванням описаних переваг та недоліків було зроблено порівняння мов програмування за певними критеріями, що описані в таблиці 3.1. Розглянемо більш детально ці критерії та виставлені оцінки.

Швидкодія розробленого ПЗ – всі розглянуті мови програмування є компільованими, за винятком Python. Додаток, написаний на інтерпретованому Python, об’єктивно не може працювати так само ефективно, як при використанні інших розглянутих мов.

Синтаксис та зручність розробки – за цим критерієм Python та Golang перемагають, так як мають спрощений синтаксис і зручні бібліотеки, що дозволяють за меншу кількість рядків коду досягти бажаного результату. C++ має докладний синтаксис, а для деяких можливостей необхідно додатково використовувати препроцесор, що лише ускладнює розробку. C# отримує лише 0,5 бала через подібний до C++ синтаксис, але при цьому мова має велику кількість синтаксичного «цукру», який значно спрощує деякі операції.

Таблиця 3.1 – Порівняння мов програмування

Критерій	C++	C#	Python	Go
Швидкодія розробленого ПЗ	1	1	0	1
Синтаксис та зручність розробки	0	0,5	1	1
Менеджмент залежностей (бібліотек)	0	1	1	1
Кросплатформність	1	0,5	0,5	1
Широкий вибір IDE та інших інструментів	1	0,5	1	0,5
Підсумок	3	3,5	3,5	4,5

Менеджмент залежностей – цей критерій має на увазі простоту керування проектом та бібліотеками-залежностями. C++ не надає стандартного інструменту для цього, хоча CMake і має певну популярність серед користувачів Linux.

Кросплатформність – простота розробки та збірки програмного забезпечення на різні платформи. C# вимагає CLR для роботи, а Python – установку інтерпретатора, тому ці мови отримують лише по 0,5 бала.

Широкий вибір IDE та інших інструментів розробки – в цій категорії C++ та Python мають найбільшу кількість ПЗ для вибору. Для C# через орієнтацію на ОС Windows та .NET Framework вибір обмежений. Golang набув популярності пізніше, якщо порівнювати з іншими мовами, тому кількість IDE для нього невелика.

За результатами порівняння найкраще для розробки програмного забезпечення для управління конфігураціями підходить Go. Ця мова

програмування популярна серед DevOps-інженерів та за допомогою неї було розроблено багато відомих продуктів, таких як Kubernetes, Hugo, CoreDNS та інших.

Таким чином, в даному підрозділі було розглянуто переваги й недоліки C++, C#, Python та Go. Було проведено порівняння мов програмування, результати зведено в відповідну таблицю, описано проаналізовані критерії. За результатами порівняння було обрано Go в якості мови програмування для розробки додатку.

3.2 Вибір середовища розробки

Інтегроване середовище розробки (IDE) – це програмне забезпечення, що спроектоване для об’єднання і виконання більшості задач розробника в одному додатку. IDE – це централізований інтерфейс доступу до всіх інструментів, які необхідні під час створення нового ПЗ. Зазвичай середовище розробки об’єднує в собі:

– редактор коду – текстовий редактор, який має додатковий функціонал для розпізнавання, підсвітки та аналізу коду;

– компілятор або інтерпретатор – ПЗ для трансляції та виконання вихідного коду додатку;

– налагоджувач (debugger) – ПЗ для виявлення помилок та аналізу процесу виконання програми;

– інструменти автоматизації – різноманітні додаткові програми для спрощення виконання типових задач (наприклад, роботи з git-репозиторієм).

Для Golang вибір IDE є не таким великим, як для інших мов програмування, але для розробників все одно доступно декілька опцій з різним функціоналом. Розглянемо більш детально LiteIDE, Visual Studio Code та GoLand.

LiteIDE – одне з найстаріших середовищ розробки для Go, що розвивається ще з 2012 року. Розповсюджується безкоштовно та має відкритий вихідний код, адже підтримується силами спільноти й немає єдиного

розробника-власника. До переваг IDE відносять швидкість роботи й активний розвиток функціоналу для роботи з Golang [26]. Основними недоліками є обмеженість редактору коду та недостатня розширюваність – плагіни для середовища є малочисельними та розробляються на C++. Графічний інтерфейс IDE наведений на рисунку 3.1.

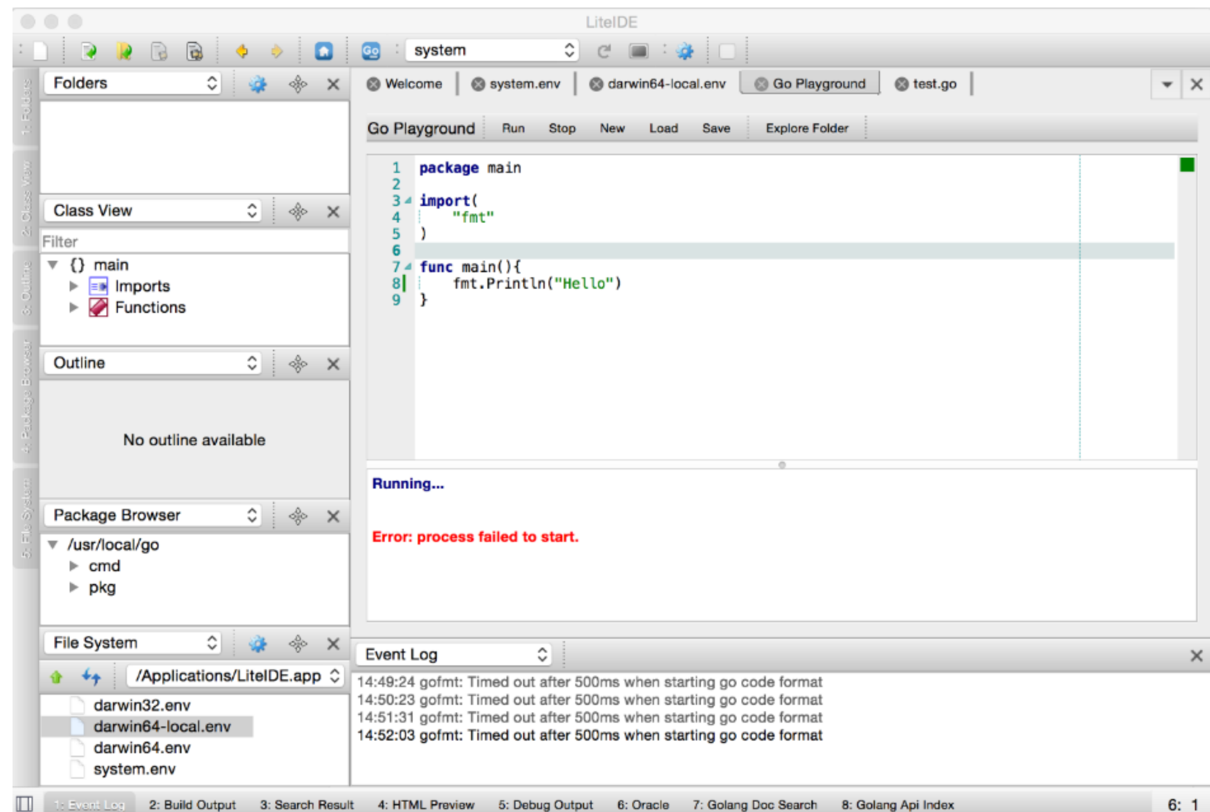


Рисунок 3.1 – Приклад інтерфейсу LiteIDE

Visual Studio Code – популярний редактор коду від Microsoft, який має відкритий вихідний код та активно розширюється спільнотою. Базовий функціонал редактора є невеликим, але обширний каталог плагінів дозволяє розширити його і використовувати VS Code для розробки додатків на практично будь-якій мові програмування. Golang в VS Code підтримується офіційним плагіном від Google [27]. До переваг можна віднести швидкодію IDE та зручну систему розширення функціоналу. Основними недоліками є обмежена система збірки проектів та недостатньо зручний аналізатор коду. Графічний інтерфейс середовища зображено на рисунку 3.2.

Рисунок 3.2 – Приклад інтерфейсу Visual Studio Code

GoLand – середовище розробки від JetBrains. Побудоване на основі платформи Intellij IDEA, як і інші IDE від цієї компанії, що дає доступ до потужного аналізатора коду та системи плагінів. На відміну від інших IDE GoLand має функціонал для віддаленої розробки, а також дозволяє ефективно редагувати код з просунутими інструментами рефакторингу. Основні переваги цього середовища розробки, як було описано до цього, полягають в потужних інструментах аналізу та редагування коду [28]. До недоліків можна віднести погану роботу IDE на малопотужних машинах, адже середовище побудоване з використанням Java та має підвищені потреби в RAM. Графічний інтерфейс GoLand наведено на рисунку 3.3.

Результати порівняння досліджених IDE за певними характеристика наведено в таблиці 3.2. Розглянемо більш детально критерії оцінювання.

Кросплатформність – чи підтримує певне середовище розробки найбільш популярні ОС. Усі розглянуті IDE є кросплатформними. Швидкість роботи – наскільки ефективно IDE використовує ресурси системи та швидко реагує на дії користувача. За цією характеристикою LiteIDE виграє, так як розроблена з

використанням С++ та має хорошу продуктивність. Інші IDE показують гіршу швидкість в ряді сценаріїв використання, хоча й мають більший функціонал.

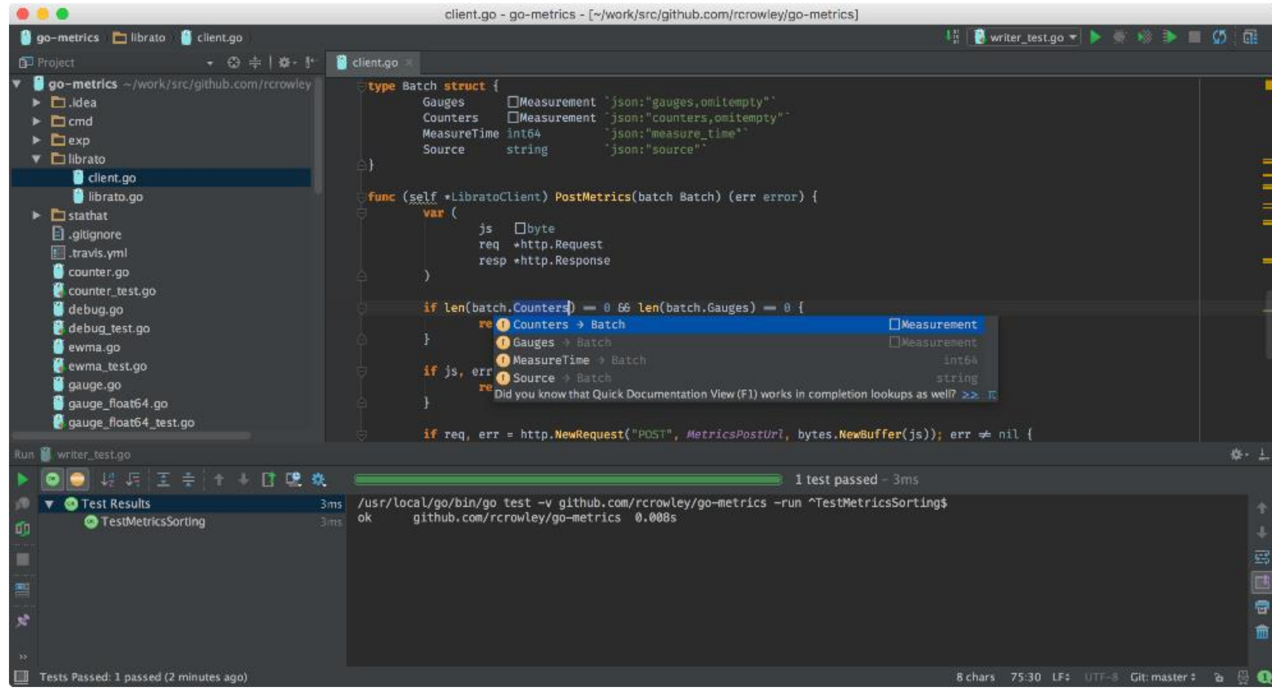


Рисунок 3.3 – Приклад інтерфейсу GoLand

Розширюваність – наскільки легко створювати плагіни для середовища розробки та наскільки великий каталог уже готових рішень. В цій категорії виграють VS Code та GoLand, система плагінів в LiteIDE є більш обмеженою.

Таблиця 3.2 – Порівняння інтегрованих середовищ розробки

Критерій	LiteIDE	VS Code	GoLand
Кросплатформність	1	1	1
Швидкість роботи	1	0,5	0,5
Розширюваність (система плагінів)	0,5	1	1
Інструменти для рефакторингу	0	0,5	1
Інструменти аналізу коду	0	0,5	1
Підсумок	2,5	3,5	4,5

Інструменти для рефакторингу – функціонал для роботи з кодом, його ефективного редагування. Найбільший список можливостей надає GoLand,

VS Code – частково відстає та сильно залежить від установлених користувачем плагінів, LiteIDE – найгірші опції для рефакторингу.

Інструменти для аналізу коду – наскільки добре IDE розуміє текстові файли з кодом, аналізує їх та надає можливість автодоповнення. LiteIDE має дуже обмежений функціонал. Аналізатор VS Code непоганий, але автодоповнення може працювати не найкращим чином у певних сценаріях використання. Вбудоване в GoLand рішення – одне з найкращих серед існуючих середовищ, аналізатор рідко помиляється при автодоповненні, система добре розуміє поточний контекст у коді.

Отже, в даному підрозділі було розглянуто базові відомості про IDE та проведено аналіз популярних середовищ розробки для мови програмування Go. Було досліджено LiteIDE, Visual Studio Code та GoLand, результати аналізу зведено в таблицю, окремі критерії якої детально описано. За результатами порівняння найкраще для розробки програмного продукту підходить IDE GoLand від JetBrains.

3.3 Програмна реалізація додатку

В процесі реалізації додатку для управління конфігураціями було розроблено велику кількість алгоритмів, включаючи описані в підрозділах 2.2 та 2.3, а також створено багато додаткових компонентів, що необхідні для коректної роботи функціоналу продукту. Розглянемо більш детально розроблений код додатку.

Типове програмне забезпечення для управління конфігураціями передбачає декілька точок входу, що необхідно для запуску в різних режимах функціонування. Наприклад, таке ПЗ часто може використовуватися в CI/CD-процесах, де його виклик частково або повністю автоматизовано й застосування GUI є недоцільним.

Golang реалізує декілька концепцій для зручного керування великою кількістю коду та його систематизації – це пакети та модулі. Пакети працюють в схожий до інших мов програмування спосіб – об’єднують частини коду

відповідно до обраного розробником контексту. Модулі мають більш важливе значення, адже крім об’єднання пакетів в один проект дозволяють більш гнучко керувати залежностями та версіонуванням модуля [25]. Отже, з урахуванням даних особливостей було прийняте рішення розділити створюваний додаток на декілька окремих модулів, що наведено на рисунку 3.4.

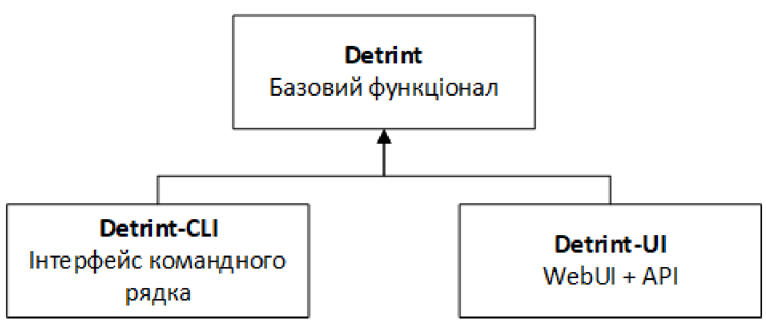


Рисунок 3.4 – Схема модулів додатку

Модуль «Detrint» – це основний модуль, який містить лише самий базовий функціонал та основні алгоритми додатку. Цей список включає в себе роботу зі структурами інвентаря та бажаного стану, керування інтерпретатором скриптів, різноманітну взаємодію з його контекстом виконання, реалізацію gRPC-сервісів для роботи модулів та інше. Будь-яка взаємодія з користувачем (не тільки через інтерфейс, а й через роботу з файлами інвентаря й стану) винесена в окремі модулі й повинна бути виконана до виклику основних функцій.

Модуль «Detrint-CLI» – це модуль, що реалізує інтерфейс командного рядка. Його основна мета полягає в наданні простого інструменту для завантаження файлів інвентаря та стану й подальшого виконання скриптів.

Модуль «Detrint-UI» – це більш комплексний модуль, що реалізує графічний інтерфейс користувача, який доступний через вікно додатку або у вигляді сторінки в браузері. Такий підхід також вимагає додаткової розробки REST API для взаємодії з основними функціями та даними додатку.

Таке розділення на модулі не тільки добре ізолює окремі функції та підвищує безпечність коду, а й дозволяти більш гнучко з ним працювати.

Наприклад, є можливість створити окремі виконавчі файли з GUI та CLI, в залежності від потреб, що відповідно зменшить розмір ПЗ. Крім того, основний модуль можна самостійно використовувати для розробки додаткових систем і інтеграцій в майбутньому – немає необхідності тягнути інші частини з зайвими залежностями.

Розглянемо більш детально вхідні точки додатку. CLI побудовано з використанням фреймворку «Cobra», який є популярним рішенням для Golang. З його використанням CLI будується з різноманітних компонентів-команд, які можуть містити додаткові елементи – аргументи та параметри, що передаються з командного рядка [29]. В даному випадку корінна команда містить лише назву додатку та його короткий опис, що наведено на рисунку 3.5, додаткова допоміжна інформація (наприклад, загальний список команд) буде автоматично згенерована фреймворком. Функція «Execute», що запускає корінну команду, виконується при старті додатку в функції «main».

```
8   var rootCmd = &cobra.Command{
9       Use:   "detrint-cli",
10      Short: "Detrint is like Ansible, only in Golang and with gRPC :)",
11      Long: `Detrint is a configuration management tool developed using Golang & gRPC.
12  The main focus of the tool is speed and usability for DevOps Engineers`,
13  }
14
15  func Execute() {
16      err := rootCmd.Execute()
17      if err ≠ nil {
18          fmt.Printf( format: "Failed to start app: %v", err)
19      }
20  }
21
```

Рисунок 3.5 – Лістинг корінної команди CLI та функції «Execute»

Для запуску виконання конфігураційних скриптів та обробки переданих користувачем файлів розроблено команду «start». Вона включає в себе додаткові іменовані аргументи для вказання шляху до файлів інвентаря та стану. Опис аргументів та додавання команди «start» в якості дочірньої для

корінної команди виконується в функції «init», яка автоматично запускається при ініціації пакету, що включає її. Лістинг функції «init» наведено на рисунку 3.6.

```
49  func init() {
50      startCmd.Flags().StringVarP(
51          &statePath,
52           name: "state",
53           shorthand: "s",
54           value: "",
55           usage: "Path to state file (required)",
56      )
57      err := startCmd.MarkFlagRequired( name: "state")
58      if err ≠ nil {
59          fmt.Printf( format: "An error occured: %v", err)
60      }
61
62      startCmd.Flags().StringVarP(
63          &inventoryPath,
64           name: "inventory",
65           shorthand: "i",
66           value: "",
67           usage: "Path to inventory file (required)",
68      )
69      err = startCmd.MarkFlagRequired( name: "inventory")
70      if err ≠ nil {
71          fmt.Printf( format: "An error occured: %v", err)
72      }
73
74      rootCmd.AddCommand(startCmd)
75  }
```

Рисунок 3.6 – Лістинг функції «init»

Зчитування та десеріалізація файлів відбувається за допомогою окремих функцій «loadInventory» та «loadState», що мають схожу структуру. Лістинг функції «loadInventory» наведено на рисунку 3.7.

```
77  func loadInventory(filename string) (*inv.Inventory, error) {
78      file, err := ioutil.ReadFile(filename)
79      if err ≠ nil : nil, fmt.Errorf("failed to read file data: %v", err) ↲
82
83      var i inv.Inventory
84      err = toml.Unmarshal(file, &i)
85      if err ≠ nil : nil, fmt.Errorf("failed to unmarshal file: %v", err) ↲
88      return &i, nil
89  }
```

Рисунок 3.7 – Лістинг функції «loadInventory»

Опис команди, задання виконавчої функції та інших параметрів виконується за допомогою спеціальної структури, що записується в змінну «startCmd» та використовується в функції «init». Частину лістингу структури наведено на рисунку 3.8.

```
var startCmd = &cobra.Command{
	Use:   "start",
	Short: "Start deployment",
	Long:  "Start deployment using provided state file and inventory",
	Run: func(cmd *cobra.Command, args []string) {
		stateSet, err := loadState(statePath)
		if err != nil {
			fmt.Printf("Failed to read state file: %v", err)
			return
		}

		inventory, err := loadInventory(inventoryPath)
		if err != nil {
			fmt.Printf("Failed to read inventory file: %v", err)
			return
		}

		logger, err := zap.NewProduction()
```

Рисунок 3.8 – Частина лістингу структури команди «start»

В кінці свого виконання команда «start» запускає метод «Deploy», який належить основному модулю «Detrint» та розпочинає процес управління конфігураціями. Перед розглядом основного модуля варто ознайомитись з другою вхідною точкою додатку, що відповідає за графічний інтерфейс користувача.

Модуль «Detrint-UI» побудовано з використанням бібліотеки React, що відповідає за графічний інтерфейс користувача, та засобів стандартної бібліотеки Golang, що дозволяють розробляти легкі та швидкі HTTP-сервери, які відповідають за обслуговування статичних файлів UI та доступ до REST API. Також додатково використано бібліотеки «julienschmidt/httprouter» та «rs/cors», які спрощують маршрутизацію запитів до API та роботу з CORS (Cross-Origin Resource Sharing).

Функція «main» даного модуля відповідає за створення та запуск HTTP-серверів для UI та API. Частина лістингу функції наведено на рисунку 3.9.

```go
20 ▶ func main() {
21       // WebUI part
22       uiLn, err := net.Listen( network: "tcp",  address: "127.0.0.1:7057")
23       if err != nil {
24           log.Fatalf("Failed to create WebUI listener: #{err}")
25       }
26
27       sub, err := fs.Sub(webBuild,  dir: "web/build")
28       if err != nil {
29           log.Fatalf("Failed to create fs subtree: #{err}")
30       }
31
32       uiServer := &http.Server{Handler: http.FileServer(http.FS(sub))}
33       go func() {
34           err := uiServer.Serve(uiLn)
35           if err == http.ErrServerClosed {
36               log.Println( v...: "WebUI server is closed!")
37           } else if err != nil {
38               log.Fatalf("Failed to start WebUI server: #{err}")
39           }
40       }()
```

Рисунок 3.9 – Частина лістингу функції «main» модуля «Detrint-UI»

REST API, що використовується для взаємозв’язку інтерфейсу та основного додатку, розділено на три частини: робота з інвентарем, робота з файлом стану, обробка й запуск конфігураційних скриптів. На рисунку 3.10 зображено частину лістингу пакету, що відповідає за API для інвентаря.

```go
15  const (
16      prefix      = "/inv"
17      machinesURL = prefix + "/machines"
18      machineURL  = machinesURL + "/:name"
19      groupsURL   = prefix + "/groups"
20      groupURL    = groupsURL + "/:name"
21  )
22
23  var (
24      inventory *inv.Inventory = nil
25  )
26
27  func Register(router *httprouter.Router) {
28      router.Handle(http.MethodGet, prefix, GetInventoryStatus)
29      router.Handle(http.MethodPost, prefix, UploadInventory)
30      router.Handle(http.MethodGet, machinesURL, GetAllMachines)
31      router.Handle(http.MethodGet, machineURL, GetMachine)
32      router.Handle(http.MethodGet, groupsURL, GetAllGroups)
33      router.Handle(http.MethodGet, groupURL, GetGroup)
34  }
35
36  func GetInventoryStatus(w http.ResponseWriter, r *http.Request, _ httprouter.Params) {
37      res := struct {
38          Available bool            `json:"available"`
39          Inventory *inv.Inventory  `json:"inventory,omitempty"`
```

Рисунок 3.10 – Частина лістингу файлу з REST API для роботи з інвентарем

Цей пакет включає в себе константи, що визначають URL для доступу, змінну для зберігання структури інвентаря, функцію реєстрації всіх методів всередині HTTP-маршрутизатора та реалізацію функцій самого API. Пакети з іншими частинами REST API мають аналогічну структуру.

API для запуску конфігураційних скриптів виконує функцію «Deploy» основного модуля, тому розглянемо її більш детально. Функція реалізує алгоритм запуску конфігураційних скриптів, що описаний в підрозділі 2.3. Дана функція передбачає використання вже десеріалізованих структур даних інвентаря та стану, а також вимагає об’єкт класу «Logger», що дозволяє більш гнучко налаштувати параметри логування до початку виконання скриптів.

Функція «Deploy» представляє з себе комплексний цикл, який починає виконання усіх станів, що були передані. Спочатку в лог записується назва поточного стану та хости, що визначені для нього. Це необхідно для спрощення орієнтації користувача в лог-файлах. Список хостів визначається окремим методом «ResolveHosts», що рекурсивно перевіряє інвентар. Лістинг початку функції «Deploy» показано на рисунку 3.11.

```
22  func (s Set) Deploy(i inv.Inventory, l *zap.Logger) {
23      for stateName, state := range s {
24          l.Info( msg: "Starting state deployment", zap.String( key: "state", stateName))
25
26          hosts := i.ResolveHosts(state.Hosts)
27          l.Info( msg: "Resolved host", zap.Any( key: "hosts", hosts))
```

Рисунок 3.11 – Лістинг коду, що відповідає за визначення серверів та логування даних

Структура з описом бажаного стану містить лише одне текстове поле для вказання назви сервера або групи. Метод «ResolveHosts» призначений для зіставлення цієї назви з вмістом переданого інвентаря. Спочатку перевіряється список серверів – якщо назви співпадають, то метод повертає дані потрібного хоста. Інакше – перевіряється список груп. Якщо група має підходящу назву, то її члени також рекурсивно перевіряються даним методом, що дозволяє вирішити ситуації з вкладеними групами. В кінці метод повертає результуючу

хеш-таблицю з усіма серверами, що входять в групу. Якщо передана назва не співпадає ні з одним сервером або групою (наприклад, користувач помилився при написанні файлу бажаного стану), то повертається пуста таблиця, а отже подальша обробка даного стану в функції «Deploy» не буде виконуватись. Лістинг методу зображено на рисунку 3.12.

```go
func (i Inventory) ResolveHosts(name string) map[string]Machine {
    if machine, ok := i.Machines[name]; ok {
        return map[string]Machine{name: machine}
    } else if group, ok := i.Groups[name]; ok {
        result := make(map[string]Machine)

        for _, member := range group.Members {
            resolved := i.ResolveHosts(member)
            for s, m := range resolved {
                result[s] = m
            }
        }

        return result
    } else {
        return map[string]Machine{}
    }
}
```

Рисунок 3.12 – Лістинг методу «ResolveHosts»

Після цього починається перебір усіх розпізнаних хостів у циклі. Спочатку за допомогою методу «Target» визначається кінцевий адрес машини та встановлюється gRPC-з’єднання, що показано на рисунку 3.13.

```go
        // Get correct target address
        target, targetErr := machine.Target()
        if targetErr ≠ nil {
            l.Error(
                msg: "Can't make target string",
                zap.String( key: "address", machine.Address),
                zap.Error(targetErr),
            )
            continue
        }

        // Create gRPC-connection for client creation
        conn, dialErr := grpc.Dial(target, grpc.WithTransportCredentials(insecure.NewCredentials()))
        if dialErr ≠ nil {
            l.Error(
                msg: "Failed to create client connection, skipping target",
                zap.String( key: "target", target),
                zap.Error(dialErr),
            )
            continue
```

Рисунок 3.13 – Лістинг коду, що відповідає за з’єднання з сервером

Метод «Target» перевіряє чи включають відомості про сервер довільний порт. Якщо ні, то додається стандартний, що дозволяє сформувати коректний адрес для підключення до хоста з використанням gRPC по протоколу HTTP/2 [7]. Після встановлення з’єднання починається робота з інтерпретатором скриптів – для цього використовується бібліотека «Yaegi», яка дозволяє інтерпретувати скрипти на Golang всередині вже скомпільованого ПЗ.

Спочатку створюється об’єкт для керування інтерпретатором, в якості вхідних параметрів йому передаються параметри доступу до системних бібліотек та шлях до модулів – директорії, де користувачі зможуть розміщувати свої довільні утиліти, бібліотеки та інше. Якщо конфігураційний скрипт не знайде певну функцію серед стандартних модулів, то додатково перевірить цю директорію. Після цього для поточного сервера генерується хеш-таблиця зі змінними, за що відповідає метод «GetMachineVars». Фрагмент даної частини коду функції «Deploy» наведено на рисунку 3.14.

```
52  // Create interpreter
53  interpreter := interp.New(interp.Options{
54      GoPath:       ModulesPath,
55      Unrestricted: true,
56  })
57
58  // Get vars for current machine
59  machineVars, varsErr := i.GetMachineVars(servername)
60  if varsErr ≠ nil {
61      l.Error( msg: "Failed to get machine vars, skipping host", zap.Error(varsErr))
62      closeErr := conn.Close()
63      if closeErr ≠ nil {
64          l.Error(
65              msg: "Failed to close client connection",
66              zap.String( key: "target", target),
67              zap.Error(closeErr),
68          )
69      }
70      continue
71  }
```

Рисунок 3.14 – Лістинг коду, що відповідає за створення інтерпретатора та змінні сервера

Метод «GetMachineVars» реалізує розглянутий в підрозділі 2.2 алгоритм формування змінних хостів. Першочергово перевіряється вміст інвентаря –

якщо передана назва серверу відсутня в ньому, то метод повертає відповідну помилку, адже відсутність певних змінних може порушити процес виконання скриптів. Після цього в циклі перевіряються усі групи – якщо сервер знайдено в якості члена, то всі змінні цієї групи додаються до результуючої таблиці. В кінці до таблиці копіюються змінні сервера. Такий порядок необхідний для вирішення конфлікту імен змінних – дані індивідуальної машини завжди будуть мати більш високий пріоритет. Лістинг методу наведено на рисунку 3.15.

```
29  func (i Inventory) GetMachineVars(servername string) (map[string]interface{}, error) {
30      result := make(map[string]interface{})
31
32      machine, ok := i.Machines[servername]
33      if !ok : result, fmt.Errorf("can't find machine with name '%s'", servername) ↲
37
38      for _, group := range i.Groups {
39          for _, member := range group.Members {
40              if member == servername {
41                  for k, v := range group.Variables {
42                      result[k] = v
43                  }
44              }
45          }
46      }
47
48      for k, v := range machine.Variables {
49          result[k] = v
50      }
51
52      return result, nil
53  }
```

Рисунок 3.15 – Лістинг методу «GetMachineVars»

Після цього відбувається завантаження стандартних модулів у контекст інтерпретатора. Пакет стандартної бібліотеки під назвою «reflect» дозволяє динамічно використовувати вже скомпільовані функції усередині скриптів. Для цього використовується окрема функція «Load», яка спочатку завантажує більшу частину стандартної бібліотеки, а потім додає в контекст усі пакети для роботи з віддаленими серверами з використанням gRPC. В якості вхідних параметрів передається вказівник на об’єкт для роботи з логами, хеш-таблиця змінних поточного сервера та попередньо створене з’єднання з хостом, яке

всередині інших пакетів буде використовуватися для створення gRPC-сервісів та виклику віддалених процедур. Фрагмент функції «Load» наведено на рисунку 3.16.

```
20  func Load(interpreter *interp.Interpreter, settings Settings) error {
21      stdlibErr := interpreter.Use(stdlib.Symbols)
22      if stdlibErr ≠ nil : stdlibErr ↵
25
26      symbols := make(map[string]map[string]reflect.Value)
27
28      symbols["github.com/ghotfall/detrint/builtin/util/util"] = util.Symbols(settings.Logger, settings.Vars)
29      symbols["github.com/ghotfall/detrint/builtin/shell/shell"] = shell.Symbols(settings.Connection)
30      symbols["github.com/ghotfall/detrint/builtin/file/file"] = file.Symbols(settings.Connection)
```

Рисунок 3.16 – Лістинг фрагменту функції «Load»

Типовий пакет, що передбачає використання в контексті скриптів, включає в себе функцію «Symbols», яка створює gRPC-клієнт для сервісів, а також експортує методи для інтерпретатора. На рисунку 3.17 зображено фрагмент пакету «file», який дозволяє віддалено працювати з файловою системою.

```
12  func Symbols(conn *grpc.ClientConn) map[string]reflect.Value {
13      client = filepb.NewFileServiceClient(conn)
14
15      return map[string]reflect.Value{
16          "GetStat": reflect.ValueOf(GetStat),
17          "Info":    reflect.ValueOf((*Info)(nil)),
18      }
19  }
20
21  type Info struct {...}
29
30  func GetStat(filename string) (Info, error) {
31      stat, err := client.GetStat(context.Background(), &filepb.StatRequest{Filename: filename})
32      if err ≠ nil : Info{}, err ↵ else {
35          return Info{
36              Name:    stat.GetName(),
37              Size:    stat.GetSize(),
38              Mode:    stat.GetMode(),
39              Time:    stat.GetTime(),
40              Dir:     stat.GetDir(),
41              PathErr: stat.GetPathErr(),
42          }, nil
43      }
44  }
```

Рисунок 3.17 – Лістинг фрагменту пакету «file»

По закінченню роботи з контекстом інтерпретатора запускається виконання конфігураційних скриптів. Цей фрагмент коду наведено на рисунку 3.18.

```
96          // Execute scripts
97          l.Info( msg: "Scripts to execute", zap.Strings( key: "scripts", state.Scripts))
98          for _, script := range state.Scripts {
99              l.Info( msg: "Executing script", zap.String( key: "script", script))
100
101             _, evalErr := interpreter.EvalPath( path: ScriptsPath + "/" + script + ".go")
102             if evalErr ≠ nil {
103                 l.Error(
104                     msg: "An failure occurred during script execution",
105                     zap.String( key: "script", script),
106                     zap.Error(evalErr),
107                 )
108             }
109         }
```

Рисунок 3.18 – Лістинг коду, що відповідає за виконання скриптів

Після обробки всіх скриптів відбувається закриття gRPC-з’єднання, розпочинається нова ітерація циклу для роботи з наступним сервером, що показано на рисунку 3.19.

```
111         // Close gRPC-connection
112         closeErr := conn.Close()
113         if closeErr ≠ nil {
114             l.Error(
115                 msg: "Failed to close client connection",
116                 zap.String( key: "target", target),
117                 zap.Error(closeErr),
118             )
119         }
```

Рисунок 3.19 – Лістинг коду, що відповідає за закриття з’єднання з сервером

По закінченню обробки всіх хостів процес приведення системи в бажаний стан можна вважати завершеним. Функція «Deploy» повинна опрацювати всі стани, що були описані в вхідному файлі від користувача.

Отже, в даному підрозділі було розглянуто програмну реалізацію додатку. Описано модульну архітектуру програмних компонентів, розглянуто вхідні точки ПЗ. Було описано роботу з вхідними файлами додатку, розглянуто основні функції, що відповідають за роботу з бажаними станами серверів та інвентарем. Повний лістинг наведено в додатку В.

3.4 Висновки

У даному розділі було проведено варіантний аналіз і обґрунтування вибору мови програмування. Було проаналізовано переваги й недоліки C++, C#, Python та Go. Для розробки програмного продукту було обрано мову Go. Було досліджено ринок популярних IDE для цієї мови програмування, проаналізовано середовища розробки LiteIDE, VS Code та GoLand. На основі розглянутих критеріїв було прийнято рішення використовувати JetBrains GoLand для розробки додатку. Було детально розглянуто й описано процес розробки основних програмних компонентів продукту «Detrint».

4 ТЕСТУВАННЯ ПРОГРАМИ

4.1 Аналіз методів тестування програмного забезпечення

Тестування програмного забезпечення – це процес технічного дослідження, що призначений для вияву інформації про якість продукту відносно контексту, в якому він має використовуватися [30]. Серед причин для тестування продукту можна виділити:

– зменшення фінансових витрат – якісне тестування дозволяє виявити та виправити проблеми на ранніх етапах розробки, а також спростити підтримку продукту після його випуску;

– покращення безпеки – тестування програмного забезпечення дозволяє виявити проблеми з неавторизованим доступом до функціоналу та даних ще до того, як продукт буде в використання кінцевих користувачів;

– охоплення ринку – додаток може добре працювати на обладнанні та ОС, які використовуються для розробки, але кінцеві користувачі зазвичай мають значно ширший та більш різноманітний асортимент пристроїв, тому вчасне тестування сумісності зробить створюваний продукт більш доступним для них;

– задоволення споживача – велика кількість недоліків на момент випуску ПЗ може спричини сильні репутаційні втрати, що будуть впливати на продукт навіть після виправлення більшості проблем;

– покращення процесу розробки – добре налаштовані процеси розробки та тестування, що працюють паралельно й з урахуванням одне одного, лише пришвидшують загальний темп роботи над продуктом;

– покращення розширюваності ПЗ – тестування дозволяє розробникам отримувати швидкий зворотній зв’язок з приводу останніх змін в коді, тому значно складніше запровадити невдалі архітектурні рішення;

– покращення швидкодії – регулярне тестування ПЗ (особливо автоматизоване) дозволяє швидко виявити вузькі місця поточної реалізації та значно раніше почати роботу над їх виправленням.

Існує велика кількість видів тестування, що відрізняються за своїми цілями та методами. Деякі з них можна віднести відразу до декількох категорій, адже вони можуть включати в себе кілька тісно пов'язаних задач. Найчастіше види тестування класифікують за:

– об'єктом тестування (функціональне, навантажувальне, стрес-тестування, тестування стабільності та інші);

– знанням системи (тестування «чорного/білого/сірого ящика»);

– ступенем автоматизації (ручне, автоматизоване, напівавтоматизоване);

– ступенем ізольованості компонентів (компонентне, інтеграційне, системне).

З урахуванням архітектурних та функціональних особливостей розроблюваного додатку вирішено скористатися компонентним тестуванням та тестуванням «чорного ящика». Розглянемо ці види більш детально.

Компонентне тестування – вид тестування, що має за мету перевірити окрему частину програмного забезпечення в ізоляції від інших компонентів. Часто його також називають модульним тестування (unit testing). Цей вид тестування добре підходить для валідації окремих програмних функцій та методів додатку, коли можна максимально обмежити взаємозв'язок з іншими системами або зімітувати їх діяльність [31], тому його обрано для перевірки працездатності основного модуля «Detrint».

Тестування «чорного ящика» – вид тестування, що перевіряє зовнішні інтерфейси додатку та сценарії кінцевих користувачів, які не повинні нічого знати про внутрішню будову ПЗ [32]. Це дозволяє швидко визначити проблеми, що можуть виникнути при типових діях клієнтів, тому цей вид тестування обрано для перевірки працездатності модулів «Detrint-CLI» та «Detrint-UI».

Отже, в даному підрозділі розглянуто базові відомості про тестування ПЗ, описано головні причини й переваги тестування продукту. Було розглянуто класифікацію видів тестування, для перевірки працездатності додатку обрано компонентне тестування та тестування «чорного ящика».

4.2 Тестування розробленого програмного продукту

Для програмного забезпечення для управління конфігураціями було розроблено ряд модульних тестів, що перевіряють реалізацію алгоритмів роботи з інвентарем, формування змінних хостів та інших функцій додатку з комплексною логікою. На рисунку 4.1 наведено фрагмент коду одного з модульних тестів.

```
105            {
106                name:    "no_server",
107                input:   "server_5",
108                want:    map[string]interface{}{},
109                wantErr: true,
110            },
111        }
112        for _, tt := range tests {
113            t.Run(tt.name, func(t *testing.T) {
114                got, err := i.GetMachineVars(tt.input)
115                if (err ≠ nil) ≠ tt.wantErr {
116                    t.Errorf( format: "GetMachineVars() error = %v, wantErr %v", err, tt.wantErr)
117                    return
118                }
119                if !reflect.DeepEqual(got, tt.want) {
120                    t.Errorf( format: "GetMachineVars() got = %v, want %v", got, tt.want)
```

Рисунок 4.1 – Частина лістингу модульного тесту для перевірки формування змінних хостів

Всього було розроблено 18 тестів з різними вхідними даними та сценаріями виконання. Результати автоматизованого компонентного тестування наведено на рисунку 4.2.

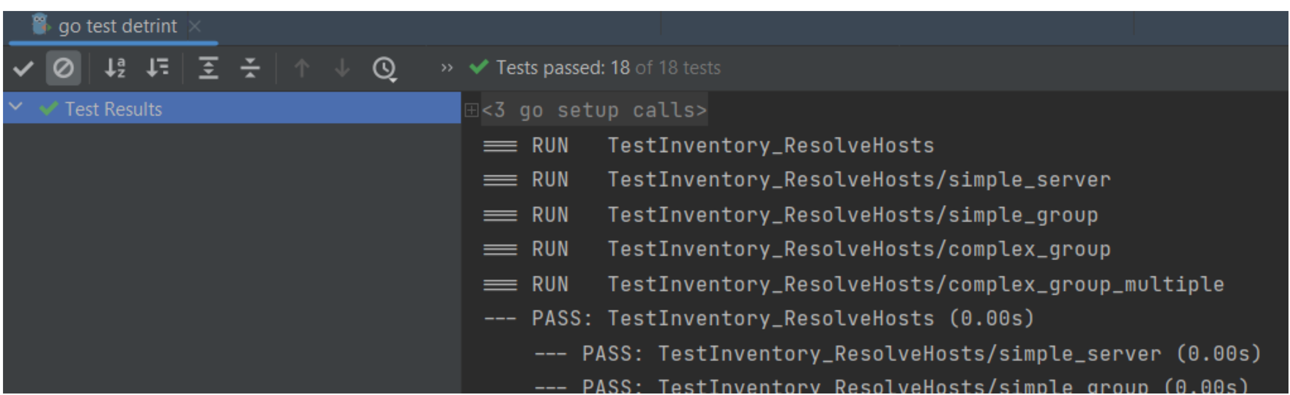


Рисунок 4.2 – Результати компонентного тестування

Тестування за методом «чорного ящика» було розділено на дві частини: перевірка роботи додатку через CLI та через GUI. Для обох сценаріїв було розроблено TOML-файли інвентаря та стану. Інвентар включає два сервери та дві групи з додатковими змінними, а стан – запуск скрипту з трьома різними модулями на одному з хостів. Ці тестові файли дозволяють перевірити роботу ПЗ для машини-контролера та машин-агентів, а також протестувати виконання інструкцій.

Тест-кейс №1 «Запуск конфігураційних скриптів через CLI»:

1) на кожному сервері запустити програму-агент «agent.exe» для роботи з контролером;

2) скопіювати попередньо підготовлені TOML-файли на контролер;

3) на машині-контролері запустити виконання інструкцій командою в форматі «detrint-cli.exe start -i <шлях до файлу інвентаря> -s <шлях до файлу стану>»;

4) перевірити логи CLI на відсутність помилок.

Приклад виконання тест-кейсу №1 наведено на рисунку 4.3.

```
Администратор: C:\Windows\System32\cmd.exe - agent.exe

E:\projects\goland\detrint\build>agent.exe
2022-05-09T16:11:19.054+0300    INFO    shell/shell.go:21       Registered new service  {"service": "shell"}
2022-05-09T16:11:19.062+0300    INFO    file/file.go:20 Registered new service  {"service": "file"}
2022-05-09T16:11:19.062+0300    INFO    server/server.go:28     Starting gRPC server...
2022-05-09T16:15:30.760+0300    INFO    shell/shell.go:29       Method shell.execute is called  {"request": "script:\"[S
ystem.Net.Dns]::GetHostName()\""}
2022-05-09T16:15:30.926+0300    DEBUG   shell/shell.go:56       Execution of shell.execute finished     {"stdout": "DESK
TOP-RC0NMC0\r\n", "stderr": "", "code": 0}
2022-05-09T16:15:30.927+0300    INFO    file/file.go:28 Method file.getstat is called   {"request": "filename:\"c:\\\\wi
ndows\\\\system32\\\\drivers\\\\etc\\\\hosts\""}
2022-05-09T16:15:30.928+0300    DEBUG   file/file.go:49 Execution of shell.execute finished     {"result": "name:\"hosts
\"  size:1325  mode:438  time:\"2022-05-03T12:08:29+03:00\""}

Администратор: C:\Windows\System32\cmd.exe

E:\projects\goland\detrint-cli\build>detrint-cli.exe start -i test.toml -s play.toml
{"level":"info","ts":1652104000.8371406,"caller":"state/base_structs.go:24","msg":"Starting state deployment","state":"C
heck module functionality"}
{"level":"info","ts":1652104000.83767,"caller":"state/base_structs.go:27","msg":"Resolved host","hosts":{"server1":{"Add
ress":"127.0.0.1","Username":"Test","Password":"Adm123","Variables":null}}}
{"level":"info","ts":1652104000.8481805,"caller":"state/base_structs.go:97","msg":"Scripts to execute","scripts":["scrip
t1"]}
{"level":"info","ts":1652104000.848705,"caller":"state/base_structs.go:99","msg":"Executing script","script":"script1"}
{"level":"info","ts":1652104000.8492298,"caller":"reflect/value.go:556","msg":"Script 1 is started!"}
{"level":"info","ts":1652104000.8492298,"caller":"reflect/value.go:556","msg":"Variable value: 4.55E+01"}
{"level":"info","ts":1652104001.0131135,"caller":"reflect/value.go:556","msg":"DESKTOP-RC0NMC0\r\n"}
{"level":"info","ts":1652104001.0141976,"caller":"reflect/value.go:556","msg":"1325"}
```

Рисунок 4.3 – Результати виконання тест-кейсу №1

Для тестування модуля «Detrint-UI» використано аналогічні TOML-файли, що дозволяють повноцінно перевірити процес управління конфігураціями. Розроблено окремий тест-кейс, що враховує особливості роботи з графічним інтерфейсом.

Тест-кейс №2 «Запуск конфігураційних скриптів через GUI»:

1) на кожному сервері запустити програму-агент «agent.exe» для роботи з контролером;

2) скопіювати попередньо підготовлені TOML-файли на контролер;

3) на машині-контролері запустити додаток «detrint-ui.exe»;

4) перейти на вкладку «Інвентар»;

5) завантажити файл інвентаря за допомогою відповідної кнопки;

6) перейти на вкладку «Стан»;

7) завантажити файл стану за допомогою відповідної кнопки;

8) натиснути кнопку запуску виконання скриптів;

9) перейти на вкладку «Виконання»;

10) перевірити логи на відсутність помилок.

Приклад виконання тест-кейсу №2 наведено на рисунку 4.4.

Рисунок 4.4 – Результат виконання тест-кейсу №2

На рисунку 4.5 наведено зміст файлів стану та інвентаря, що використовувалися для тестування.

```
[["Check module functionality"]]
hosts = "server1"
scripts = [
    "script1"
]
```

```
[machines.server1]
address = '127.0.0.1'
username = 'Test'
password = 'Adm123'

[machines.server2]
address = '192.168.10.19'
username = 'Admin'
password = 'Password2'
variables = {test_var_1 = true, test_var_2 = 16, test_var_3 = 12.8, test_var_4 = 'string data'}

[groups.group1]
members = ['server1', 'server2']
[groups.group1.variables]
test_group_var_another = 45.5

[groups.group2]
members = ['server2']
[groups.group2.variables]
test_group_var_1 = 25
test_group_var_2 = false
```

Рисунок 4.5 – Зміст файлів стану (зверху) та інвентаря (знизу)

Лістинг конфігураційного скрипту, що використовувався для тестування, наведено на рисунку 4.6.

```
    util.Logger.Info( msg: "Script 1 is started!")

    groupVar := util.Vars["test_group_var_another"]
    util.Logger.Info("Variable value: " + strconv.FormatFloat(groupVar.(float64),  fmt: 'E',  prec: -1,  bitSize: 64))

    stdout, _, _, err := shell.Execute( script: "[System.Net.Dns]::GetHostName()")
    if err ≠ nil {
        util.Logger.Error( msg: "Failed to run shell script")
        util.Logger.Error(err.Error())
    } else {
        util.Logger.Info(stdout)
    }

    info, err := file.GetStat( filename: "c:\\windows\\system32\\drivers\\etc\\hosts")
    if err ≠ nil {
        util.Logger.Error( msg: "Failed to get file info")
        util.Logger.Error(err.Error())
    } else {
        util.Logger.Info(strconv.FormatInt(info.Size,  base: 10))
    }
```

Рисунок 4.6 – Лістинг конфігураційного скрипту для тестування

Отже, було проведено тестування програмного забезпечення для управління конфігураціями. Для перевірки працездатності реалізації алгоритмів та інших функціональних компонентів ПЗ було розроблено 18 модульних тестів, тестування не виявило проблем. Для перевірки CLI та GUI було розроблено два тест-кейси та відповідні TOML-файли з даними для тестування, перевірка функціоналу не виявила проблем.

4.3 Розробка інструкції користувача

Для використання «Detrint» необхідно два виконавчих файли: програма-агент, що буде виконуватись на цільових серверах, а також програма-контролер, яка керує виконанням конфігураційних скриптів. Програма-агент запускається у вигляді консольного додатку без яких-небудь додаткових параметрів та відразу починає відображати логи виконання. Спочатку відбувається реєстрація gRPC-сервісів, після чого агент закінчує ініціацію і поточний сервер готовий для використання в якості цільового хоста «Detrint». Приклад запущеної програми-агента показано на рисунку 4.7.

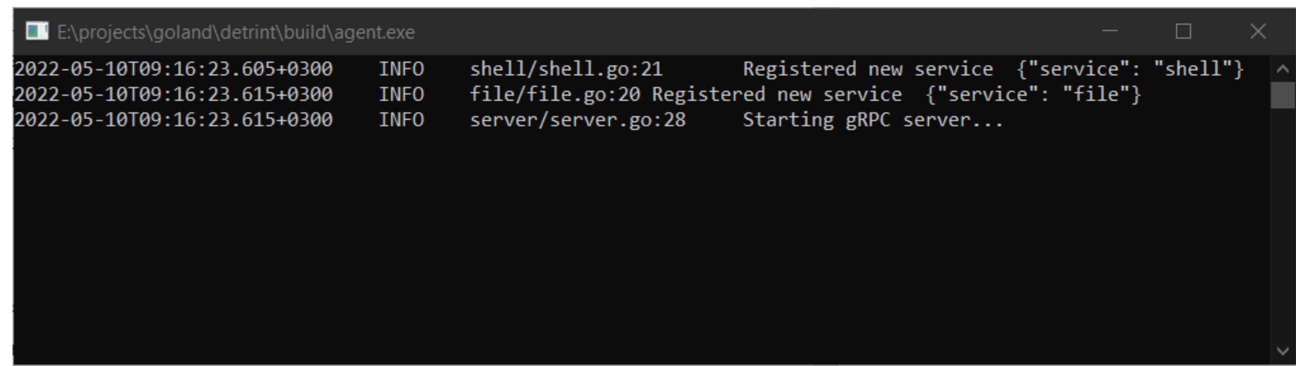


Рисунок 4.7 – Логи готової до роботи програма-агент

Програма-контролер доступна в двох варіантах: CLI та GUI. Додаток «Detrint-CLI» представлений у вигляді швидкого й простого в використанні інструменту для командного рядка, що може бути легко інтегрований в існуючу систему автоматизації або CI/CD-пайплайни. При запуску програми без додаткових параметрів буде виведено коротку інформацію про «Detrint», а

також надано список доступних для користувача команд. Приклад запуску без параметрів наведено на рисунку 4.8.

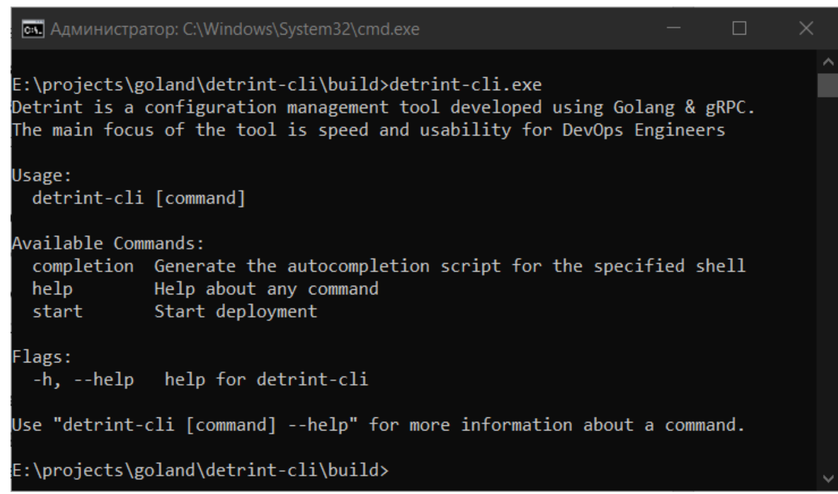


Рисунок 4.8 – Приклад запуску «Detrint-CLI» без додаткових параметрів

Для запуску виконання конфігураційних скриптів необхідно скористатися командою «start», яка має два обов’язкових аргументи: шлях до файлу стану та шлях до файлу інвентаря. Правильний формат команди має наступний вигляд: «detrint-cli.exe start -i <шлях до файлу інвентаря> -s <шлях до файлу стану>».

Варіант програми-контролера з графічним інтерфейсом має назву «Detrint-UI», а також надає користувачеві доступ до ряду додаткових функцій. Після запуску виконавчого файлу користувач отримає доступ до інтерфейсу з трьома вкладками, між якими можна вільно переключатися у будь-який момент часу. Перша вкладка «Інвентар» відповідає за завантаження файлу інвентаря та відображення його вмісту. Після успішного зчитування вмісту TOML-файлу вкладка автоматично оновить інтерфейс і в лівій частині вікна буде доступний список усіх серверів та груп. При натисканні на сервер або групу буде відображено всі доступні відомості про обраний об’єкт. Помилки при завантаженні або зчитуванні файлу, такі як неправильний синтаксис опису інвентаря або неуспішний виклик API, будуть відображатися у вигляді спливаючих оповіщень усередині вікна програми. Приклад інтерфейсу вкладки «Інвентар» з завантаженим файлом наведено на рисунку 4.9.

Рисунок 4.9 – Приклад інтерфейсу вкладки «Інвентар»

Після роботи з інвентарем необхідно перейти на наступну вкладку «Стан» та завантажити файл стану. Аналогічно до керування інвентарем користувач може переглянути список станів та вивчити додаткову інформацію про будь-який з них, що наведено на рисунку 4.10.

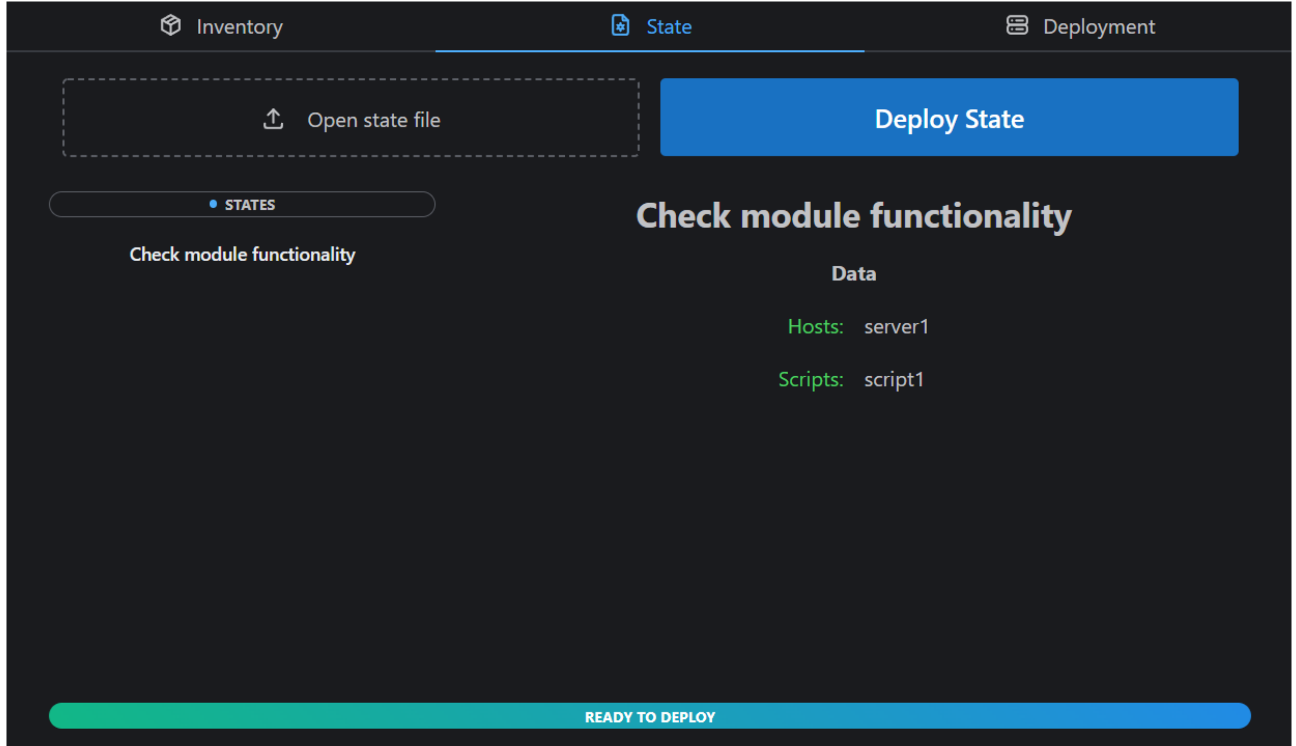


Рисунок 4.10 – Приклад інтерфейсу вкладки «Стан»

Якщо файл стану має коректний синтаксис та був успішно завантажений і прочитаний, то індикатор внизу вікна буде відображати відповідний текст про готовність початку приведення системи в бажаний стан. Кнопка в правій верхній частині вкладки дозволяє запустити виконання конфігураційних скриптів. На останній вкладці «Виконання» можливо переглянути всі доступні логи та поточний статус. Приклад інтерфейсу наведено на рисунку 4.11.

Inventory State Deployment

DEPLOYMENT IS COMPLETED 1/1


```
1 {"level":"info","ts":1652166305.4550092,"msg":"Starting state deployment","state":"Check module functionality"}
2 {"level":"info","ts":1652166305.4550092,"msg":"Resolved host","hosts":{"server1":{"address":"127.0.0.1","username":"Test","pa
3 {"level":"info","ts":1652166305.4587233,"msg":"Scripts to execute","scripts":["script1"]}
4 {"level":"info","ts":1652166305.4587233,"msg":"Executing script","script":"script1"}
5 {"level":"info","ts":1652166305.4646978,"msg":"Script 1 is started!"}
6 {"level":"info","ts":1652166305.4646978,"msg":"Variable value: 4.55E+01"}
7 {"level":"info","ts":1652166305.7782,"msg":"DESKTOP-RC0NMC0\r\n"}
8 {"level":"info","ts":1652166305.779247,"msg":"1325"}
```


Рисунок 4.11 – Приклад інтерфейсу вкладки «Виконання»

Файли стану та інвентаря описуються у форматі TOML – простому текстовому форматі для конфігураційних файлів з легким у використанні синтаксисом [33]. Приклад синтаксису для опису бажаного стану системи наведено на рисунку 4.12.

```
1  [["Install Web Server"]] # Ім'я стану, довільний текстовий рядок
2  hosts = "server3" # Ім'я цільового серверу
3  scripts = [ # Список назв скриптів для виконання
4      "web_server",
5      "site_files"
6  ]
7
8  [["Install Filse Server"]]
9  hosts = "fs_group" # Можна вказати відразу групу серверів
10 scripts = [
11     "file_server"
12 ]
```

Рисунок 4.12 – Приклад синтаксису файлу стану з коментарями

Приклад синтаксису файлу інвентаря наведено на рисунку 4.13.

```
1  [machines.server1] # Заголовок опису сервера в форматі "machines.<назва>", без спец. символів
2  address = '127.0.0.1' # Обов'язковий параметр, може бути IPv4 або IPv6, порт опціональний
3  username = 'Test' # Опціональне поле, що може бути використане для додаткового логіну в систему
4  password = 'Adm123'# Опціональне поле, що може бути використане для додаткового логіну в систему
5
6  [machines.server2]
7  address = '192.168.10.19'
8  username = 'Admin'
9  password = 'Password2'
10 variables = {test_var_1 = true, test_var_2 = 16, test_var_3 = 12.8, test_var_4 = 'string data'}
11 # Поле з масивом змінних є опціональним
12
13 [groups.group1] # Заголовок групи в форматі "groups.<назва>", без спец. символів
14 members = ['server1', 'server2'] # Список серверів та/або груп, що входять до складу
15 [groups.group1.variables] # Масив змінних може мати інший синтаксис, відповідно до формату TOML
16 test_group_var_another = 45.5
```

Рисунок 4.13 – Приклад синтаксису файлу інвентаря з коментарями

Скрипти для управління конфігураціями пишуться з використанням мови Golang. За замовчуванням їх пошук відбувається в директорії «scripts» всередині робочої папки програми-контролера.

Отже, було розроблено інструкцію користувача. Описано використання програми-агента, розглянуто приклади роботи з програмами-контролерами «Detrint-CLI» та «Detrint-UI». Для CLI-варіанту додатку описано доступні команди та обов’язкові параметри, наведено приклад формату команди для запуску конфігураційних скриптів. Для GUI-варіанту додатку наведено опис інтерфейсу й порядок роботи з ним, розглянуто всі реалізовані функції. Наведено приклади синтаксису для TOML-файлів інвентаря та стану.

4.4 Системні вимоги

Мова програмування Go підтримує крос-компіляцію [34] та не обмежує розробника в виборі цільової платформи. Крім того, всі використані бібліотеки є крос-платформними, тому додаток «Detrint» доступний для десктопних ОС Windows, macOS та різноманітних дистрибутивів на базі ядра Linux. Додаток розповсюджується у вигляді виконавчих файлів для відповідних платформ.

Підтримується робота на пристроях з архітектурою x86-64 та ARM64. Мінімальні та рекомендовані системні вимоги для пристроїв наведено в таблиці 4.1.

Таблиця 4.1 – Системні вимоги для «Detrint»

Параметр	Мінімальна конфігурація	Рекомендована конфігурація
CPU	Процесор на архітектурі x86-64 або ARM64 з тактовою частотою 1,6 ГГц або більше	Процесор на архітектурі x86-64 або ARM64 з тактовою частотою 2,4 ГГц або більше
RAM	512 МБ для дистрибутивів Linux та 4 ГБ для ОС Windows або MacOS	2 ГБ для дистрибутивів Linux та 8 ГБ для ОС Windows або MacOS
GPU	Інтегрований графічний пристрій (тільки для «Detrint-UI»)	Інтегрований графічний пристрій (тільки для «Detrint-UI»)
Накопичувач	HDD, 128 МБ вільного місця	SSD, 512 МБ вільного місця
ОС	Windows 7 або дистрибутив з ядром Linux 2.6.23 або macOS High Sierra 10.13	Windows 10 або дистрибутив з ядром Linux 5.17.6 або macOS Monterey 12

Отже, в даному підрозділі було розглянуто системні вимоги для додатку «Detrint». Описано мінімальну та рекомендовану конфігурацію для роботи з програмним забезпеченням.

4.5 Висновки

У даному розділі було розглянуто базові відомості про тестування програмного забезпечення. Для перевірки працездатності було обрано компонентне тестування та тестування «чорного ящика». Розглянуто створені тест-кейси та модульні тести. За результатом тестуванням проблем не виявлено, робота функцій додатку відповідає очікуванням. Було розроблено докладну інструкцію користувача, розглянуто використання усіх складових компонентів програмного забезпечення, наведено приклади файлів інвентаря та стану. Системні вимоги для роботи з ПЗ оформлено у вигляді таблиці.

ВИСНОВКИ

У бакалаврській дипломній роботі було розроблено програмний додаток для управління конфігураціями при розгорненні та масштабуванні електронних ресурсів під назвою «Detrint». Розроблене ПЗ призначено для підвищення ефективності та гнучкості процесу управління конфігураціями при розгорненні та масштабуванні електронних ресурсів шляхом удосконалення алгоритмів формування змінних хостів і запуску конфігураційних скриптів.

Було проведено аналіз сфери керування електронними ресурсами. Розглянуто існуючі аналоги ПЗ для управління конфігураціями, проаналізовано їх переваги та недоліки, доведено доцільність розробки власного програмного продукту. Досліджено існуючі підходи до вирішення задачі. З використанням отриманих даних виконано постановку задач розробки програмного продукту.

Було розроблено архітектуру створюваного ПЗ, розроблено схему компонентів і їх взаємодії, створено UML-діаграму діяльності. Розроблено алгоритм формування змінних хостів, що дає можливість виконувати більш складні конфігурації. Розроблено алгоритм запуску конфігураційних скриптів, який дає можливість використовувати інтерпретовані скрипти та модулі для більш гнучкого управління конфігураціями. Було розроблено структурні схеми інтерфейсу додатку, детально описано окремі елементи GUI.

Було проведено варіантний аналіз та обґрунтування вибору мови програмування, досліджено переваги й недоліки популярних IDE. Для розробки додатку обрано мову програмування Go та IDE JetBrains GoLand. Детально описано процес розробки основних модулів програмного забезпечення.

Було розглянуто основні види тестування ПЗ, обрано компонентне тестування та тестування «чорного ящика» для перевірки працездатності додатку. Розроблено модульні тести та тест-кейси для ручного тестування, за результатами тестування проблем не виявлено. Було розроблено інструкцію користувача та сформовані системні вимоги для обладнання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Martineau K. What a little more computing power can do. Massachusetts Institute of Technology. URL: https://news.mit.edu/2019/what-extra-computing-power-can-do-0916.

2. Belmont J.-M. Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI. Birmingham : Packt Publishing, 2018. 416 p.

3. The cost of IT downtime. The 20. URL: https://www.the20.com/blog/the-cost-of-it-downtime/.

4. Infrastructure as code. IBM Cloud Learn Hub. URL: https://www.ibm.com/cloud/learn/infrastructure-as-code.

5. Yurt T. Good and bad parts of ansible after 2 years of usage. URL: https://mtyurt.net/post/2020/good-bad-parts-of-ansible-after-two-years.html.

6. Миргородський А. В., Романюк О. В. Використання засобів для керування конфігураціями при розгорненні та масштабуванні електронних ресурсів. Міжнародна науково-практична інтернет-конференція «Електронні інформаційні ресурси: створення, використання, доступ», м. Вінниця, 9-10 листоп. 2021 р. Вінниця, 2021. URL: https://ir.lib.vntu.edu.ua/handle/123456789/34864?show=full.

7. Миргородський А. В., Романюк О. В. Використання фреймворку gRPC для розробки гнучких клієнт-серверних додатків. LI Науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії, м. Вінниця, 31 трав. 2022 р. Вінниця, 2022. URL: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2022/paper/view/14918.

8. Миргородський А. В., Романюк О. В. Розробка алгоритму запуску скриптів при управлінні конфігураціями. XXII Всеукраїнська науково-технічна конференція молодих вчених, аспірантів та студентів «Стан, досягнення та перспективи інформаційних систем і технологій», м. Одеса, 21–22 квіт. 2022 р.

Одеса, 2022. URL: https://www.onaft.edu.ua/download/konfi/2022/Conference_abstract-IT-21-22-04-22.pdf.

9. Morris K. Infrastructure as code: dynamic systems for the cloud age. 2nd ed. Sebastopol : O'Reilly Media, 2021. 430 p.

10. Horizontal scaling vs. vertical scaling: how to choose which is right for you. MongoDB. URL: https://www.mongodb.com/basics/horizontal-vs-vertical-scaling.

11. Geerling J. Ansible for DevOps: server and configuration management for humans. St. Louis : Midwestern Mac, 2020. 478 p.

12. Taylor M., Vargo S. Learning Chef: a guide to configuration management and automation. Sebastopol : O'Reilly Media, 2013. 366 p.

13. Uphill T. Mastering Puppet. Birmingham : Packt Publishing, 2014. 280 p.

14. Myers C. Learning SaltStack: build, manage, and secure your infrastructure with the power of saltstack. 2nd ed. Birmingham : Packt Publishing, 2016. 202 p.

15. How to build your inventory. Ansible Documentation. URL: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html.

16. Silva J. What you need to know about Ansible modules. URL: https://opensource.com/article/19/3/developing-ansible-modules.

17. Spaleta J. Implementing infrastructure as code with Ansible. Sensu. URL: https://sensu.io/blog/implementing-infrastructure-as-code-with-ansible.

18. Hochstein L., Moser R. Ansible: up and running, automating configuration management and deployment the easy way. 2nd ed. Sebastopol : O'Reilly Media, 2017. 430 p.

19. Using variables. Ansible Documentation. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html.

20. Heidi E. How to define tasks in Ansible playbooks. DigitalOcean. URL: https://www.digitalocean.com/community/tutorials/how-to-define-tasks-in-ansible-playbooks.

21. Pedamkar P. What is GUI?. EDUCBA. URL: https://www.educba.com/what-is-gui/.

22. Stroustrup B. Programming: principles and practice using C++. 2nd ed. Boston : Addison-Wesley Professional, 2014. 1312 p.

23. Skeet J. C# in depth. 4th ed. Shelter Island : Manning, 2019. 528 p.

24. Lubanovic B. Introducing Python: modern computing in simple packages. 2nd ed. Sebastopol : O'Reilly Media, 2019. 630 p.

25. Tsoukalos M. Mastering Go: Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures. Birmingham : Packt Publishing, 2019. 798 p.

26. LiteIDE X. GitHub. URL: https://github.com/visualfc/liteide.

27. Configure Visual Studio Code for Go development. Microsoft Docs. URL: https://docs.microsoft.com/en-us/azure/developer/go/configure-visual-studio-code.

28. Roberts E. GoLand vs Visual Studio Code: which one is the best IDE for Go. URL: https://ethr.me/goland-vs-visual-studio-code-which-one-is-the-best-ide-for-go-5f2760e4006b.

29. Chadokar S. How to create a CLI in golang with cobra. URL: https://towardsdatascience.com/how-to-create-a-cli-in-golang-with-cobra-d729641c7177.

30. Ammann P., Offutt J. Introduction to software testing. 2nd ed. Cambridge : Cambridge University Press, 2016. 364 p.

31. Siddiqui S. Learning test-driven development: a polyglot guide to writing uncluttered code. Sebastopol : O'Reilly Media, 2021. 280 p.

32. Beizer B. Black-box testing: techniques for functional testing of software and systems. Hoboken : Wiley, 1995. 320 p.

33. Preston-Werner T., Gedam P. TOML Tom's Obvious, Minimal Language. URL: https://toml.io/en/v1.0.0.

34. Claerhout P. Cross compiling Go apps. YellowDuck. URL: https://www.yellowduck.be/posts/cross-compile.

ДОДАТКИ

Додаток А. Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедрою ПЗ
д.т.н., проф.
____________ О. Н. Романюк
"31" ___березня___ 2022 р.

Технічне завдання
на бакалаврську дипломну роботу «Розробка програмного забезпечення для управління конфігураціями при розгорненні та масштабуванні електронних ресурсів» за спеціальністю 121 – Інженерія програмного забезпечення

Керівник бакалаврської дипломної роботи:
____________ к.т.н., доцент О.В. Романюк
"31" ___березня___ 2022 р.

Виконав:
____________ студент гр. 3ПІ-18б А.В. Миргородський
"31" ___березня___ 2022 р.

Вінниця – 2022 року

1. Найменування та галузь застосування

Бакалаврська дипломна робота: «Розробка програмного забезпечення для управління конфігураціями при розгорненні та масштабуванні електронних ресурсів».

Галузь застосування – автоматизація процесів управління конфігураціями, розгорнення і масштабування електронних ресурсів.

2. Підстава для розробки.

Підставою для виконання бакалаврської дипломної роботи (БДР) є індивідуальне завдання на БДР та наказ №66 від 24 березня 2022 р. ректора по ВНТУ про закріплення тем БДР.

3. Мета та призначення розробки.

Метою бакалаврської роботи є підвищення ефективності та гнучкості процесу управління конфігураціями при розгорненні та масштабуванні електронних ресурсів шляхом удосконалення алгоритмів формування змінних хостів і запуску конфігураційних скриптів.

Призначення роботи – розробка та програмна реалізація додатку для управління конфігураціями при розгорненні та масштабуванні електронних ресурсів.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БДР.

1. Morris K. Infrastructure as code: dynamic systems for the cloud age. 2nd ed. Sebastopol : O'Reilly Media, 2021. 430 p.

2. Tsoukalos M. Mastering Go: Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures. Birmingham : Packt Publishing, 2019. 798 p.

3. Ammann P., Offutt J. Introduction to software testing. 2nd ed. Cambridge : Cambridge University Press, 2016. 364 p.

5. Технічні вимоги

Модель розробки – ітеративна; метод передачі повідомлень між серверами – виклик віддалених процедур (RPC); вхідні дані – текстові файли в форматі TOML з описом бажаного стану системи та налаштуваннями для підключення і роботи з серверами; вихідні дані – логи виконання інструкцій на віддалених серверах; середовище розробки – JetBrains GoLand; мова програмування – Go.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

1. Пояснювальна записка до БДР;
2. Технічне завдання;
3. Лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання і вибір методу вирішення поставленої задачі дослідження	26.03.2022 – 02.04.2022	Вик.
2	Розробка архітектури програмного додатку	03.04.2022 – 09.04.2022	Вик.
3	Розробка алгоритму формування змінних хостів	10.04.2022 – 17.04.2022	Вик.
4	Розробка алгоритму запуску конфігураційних скриптів	18.04.2022 – 28.04.2022	Вик.
5	Аналіз і вибір мови програмування та середовища розробки	29.04.2022 – 05.05.2022	Вик.
6	Програмна реалізація додатку	06.05.2022 – 20.05.2022	Вик.
7	Тестування програмного забезпечення	21.05.2022 – 25.05.2022	Вик.
8	Оформлення матеріалів до захисту БДР	26.05.2022 – 10.06.2022	Вик.

10. Порядок контролю та прийняття

Виконання етапів бакалаврської дипломної роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття бакалаврської дипломної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

Додаток Б. Протокол перевірки кваліфікаційної роботи

на наявність текстових запозичень

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: «Розробка програмного забезпечення для управління конфігураціями при розгорненні та масштабуванні електронних ресурсів»

Тип роботи: БДР

Підрозділ : кафедра програмного забезпечення, ФІТКІ

Науковий керівник: Романюк О. В.

Оригінальність	**97,4%**
Схожість	2,6%

Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку ____________ Черноволик Г. О.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck

Автор роботи ____________ Миргородський А. В.

Керівник роботи ____________ Романюк О. В.

Додаток В. Лістинг програми

agent\server\server.go

```
package server

import (
   "github.com/ghotfall/detrint/agent/service/file"
   "github.com/ghotfall/detrint/agent/service/shell"
   "go.uber.org/zap"
   "google.golang.org/grpc"
   "net"
)

var DefaultPort = "7056"
var DefaultHost = "0.0.0.0"

func Start(host, port string, l *zap.Logger) {
   address := net.JoinHostPort(host, port)
   listener, listenerErr := net.Listen("tcp", address)
   if listenerErr != nil {
      l.Error("Failed to create listener", zap.String("address", address),
zap.Error(listenerErr))
      return
   }

   server := grpc.NewServer()

   // Register services
   shell.Register(server, l)
   file.Register(server, l)

   l.Info("Starting gRPC server...")
   servErr := server.Serve(listener)
   if servErr != nil {
      l.Error("Failed to start gRPC server", zap.Error(servErr))
      return
   }
}
```

agent\service\file\file.go

```
package file

import (
   "context"
   "github.com/ghotfall/detrint/grpc/filepb"
   "go.uber.org/zap"
   "google.golang.org/grpc"
   "google.golang.org/grpc/codes"
   "google.golang.org/grpc/status"
   "os"
   "time"
)

var logger *zap.Logger

func Register(s grpc.ServiceRegistrar, l *zap.Logger) {
   logger = l

   filepb.RegisterFileServiceServer(s, Server{})
   logger.Info("Registered new service", zap.String("service", "file"))
```

```
}

type Server struct {
    filepb.UnimplementedFileServiceServer
}

func (s Server) GetStat(_ context.Context, request *filepb.StatRequest)
(*filepb.StatResponse, error) {
    logger.Info("Method file.getstat is called", zap.String("request",
request.String()))

    filename := request.GetFilename()
    resp := filepb.StatResponse{}

    fileInfo, err := os.Stat(filename)
    if err != nil {
        pathErr, ok := err.(*os.PathError)
        if ok {
            resp.PathErr = pathErr.Error()
        } else {
            return nil, status.Errorf(codes.Internal, "failed to get file stat:
%s", err.Error())
        }
    } else {
        resp.Name = fileInfo.Name()
        resp.Size = fileInfo.Size()
        resp.Mode = uint32(fileInfo.Mode())
        resp.Time = fileInfo.ModTime().Format(time.RFC3339)
        resp.Dir = fileInfo.IsDir()
    }

    logger.Debug("Execution of shell.execute finished", zap.Any("result", &resp))
    return &resp, nil
}
```

agent\service\shell\shell.go

```
package shell

import (
    "bytes"
    "context"
    "github.com/ghotfall/detrint/grpc/shellpb"
    "go.uber.org/zap"
    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "os/exec"
    "strings"
)

var logger *zap.Logger

func Register(s grpc.ServiceRegistrar, l *zap.Logger) {
    logger = l

    shellpb.RegisterShellServiceServer(s, Server{})
    logger.Info("Registered new service", zap.String("service", "shell"))
}

type Server struct {
    shellpb.UnimplementedShellServiceServer
}
```

```
func (s Server) Execute(_ context.Context, request *shellpb.ExecuteRequest)
(*shellpb.ExecuteResponse, error) {
   logger.Info("Method shell.execute is called", zap.String("request",
request.String()))

   var stdout bytes.Buffer
   var stderr bytes.Buffer

   resp := shellpb.ExecuteResponse{}

   cmd := exec.Command("powershell", "-NoProfile", request.GetScript())
   cmd.Stdout = &stdout
   cmd.Stderr = &stderr

   err := cmd.Run()
   resp.Stdout = strings.ToValidUTF8(stdout.String(), "")
   resp.Stderr = strings.ToValidUTF8(stderr.String(), "")

   if err != nil {
      exitError, ok := err.(*exec.ExitError)
      if ok {
         resp.Code = int32(exitError.ExitCode())
      } else {
         logger.Error("Unexpected error while executing shell.execute",
zap.Error(err))
         return nil, status.Errorf(codes.Internal, "failed to run script: %s",
err.Error())
      }
   } else {
      resp.Code = 0
   }

   logger.Debug(
      "Execution of shell.execute finished",
      zap.String("stdout", resp.Stdout),
      zap.String("stderr", resp.Stderr),
      zap.Int("code", int(resp.Code)),
   )

   return &resp, nil
}
```

builtin\builtin.go

```
package builtin

import (
   "github.com/ghotfall/detrint/builtin/file"
   "github.com/ghotfall/detrint/builtin/shell"
   "github.com/ghotfall/detrint/builtin/util"
   "github.com/traefik/yaegi/interp"
   "github.com/traefik/yaegi/stdlib"
   "go.uber.org/zap"
   "google.golang.org/grpc"
   "reflect"
)

type Settings struct {
   Logger     *zap.Logger
   Vars       map[string]interface{}
   Connection *grpc.ClientConn
}

func Load(interpreter *interp.Interpreter, settings Settings) error {
```

```
	stdlibErr := interpreter.Use(stdlib.Symbols)
	if stdlibErr != nil {
		return stdlibErr
	}

	symbols := make(map[string]map[string]reflect.Value)

	symbols["github.com/ghotfall/detrint/builtin/util/util"] =
util.Symbols(settings.Logger, settings.Vars)
	symbols["github.com/ghotfall/detrint/builtin/shell/shell"] =
shell.Symbols(settings.Connection)
	symbols["github.com/ghotfall/detrint/builtin/file/file"] =
file.Symbols(settings.Connection)

	symbolsErr := interpreter.Use(symbols)
	if symbolsErr != nil {
		return symbolsErr
	}

	return nil
}
```

builtin\file\file.go

```
package file

import (
	"context"
	"github.com/ghotfall/detrint/grpc/filepb"
	"google.golang.org/grpc"
	"reflect"
)

var client filepb.FileServiceClient

func Symbols(conn *grpc.ClientConn) map[string]reflect.Value {
	client = filepb.NewFileServiceClient(conn)

	return map[string]reflect.Value{
		"GetStat": reflect.ValueOf(GetStat),
		"Info":    reflect.ValueOf((*Info)(nil)),
	}
}

type Info struct {
	Name    string
	Size    int64
	Mode    uint32
	Time    string
	Dir     bool
	PathErr string
}

func GetStat(filename string) (Info, error) {
	stat, err := client.GetStat(context.Background(),
&filepb.StatRequest{Filename: filename})
	if err != nil {
		return Info{}, err
	} else {
		return Info{
			Name:    stat.GetName(),
			Size:    stat.GetSize(),
			Mode:    stat.GetMode(),
			Time:    stat.GetTime(),
```

```
            Dir:     stat.GetDir(),
            PathErr: stat.GetPathErr(),
        }, nil
    }
}
```

builtin\shell\shell.go

```
package shell

import (
    "context"
    "github.com/ghotfall/detrint/grpc/shellpb"
    "google.golang.org/grpc"
    "reflect"
)

var client shellpb.ShellServiceClient

func Symbols(conn *grpc.ClientConn) map[string]reflect.Value {
    client = shellpb.NewShellServiceClient(conn)

    return map[string]reflect.Value{
        "Execute": reflect.ValueOf(Execute),
    }
}

func Execute(script string) (stdout, stderr string, code int, err error) {
    response, err := client.Execute(context.Background(),
&shellpb.ExecuteRequest{Script: script})
    if err != nil {
        return "", "", 0, err
    } else {
        return response.GetStdout(), response.GetStderr(),
int(response.GetCode()), nil
    }
}
```

builtin\util\util.go

```
package util

import (
    "go.uber.org/zap"
    "reflect"
)

var Logger *zap.Logger
var Vars map[string]interface{}

func Symbols(l *zap.Logger, v map[string]interface{}) map[string]reflect.Value {
    Logger = l
    Vars = v

    return map[string]reflect.Value{
        "Logger": reflect.ValueOf(Logger), // TODO: wrapper for logger?
        "Vars":   reflect.ValueOf(Vars),
    }
}
```

grpc\filepb\file.proto

```
syntax = "proto3";
```

```
package file;
option go_package="./;filepb";

message StatRequest {
  string filename = 1;
}

message StatResponse {
  string name = 1;
  int64 size = 2;
  uint32 mode = 3;
  string time = 4;
  bool dir = 5;
  string pathErr = 6;
}

service FileService {
  rpc GetStat(StatRequest) returns (StatResponse) {};
}
```

grpc\filepb\file.pb.go

```
package filepb

import (
   protoreflect "google.golang.org/protobuf/reflect/protoreflect"
   protoimpl "google.golang.org/protobuf/runtime/protoimpl"
   reflect "reflect"
   sync "sync"
)

const (
   _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
   _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)

type StatRequest struct {
   state         protoimpl.MessageState
   sizeCache     protoimpl.SizeCache
   unknownFields protoimpl.UnknownFields

   Filename string `protobuf:"bytes,1,opt,name=filename,proto3"
json:"filename,omitempty"`
}

func (x *StatRequest) Reset() {
   *x = StatRequest{}
   if protoimpl.UnsafeEnabled {
      mi := &file_grpc_filepb_file_proto_msgTypes[0]
      ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
      ms.StoreMessageInfo(mi)
   }
}

func (x *StatRequest) String() string {
   return protoimpl.X.MessageStringOf(x)
}

func (*StatRequest) ProtoMessage() {}

func (x *StatRequest) ProtoReflect() protoreflect.Message {
   mi := &file_grpc_filepb_file_proto_msgTypes[0]
   if protoimpl.UnsafeEnabled && x != nil {
      ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
```

```
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}

func (*StatRequest) Descriptor() ([]byte, []int) {
    return file_grpc_filepb_file_proto_rawDescGZIP(), []int{0}
}

func (x *StatRequest) GetFilename() string {
    if x != nil {
        return x.Filename
    }
    return ""
}

type StatResponse struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Name    string `protobuf:"bytes,1,opt,name=name,proto3"
json:"name,omitempty"`
    Size    int64  `protobuf:"varint,2,opt,name=size,proto3"
json:"size,omitempty"`
    Mode    uint32 `protobuf:"varint,3,opt,name=mode,proto3"
json:"mode,omitempty"`
    Time    string `protobuf:"bytes,4,opt,name=time,proto3"
json:"time,omitempty"`
    Dir     bool   `protobuf:"varint,5,opt,name=dir,proto3" json:"dir,omitempty"`
    PathErr string `protobuf:"bytes,6,opt,name=pathErr,proto3"
json:"pathErr,omitempty"`
}

func (x *StatResponse) Reset() {
    *x = StatResponse{}
    if protoimpl.UnsafeEnabled {
        mi := &file_grpc_filepb_file_proto_msgTypes[1]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *StatResponse) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*StatResponse) ProtoMessage() {}

func (x *StatResponse) ProtoReflect() protoreflect.Message {
    mi := &file_grpc_filepb_file_proto_msgTypes[1]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}
```

```
func (*StatResponse) Descriptor() ([]byte, []int) {
   return file_grpc_filepb_file_proto_rawDescGZIP(), []int{1}
}

func (x *StatResponse) GetName() string {
   if x != nil {
      return x.Name
   }
   return ""
}

func (x *StatResponse) GetSize() int64 {
   if x != nil {
      return x.Size
   }
   return 0
}

func (x *StatResponse) GetMode() uint32 {
   if x != nil {
      return x.Mode
   }
   return 0
}

func (x *StatResponse) GetTime() string {
   if x != nil {
      return x.Time
   }
   return ""
}

func (x *StatResponse) GetDir() bool {
   if x != nil {
      return x.Dir
   }
   return false
}

func (x *StatResponse) GetPathErr() string {
   if x != nil {
      return x.PathErr
   }
   return ""
}

var File_grpc_filepb_file_proto protoreflect.FileDescriptor

var file_grpc_filepb_file_proto_rawDesc = []byte{
   0x0a, 0x16, 0x67, 0x72, 0x70, 0x63, 0x2f, 0x66, 0x69, 0x6c, 0x65, 0x70, 0x62,
0x2f, 0x66, 0x69,
   0x6c, 0x65, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x12, 0x04, 0x66, 0x69, 0x6c,
0x65, 0x22, 0x29,
   0x0a, 0x0b, 0x53, 0x74, 0x61, 0x74, 0x52, 0x65, 0x71, 0x75, 0x65, 0x73, 0x74,
0x12, 0x1a, 0x0a,
   0x08, 0x66, 0x69, 0x6c, 0x65, 0x6e, 0x61, 0x6d, 0x65, 0x18, 0x01, 0x20, 0x01,
0x28, 0x09, 0x52,
   0x08, 0x66, 0x69, 0x6c, 0x65, 0x6e, 0x61, 0x6d, 0x65, 0x22, 0x8a, 0x01, 0x0a,
0x0c, 0x53, 0x74,
   0x61, 0x74, 0x52, 0x65, 0x73, 0x70, 0x6f, 0x6e, 0x73, 0x65, 0x12, 0x12, 0x0a,
0x04, 0x6e, 0x61,
   0x6d, 0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d,
0x65, 0x12, 0x12,
   0x0a, 0x04, 0x73, 0x69, 0x7a, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x03, 0x52,
```

```
0x04, 0x73, 0x69,
   0x7a, 0x65, 0x12, 0x12, 0x0a, 0x04, 0x6d, 0x6f, 0x64, 0x65, 0x18, 0x03, 0x20,
0x01, 0x28, 0x0d,
   0x52, 0x04, 0x6d, 0x6f, 0x64, 0x65, 0x12, 0x12, 0x0a, 0x04, 0x74, 0x69, 0x6d,
0x65, 0x18, 0x04,
   0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x74, 0x69, 0x6d, 0x65, 0x12, 0x10, 0x0a,
0x03, 0x64, 0x69,
   0x72, 0x18, 0x05, 0x20, 0x01, 0x28, 0x08, 0x52, 0x03, 0x64, 0x69, 0x72, 0x12,
0x18, 0x0a, 0x07,
   0x70, 0x61, 0x74, 0x68, 0x45, 0x72, 0x72, 0x18, 0x06, 0x20, 0x01, 0x28, 0x09,
0x52, 0x07, 0x70,
   0x61, 0x74, 0x68, 0x45, 0x72, 0x72, 0x32, 0x41, 0x0a, 0x0b, 0x46, 0x69, 0x6c,
0x65, 0x53, 0x65,
   0x72, 0x76, 0x69, 0x63, 0x65, 0x12, 0x32, 0x0a, 0x07, 0x47, 0x65, 0x74, 0x53,
0x74, 0x61, 0x74,
   0x12, 0x11, 0x2e, 0x66, 0x69, 0x6c, 0x65, 0x2e, 0x53, 0x74, 0x61, 0x74, 0x52,
0x65, 0x71, 0x75,
   0x65, 0x73, 0x74, 0x1a, 0x12, 0x2e, 0x66, 0x69, 0x6c, 0x65, 0x2e, 0x53, 0x74,
0x61, 0x74, 0x52,
   0x65, 0x73, 0x70, 0x6f, 0x6e, 0x73, 0x65, 0x22, 0x00, 0x42, 0x0b, 0x5a, 0x09,
0x2e, 0x2f, 0x3b,
   0x66, 0x69, 0x6c, 0x65, 0x70, 0x62, 0x62, 0x06, 0x70, 0x72, 0x6f, 0x74, 0x6f,
0x33,
}

var (
   file_grpc_filepb_file_proto_rawDescOnce sync.Once
   file_grpc_filepb_file_proto_rawDescData = file_grpc_filepb_file_proto_rawDesc
)

func file_grpc_filepb_file_proto_rawDescGZIP() []byte {
   file_grpc_filepb_file_proto_rawDescOnce.Do(func() {
      file_grpc_filepb_file_proto_rawDescData =
protoimpl.X.CompressGZIP(file_grpc_filepb_file_proto_rawDescData)
   })
   return file_grpc_filepb_file_proto_rawDescData
}

var file_grpc_filepb_file_proto_msgTypes = make([]protoimpl.MessageInfo, 2)
var file_grpc_filepb_file_proto_goTypes = []interface{}{
   (*StatRequest)(nil),  // 0: file.StatRequest
   (*StatResponse)(nil), // 1: file.StatResponse
}
var file_grpc_filepb_file_proto_depIdxs = []int32{
   0, // 0: file.FileService.GetStat:input_type -> file.StatRequest
   1, // 1: file.FileService.GetStat:output_type -> file.StatResponse
   1, // [1:2] is the sub-list for method output_type
   0, // [0:1] is the sub-list for method input_type
   0, // [0:0] is the sub-list for extension type_name
   0, // [0:0] is the sub-list for extension extendee
   0, // [0:0] is the sub-list for field type_name
}

func init() { file_grpc_filepb_file_proto_init() }
func file_grpc_filepb_file_proto_init() {
   if File_grpc_filepb_file_proto != nil {
      return
   }
   if !protoimpl.UnsafeEnabled {
      file_grpc_filepb_file_proto_msgTypes[0].Exporter = func(v interface{}, i
int) interface{} {
         switch v := v.(*StatRequest); i {
         case 0:
            return &v.state
```

```
			case 1:
				return &v.sizeCache
			case 2:
				return &v.unknownFields
			default:
				return nil
			}
		}
		file_grpc_filepb_file_proto_msgTypes[1].Exporter = func(v interface{}, i 
int) interface{} {
			switch v := v.(*StatResponse); i {
			case 0:
				return &v.state
			case 1:
				return &v.sizeCache
			case 2:
				return &v.unknownFields
			default:
				return nil
			}
		}
	}
	type x struct{}
	out := protoimpl.TypeBuilder{
		File: protoimpl.DescBuilder{
			GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
			RawDescriptor: file_grpc_filepb_file_proto_rawDesc,
			NumEnums:      0,
			NumMessages:   2,
			NumExtensions: 0,
			NumServices:   1,
		},
		GoTypes:           file_grpc_filepb_file_proto_goTypes,
		DependencyIndexes: file_grpc_filepb_file_proto_depIdxs,
		MessageInfos:      file_grpc_filepb_file_proto_msgTypes,
	}.Build()
	File_grpc_filepb_file_proto = out.File
	file_grpc_filepb_file_proto_rawDesc = nil
	file_grpc_filepb_file_proto_goTypes = nil
	file_grpc_filepb_file_proto_depIdxs = nil
}
```

grpc\filepb\file_grpc.pb.go

```
package filepb

import (
	context "context"
	grpc "google.golang.org/grpc"
	codes "google.golang.org/grpc/codes"
	status "google.golang.org/grpc/status"
)

const _ = grpc.SupportPackageIsVersion7

type FileServiceClient interface {
	GetStat(ctx context.Context, in *StatRequest, opts ...grpc.CallOption) 
(*StatResponse, error)
}

type fileServiceClient struct {
	cc grpc.ClientConnInterface
}
```

```
func NewFileServiceClient(cc grpc.ClientConnInterface) FileServiceClient {
   return &fileServiceClient{cc}
}

func (c *fileServiceClient) GetStat(ctx context.Context, in *StatRequest, opts
...grpc.CallOption) (*StatResponse, error) {
   out := new(StatResponse)
   err := c.cc.Invoke(ctx, "/file.FileService/GetStat", in, out, opts...)
   if err != nil {
      return nil, err
   }
   return out, nil
}

type FileServiceServer interface {
   GetStat(context.Context, *StatRequest) (*StatResponse, error)
   mustEmbedUnimplementedFileServiceServer()
}

type UnimplementedFileServiceServer struct {
}

func (UnimplementedFileServiceServer) GetStat(context.Context, *StatRequest)
(*StatResponse, error) {
   return nil, status.Errorf(codes.Unimplemented, "method GetStat not
implemented")
}
func (UnimplementedFileServiceServer) mustEmbedUnimplementedFileServiceServer()
{}

type UnsafeFileServiceServer interface {
   mustEmbedUnimplementedFileServiceServer()
}

func RegisterFileServiceServer(s grpc.ServiceRegistrar, srv FileServiceServer) {
   s.RegisterService(&FileService_ServiceDesc, srv)
}

func _FileService_GetStat_Handler(srv interface{}, ctx context.Context, dec
func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{},
error) {
   in := new(StatRequest)
   if err := dec(in); err != nil {
      return nil, err
   }
   if interceptor == nil {
      return srv.(FileServiceServer).GetStat(ctx, in)
   }
   info := &grpc.UnaryServerInfo{
      Server:     srv,
      FullMethod: "/file.FileService/GetStat",
   }
   handler := func(ctx context.Context, req interface{}) (interface{}, error) {
      return srv.(FileServiceServer).GetStat(ctx, req.(*StatRequest))
   }
   return interceptor(ctx, in, info, handler)
}

var FileService_ServiceDesc = grpc.ServiceDesc{
   ServiceName: "file.FileService",
   HandlerType: (*FileServiceServer)(nil),
   Methods: []grpc.MethodDesc{
      {
         MethodName: "GetStat",
```

```
            Handler:    _FileService_GetStat_Handler,
        },
    },
    Streams:  []grpc.StreamDesc{},
    Metadata: "grpc/filepb/file.proto",
}
```

grpc\shellpb\shell.proto

```
syntax = "proto3";

package shell;
option go_package="./;shellpb";

message ExecuteRequest {
  string script = 1;
}

message ExecuteResponse {
  string stdout = 1;
  string stderr = 2;
  int32 code = 3;
}

service ShellService {
  rpc Execute(ExecuteRequest) returns (ExecuteResponse) {};
}
```

grpc\shellpb\shell.pb.go

```
package shellpb

import (
   protoreflect "google.golang.org/protobuf/reflect/protoreflect"
   protoimpl "google.golang.org/protobuf/runtime/protoimpl"
   reflect "reflect"
   sync "sync"
)

const (
   _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
   _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)

type ExecuteRequest struct {
   state         protoimpl.MessageState
   sizeCache     protoimpl.SizeCache
   unknownFields protoimpl.UnknownFields

   Script string `protobuf:"bytes,1,opt,name=script,proto3"
json:"script,omitempty"`
}

func (x *ExecuteRequest) Reset() {
   *x = ExecuteRequest{}
   if protoimpl.UnsafeEnabled {
      mi := &file_grpc_shellpb_shell_proto_msgTypes[0]
      ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
      ms.StoreMessageInfo(mi)
   }
}

func (x *ExecuteRequest) String() string {
   return protoimpl.X.MessageStringOf(x)
```

```
}

func (*ExecuteRequest) ProtoMessage() {}

func (x *ExecuteRequest) ProtoReflect() protoreflect.Message {
   mi := &file_grpc_shellpb_shell_proto_msgTypes[0]
   if protoimpl.UnsafeEnabled && x != nil {
      ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
      if ms.LoadMessageInfo() == nil {
         ms.StoreMessageInfo(mi)
      }
      return ms
   }
   return mi.MessageOf(x)
}

func (*ExecuteRequest) Descriptor() ([]byte, []int) {
   return file_grpc_shellpb_shell_proto_rawDescGZIP(), []int{0}
}

func (x *ExecuteRequest) GetScript() string {
   if x != nil {
      return x.Script
   }
   return ""
}

type ExecuteResponse struct {
   state         protoimpl.MessageState
   sizeCache     protoimpl.SizeCache
   unknownFields protoimpl.UnknownFields

   Stdout string `protobuf:"bytes,1,opt,name=stdout,proto3"
json:"stdout,omitempty"`
   Stderr string `protobuf:"bytes,2,opt,name=stderr,proto3"
json:"stderr,omitempty"`
   Code   int32  `protobuf:"varint,3,opt,name=code,proto3"
json:"code,omitempty"`
}

func (x *ExecuteResponse) Reset() {
   *x = ExecuteResponse{}
   if protoimpl.UnsafeEnabled {
      mi := &file_grpc_shellpb_shell_proto_msgTypes[1]
      ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
      ms.StoreMessageInfo(mi)
   }
}

func (x *ExecuteResponse) String() string {
   return protoimpl.X.MessageStringOf(x)
}

func (*ExecuteResponse) ProtoMessage() {}

func (x *ExecuteResponse) ProtoReflect() protoreflect.Message {
   mi := &file_grpc_shellpb_shell_proto_msgTypes[1]
   if protoimpl.UnsafeEnabled && x != nil {
      ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
      if ms.LoadMessageInfo() == nil {
         ms.StoreMessageInfo(mi)
      }
      return ms
   }
```

```
    return mi.MessageOf(x)
}

func (*ExecuteResponse) Descriptor() ([]byte, []int) {
    return file_grpc_shellpb_shell_proto_rawDescGZIP(), []int{1}
}

func (x *ExecuteResponse) GetStdout() string {
    if x != nil {
        return x.Stdout
    }
    return ""
}

func (x *ExecuteResponse) GetStderr() string {
    if x != nil {
        return x.Stderr
    }
    return ""
}

func (x *ExecuteResponse) GetCode() int32 {
    if x != nil {
        return x.Code
    }
    return 0
}

var File_grpc_shellpb_shell_proto protoreflect.FileDescriptor

var file_grpc_shellpb_shell_proto_rawDesc = []byte{
    0x0a, 0x18, 0x67, 0x72, 0x70, 0x63, 0x2f, 0x73, 0x68, 0x65, 0x6c, 0x6c, 0x70,
0x62, 0x2f, 0x73,
    0x68, 0x65, 0x6c, 0x6c, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x12, 0x05, 0x73,
0x68, 0x65, 0x6c,
    0x6c, 0x22, 0x28, 0x0a, 0x0e, 0x45, 0x78, 0x65, 0x63, 0x75, 0x74, 0x65, 0x52,
0x65, 0x71, 0x75,
    0x65, 0x73, 0x74, 0x12, 0x16, 0x0a, 0x06, 0x73, 0x63, 0x72, 0x69, 0x70, 0x74,
0x18, 0x01, 0x20,
    0x01, 0x28, 0x09, 0x52, 0x06, 0x73, 0x63, 0x72, 0x69, 0x70, 0x74, 0x22, 0x55,
0x0a, 0x0f, 0x45,
    0x78, 0x65, 0x63, 0x75, 0x74, 0x65, 0x52, 0x65, 0x73, 0x70, 0x6f, 0x6e, 0x73,
0x65, 0x12, 0x16,
    0x0a, 0x06, 0x73, 0x74, 0x64, 0x6f, 0x75, 0x74, 0x18, 0x01, 0x20, 0x01, 0x28,
0x09, 0x52, 0x06,
    0x73, 0x74, 0x64, 0x6f, 0x75, 0x74, 0x12, 0x16, 0x0a, 0x06, 0x73, 0x74, 0x64,
0x65, 0x72, 0x72,
    0x18, 0x02, 0x20, 0x01, 0x28, 0x09, 0x52, 0x06, 0x73, 0x74, 0x64, 0x65, 0x72,
0x72, 0x12, 0x12,
    0x0a, 0x04, 0x63, 0x6f, 0x64, 0x65, 0x18, 0x03, 0x20, 0x01, 0x28, 0x05, 0x52,
0x04, 0x63, 0x6f,
    0x64, 0x65, 0x32, 0x4a, 0x0a, 0x0c, 0x53, 0x68, 0x65, 0x6c, 0x6c, 0x53, 0x65,
0x72, 0x76, 0x69,
    0x63, 0x65, 0x12, 0x3a, 0x0a, 0x07, 0x45, 0x78, 0x65, 0x63, 0x75, 0x74, 0x65,
0x12, 0x15, 0x2e,
    0x73, 0x68, 0x65, 0x6c, 0x6c, 0x2e, 0x45, 0x78, 0x65, 0x63, 0x75, 0x74, 0x65,
0x52, 0x65, 0x71,
    0x75, 0x65, 0x73, 0x74, 0x1a, 0x16, 0x2e, 0x73, 0x68, 0x65, 0x6c, 0x6c, 0x2e,
0x45, 0x78, 0x65,
    0x63, 0x75, 0x74, 0x65, 0x52, 0x65, 0x73, 0x70, 0x6f, 0x6e, 0x73, 0x65, 0x22,
0x00, 0x42, 0x0c,
    0x5a, 0x0a, 0x2e, 0x2f, 0x3b, 0x73, 0x68, 0x65, 0x6c, 0x6c, 0x70, 0x62, 0x62,
0x06, 0x70, 0x72,
    0x6f, 0x74, 0x6f, 0x33,
```

```
}

var (
	file_grpc_shellpb_shell_proto_rawDescOnce sync.Once
	file_grpc_shellpb_shell_proto_rawDescData =
file_grpc_shellpb_shell_proto_rawDesc
)

func file_grpc_shellpb_shell_proto_rawDescGZIP() []byte {
	file_grpc_shellpb_shell_proto_rawDescOnce.Do(func() {
		file_grpc_shellpb_shell_proto_rawDescData =
protoimpl.X.CompressGZIP(file_grpc_shellpb_shell_proto_rawDescData)
	})
	return file_grpc_shellpb_shell_proto_rawDescData
}

var file_grpc_shellpb_shell_proto_msgTypes = make([]protoimpl.MessageInfo, 2)
var file_grpc_shellpb_shell_proto_goTypes = []interface{}{
	(*ExecuteRequest)(nil),  // 0: shell.ExecuteRequest
	(*ExecuteResponse)(nil), // 1: shell.ExecuteResponse
}
var file_grpc_shellpb_shell_proto_depIdxs = []int32{
	0, // 0: shell.ShellService.Execute:input_type -> shell.ExecuteRequest
	1, // 1: shell.ShellService.Execute:output_type -> shell.ExecuteResponse
	1, // [1:2] is the sub-list for method output_type
	0, // [0:1] is the sub-list for method input_type
	0, // [0:0] is the sub-list for extension type_name
	0, // [0:0] is the sub-list for extension extendee
	0, // [0:0] is the sub-list for field type_name
}

func init() { file_grpc_shellpb_shell_proto_init() }
func file_grpc_shellpb_shell_proto_init() {
	if File_grpc_shellpb_shell_proto != nil {
		return
	}
	if !protoimpl.UnsafeEnabled {
		file_grpc_shellpb_shell_proto_msgTypes[0].Exporter = func(v interface{}, i
int) interface{} {
			switch v := v.(*ExecuteRequest); i {
			case 0:
				return &v.state
			case 1:
				return &v.sizeCache
			case 2:
				return &v.unknownFields
			default:
				return nil
			}
		}
		file_grpc_shellpb_shell_proto_msgTypes[1].Exporter = func(v interface{}, i
int) interface{} {
			switch v := v.(*ExecuteResponse); i {
			case 0:
				return &v.state
			case 1:
				return &v.sizeCache
			case 2:
				return &v.unknownFields
			default:
				return nil
			}
		}
	}
```

```
	type x struct{}
	out := protoimpl.TypeBuilder{
		File: protoimpl.DescBuilder{
			GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
			RawDescriptor: file_grpc_shellpb_shell_proto_rawDesc,
			NumEnums:      0,
			NumMessages:   2,
			NumExtensions: 0,
			NumServices:   1,
		},
		GoTypes:           file_grpc_shellpb_shell_proto_goTypes,
		DependencyIndexes: file_grpc_shellpb_shell_proto_depIdxs,
		MessageInfos:      file_grpc_shellpb_shell_proto_msgTypes,
	}.Build()
	File_grpc_shellpb_shell_proto = out.File
	file_grpc_shellpb_shell_proto_rawDesc = nil
	file_grpc_shellpb_shell_proto_goTypes = nil
	file_grpc_shellpb_shell_proto_depIdxs = nil
}
```

grpc\shellpb\shell_grpc.pb.go

```
package shellpb

import (
	context "context"
	grpc "google.golang.org/grpc"
	codes "google.golang.org/grpc/codes"
	status "google.golang.org/grpc/status"
)

const _ = grpc.SupportPackageIsVersion7

type ShellServiceClient interface {
	Execute(ctx context.Context, in *ExecuteRequest, opts ...grpc.CallOption)
(*ExecuteResponse, error)
}

type shellServiceClient struct {
	cc grpc.ClientConnInterface
}

func NewShellServiceClient(cc grpc.ClientConnInterface) ShellServiceClient {
	return &shellServiceClient{cc}
}

func (c *shellServiceClient) Execute(ctx context.Context, in *ExecuteRequest,
opts ...grpc.CallOption) (*ExecuteResponse, error) {
	out := new(ExecuteResponse)
	err := c.cc.Invoke(ctx, "/shell.ShellService/Execute", in, out, opts...)
	if err != nil {
		return nil, err
	}
	return out, nil
}

type ShellServiceServer interface {
	Execute(context.Context, *ExecuteRequest) (*ExecuteResponse, error)
	mustEmbedUnimplementedShellServiceServer()
}

type UnimplementedShellServiceServer struct {
}
```

```
func (UnimplementedShellServiceServer) Execute(context.Context, *ExecuteRequest)
(*ExecuteResponse, error) {
   return nil, status.Errorf(codes.Unimplemented, "method Execute not
implemented")
}
func (UnimplementedShellServiceServer)
mustEmbedUnimplementedShellServiceServer() {}

type UnsafeShellServiceServer interface {
   mustEmbedUnimplementedShellServiceServer()
}

func RegisterShellServiceServer(s grpc.ServiceRegistrar, srv ShellServiceServer)
{
   s.RegisterService(&ShellService_ServiceDesc, srv)
}

func _ShellService_Execute_Handler(srv interface{}, ctx context.Context, dec
func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{},
error) {
   in := new(ExecuteRequest)
   if err := dec(in); err != nil {
      return nil, err
   }
   if interceptor == nil {
      return srv.(ShellServiceServer).Execute(ctx, in)
   }
   info := &grpc.UnaryServerInfo{
      Server:     srv,
      FullMethod: "/shell.ShellService/Execute",
   }
   handler := func(ctx context.Context, req interface{}) (interface{}, error) {
      return srv.(ShellServiceServer).Execute(ctx, req.(*ExecuteRequest))
   }
   return interceptor(ctx, in, info, handler)
}

var ShellService_ServiceDesc = grpc.ServiceDesc{
   ServiceName: "shell.ShellService",
   HandlerType: (*ShellServiceServer)(nil),
   Methods: []grpc.MethodDesc{
      {
         MethodName: "Execute",
         Handler:    _ShellService_Execute_Handler,
      },
   },
   Streams:  []grpc.StreamDesc{},
   Metadata: "grpc/shellpb/shell.proto",
}
```

inv\group.go

```
package inv

type Group struct {
   Members   []string               `toml:"members" json:"members"`
   Variables map[string]interface{} `toml:"variables,omitempty"
json:"variables,omitempty"`
}
```

inv\inventory.go

```
package inv

import "fmt"

type Inventory struct {
   Machines map[string]Machine `toml:"machines" json:"machines"`
   Groups   map[string]Group   `toml:"groups" json:"groups"`
}

func (i Inventory) ResolveHosts(name string) map[string]Machine {
   if machine, ok := i.Machines[name]; ok {
      return map[string]Machine{name: machine}
   } else if group, ok := i.Groups[name]; ok {
      result := make(map[string]Machine)

      for _, member := range group.Members {
         resolved := i.ResolveHosts(member)
         for s, m := range resolved {
            result[s] = m
         }
      }

      return result
   } else {
      return map[string]Machine{}
   }
}

func (i Inventory) GetMachineVars(servername string) (map[string]interface{},
error) {
   result := make(map[string]interface{})

   machine, ok := i.Machines[servername]
   if !ok {
      // return empty map, not null
      return result, fmt.Errorf("can't find machine with name '%s'", servername)
   }

   for _, group := range i.Groups {
      for _, member := range group.Members {
         if member == servername {
            for k, v := range group.Variables {
               result[k] = v
            }
         }
      }
   }

   for k, v := range machine.Variables {
      result[k] = v
   }

   return result, nil
}
```

inv\inventory_test.go

```
package inv

import (
   "reflect"
```

```
    "testing"
)

func TestInventory_ResolveHosts(t *testing.T) {
    i := Inventory{
        Machines: map[string]Machine{
            "server_1": {Address: "10.0.0.1"},
            "server_2": {Address: "10.0.0.2"},
            "server_3": {Address: "10.0.0.3"},
            "server_4": {Address: "10.0.0.4"},
        },
        Groups: map[string]Group{
            "group_1": {Members: []string{"server_1", "server_2"}},
            "group_2": {Members: []string{"server_1", "group_1", "server_3"}},
            "group_3": {Members: []string{"group_2", "server_4"}},
        },
    }

    tests := []struct {
        name     string
        selector string
        want     map[string]Machine
    }{
        {name: "simple_server", selector: "server_2", want: map[string]Machine{
            "server_2": i.Machines["server_2"],
        }},
        {name: "simple_group", selector: "group_1", want: map[string]Machine{
            "server_1": i.Machines["server_1"],
            "server_2": i.Machines["server_2"],
        }},
        {name: "complex_group", selector: "group_2", want: map[string]Machine{
            "server_1": i.Machines["server_1"],
            "server_2": i.Machines["server_2"],
            "server_3": i.Machines["server_3"],
        }},
        {name: "complex_group_multiple", selector: "group_3", want:
map[string]Machine{
            "server_1": i.Machines["server_1"],
            "server_2": i.Machines["server_2"],
            "server_3": i.Machines["server_3"],
            "server_4": i.Machines["server_4"],
        }},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            if got := i.ResolveHosts(tt.selector); !reflect.DeepEqual(got, tt.want)
{
                t.Errorf("ResolveHosts() = %v, want %v", got, tt.want)
            }
        })
    }
}

func TestInventory_GetMachineVars(t *testing.T) {
    i := Inventory{
        Machines: map[string]Machine{
            "server_1": {Variables: map[string]interface{}{"var_1": true}},
            "server_2": {Variables: map[string]interface{}{"var_2": 123}},
            "server_3": {Variables: map[string]interface{}{"var_3": "no"}},
            "server_4": {Variables: map[string]interface{}{"var_4": 4.5, "var_41":
false}},
        },
        Groups: map[string]Group{
            "group_1": {
```

```
            Members: []string{"server_1", "server_2"},
            Variables: map[string]interface{}{
               "gr_var_1": true,
               "gr_var_2": "test",
            },
         },
      },
   }

   tests := []struct {
      name    string
      input   string
      want    map[string]interface{}
      wantErr bool
   }{
      {
         name:    "server_single_var",
         input:   "server_3",
         want:    map[string]interface{}{"var_3": "no"},
         wantErr: false,
      },
      {
         name:    "server_multiple_vars",
         input:   "server_4",
         want:    map[string]interface{}{"var_4": 4.5, "var_41": false},
         wantErr: false,
      },
      {
         name:    "server_group_1",
         input:   "server_1",
         want:    map[string]interface{}{"var_1": true, "gr_var_1": true,
"gr_var_2": "test"},
         wantErr: false,
      },
      {
         name:    "server_group_2",
         input:   "server_2",
         want:    map[string]interface{}{"var_2": 123, "gr_var_1": true,
"gr_var_2": "test"},
         wantErr: false,
      },
      {
         name:    "no_server",
         input:   "server_5",
         want:    map[string]interface{}{},
         wantErr: true,
      },
   }
   for _, tt := range tests {
      t.Run(tt.name, func(t *testing.T) {
         got, err := i.GetMachineVars(tt.input)
         if (err != nil) != tt.wantErr {
            t.Errorf("GetMachineVars() error = %v, wantErr %v", err, tt.wantErr)
            return
         }
         if !reflect.DeepEqual(got, tt.want) {
            t.Errorf("GetMachineVars() got = %v, want %v", got, tt.want)
         }
      })
   }
}
```

inv\machine.go

```
package inv

import (
   "fmt"
   "net"
   "net/url"
   "strings"
)

var DefaultPort = "7056"

type Machine struct {
   Address   string                  `toml:"address" json:"address"`
   Username  string                  `toml:"username" json:"username"`
   Password  string                  `toml:"password" json:"password"`
   Variables map[string]interface{} `toml:"variables,omitempty"
json:"variables,omitempty"`
}

func (m Machine) Target() (string, error) {
   if len(strings.TrimSpace(m.Address)) == 0 {
      return "", fmt.Errorf("address field is empty, can't make target")
   }

   u := url.URL{Host: m.Address}
   if len(u.Port()) == 0 {
      return net.JoinHostPort(u.Hostname(), DefaultPort), nil
   } else {
      return m.Address, nil
   }
}
```

inv\machine_test.go

```
package inv

import "testing"

func TestMachine_Target(t *testing.T) {
   m := Machine{}

   tests := []struct {
      name    string
      input   string
      want    string
      wantErr bool
   }{
      {name: "all_specified", input: "127.0.0.1:8080", want: "127.0.0.1:8080",
wantErr: false},
      {name: "without_port", input: "127.0.0.1", want: "127.0.0.1:" +
DefaultPort, wantErr: false},
      {name: "empty", input: "   ", want: "", wantErr: true},
      {name: "all_specified_ipv6", input: "[fe80::1%lo0]:53", want:
"[fe80::1%lo0]:53", wantErr: false},
      {name: "without_port_ipv6", input: "[fe80::1%lo0]", want: "[fe80::1%lo0]:"
+ DefaultPort, wantErr: false},
      {name: "with_colon_ipv6", input: "[fe80::1%lo0]:", want: "[fe80::1%lo0]:"
+ DefaultPort, wantErr: false},
   }

   for _, tt := range tests {
```

```
        t.Run(tt.name, func(t *testing.T) {
            m.Address = tt.input

            got, err := m.Target()
            if (err != nil) != tt.wantErr {
                t.Errorf("Target() error = %v, wantErr %v", err, tt.wantErr)
                return
            }
            if got != tt.want {
                t.Errorf("Target() got = %v, want %v", got, tt.want)
            }
        })
    }
}
```

state\base_structs.go

```
package state

import (
    "github.com/ghotfall/detrint/builtin"
    "github.com/ghotfall/detrint/inv"
    "github.com/traefik/yaegi/interp"
    "go.uber.org/zap"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
)

var ModulesPath = "./modules"
var ScriptsPath = "./scripts"

type State struct {
    Hosts   string   `toml:"hosts" json:"hosts"`
    Scripts []string `toml:"scripts" json:"scripts"`
}

type Set map[string]State

func (s Set) Deploy(i inv.Inventory, l *zap.Logger) {
    for stateName, state := range s {
        l.Info("Starting state deployment", zap.String("state", stateName))

        hosts := i.ResolveHosts(state.Hosts)
        l.Info("Resolved host", zap.Any("hosts", hosts))

        for servername, machine := range hosts {
            // Get correct target address
            target, targetErr := machine.Target()
            if targetErr != nil {
                l.Error(
                    "Can't make target string",
                    zap.String("address", machine.Address),
                    zap.Error(targetErr),
                )
                continue
            }

            // Create gRPC-connection for client creation
            conn, dialErr := grpc.Dial(target,
grpc.WithTransportCredentials(insecure.NewCredentials())) // TODO: TLS
            if dialErr != nil {
                l.Error(
```

```
            "Failed to create client connection, skipping target",
            zap.String("target", target),
            zap.Error(dialErr),
         )

         continue
      }

      // Create interpreter
      interpreter := interp.New(interp.Options{
         GoPath:       ModulesPath,
         Unrestricted: true,
      })

      // Get vars for current machine
      machineVars, varsErr := i.GetMachineVars(servername)
      if varsErr != nil {
         l.Error("Failed to get machine vars, skipping host",
zap.Error(varsErr))

         closeErr := conn.Close()

         if closeErr != nil {
            l.Error(
               "Failed to close client connection",
               zap.String("target", target),
               zap.Error(closeErr),
            )
         }

         continue
      }

      builtinErr := builtin.Load(interpreter,
         builtin.Settings{
            Logger:     l,
            Vars:       machineVars,
            Connection: conn,
         },
      )

      if builtinErr != nil {

         l.Error(
            "Failed to load builtin modules, skipping host",
            zap.Error(builtinErr),
         )

         closeErr := conn.Close()

         if closeErr != nil {
            l.Error(
               "Failed to close client connection",
               zap.String("target", target),
               zap.Error(closeErr),
            )
         }
         continue
      }

      // Execute scripts
      l.Info("Scripts to execute", zap.Strings("scripts", state.Scripts))
      for _, script := range state.Scripts {
         l.Info("Executing script", zap.String("script", script))
```

```
                _, evalErr := interpreter.EvalPath(ScriptsPath + "/" + script +
".go")

                if evalErr != nil {

                    l.Error(
                        "An failure occurred during script execution",
                        zap.String("script", script),
                        zap.Error(evalErr),
                    )
                }
            }

            // Close gRPC-connection
            closeErr := conn.Close()

            if closeErr != nil {
                l.Error(
                    "Failed to close client connection",
                    zap.String("target", target),
                    zap.Error(closeErr),
                )
            }
        }
    }
}
```

Додаток Г. Графічна частина

ГРАФІЧНА ЧАСТИНА

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ УПРАВЛІННЯ КОНФІГУРАЦІЯМИ ПРИ РОЗГОРНЕННІ ТА МАСШТАБУВАННІ ЕЛЕКТРОННИХ РЕСУРСІВ

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення

Бакалаврська дипломна робота на тему:
«РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ УПРАВЛІННЯ КОНФІГУРАЦІЯМИ ПРИ РОЗГОРНЕННІ ТА МАСШТАБУВАННІ ЕЛЕКТРОННИХ РЕСУРСІВ»

Автор: ст. гр. ЗПІ-18б Миргородський А.В.
Науковий керівник: к.т.н., доц. каф. ПЗ Романюк О.В.

Вінниця - 2022

Рисунок Г.1 – Титульний слайд

Актуальність теми

Для вирішення проблем управління цифровою інфраструктурою існує окремий прошарок програмного забезпечення, який спрощує й автоматизує такі задачі. До цієї категорії можна віднести різноманітні інструменти для IaC (Infrastructure as Code) або забезпечення для керування конфігураціями.

Функціональні та архітектурні недоліки існуючих додатків можуть стати значними обмеженнями та проблемами при певних сценаріях використання. Наприклад, відсутність GUI в базовій редакції ПЗ може стати проблемою для недосвідчених у роботі з CLI користувачів, обмежена підтримка ОС Windows може ускладнити розробку конфігурацій для певних пристроїв. Використання додаткових компонентів, як-от бібліотек для Python в продукті Ansible, збільшує час первинної підготовки систем до роботи з ПЗ для управління конфігураціями, а опис інструкцій у структурованому вигляді в файлах формату TOML або JSON може стати значним обмеженням за потреби комплексного налаштування систем.

Тому актуальною є задача підвищення ефективності управління конфігураціями шляхом розробки власного програмного продукту. Це передбачає розробку нових методів і алгоритмів роботи.

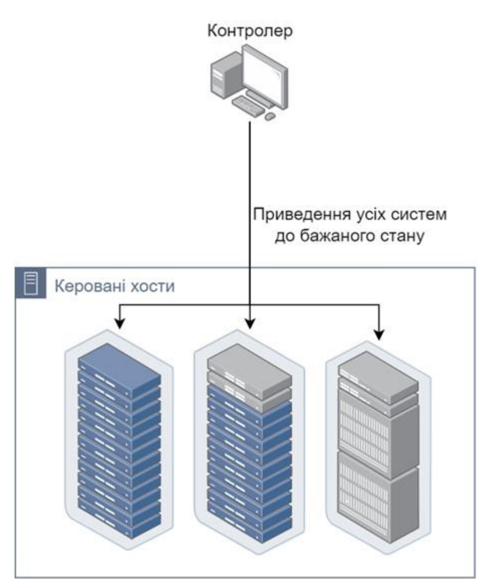


Рисунок Г.2 – Актуальність теми

Мета, об’єкт та предмет дослідження

- **Мета дослідження** – підвищення ефективності та гнучкості процесу управління конфігураціями при розгорненні та масштабуванні електронних ресурсів шляхом удосконалення алгоритмів формування змінних хостів і запуску конфігураційних скриптів.
- **Об’єкт дослідження** – процеси управління конфігураціями при розгорненні та масштабуванні електронних ресурсів.
- **Предмет дослідження** – методи та засоби розробки програмного забезпечення для управління конфігураціями.

Рисунок Г.3 – Мета, об’єкт та предмет дослідження

Задачі дослідження

Відповідно до поставленої мети в бакалаврській дипломній роботі потрібно вирішити такі завдання:

- проаналізувати стан технологій для управління конфігураціями;
- визначити високорівневу структуру та архітектуру програмного продукту;
- розробити алгоритм формування змінних хостів для їх використання при застосуванні конфігурацій;
- розробити алгоритм запуску конфігураційний скриптів;
- розробити графічний інтерфейс користувача для створюваного продукту;
- розробити програмний додаток для керування конфігураціями на основі створеної архітектури та алгоритмів;
- розробити модульні тести для автоматизованого тестування програмного додатку;
- провести ручне тестування програмного продукту з використанням комплексних сценаріїв виконання;
- розробити інструкцію користувача.

Рисунок Г.4 – Задачі дослідження

Новизна і практична цінність одержаних результатів

- Удосконалено алгоритм формування змінних хостів, у якому, на відміну від відомих алгоритмів, враховано комплексну логіку пріоритизації та перезапису даних, скореговано поведінку відповідно до зроблених архітектурних рішень та контексту виконання, що дозволило створювати та виконувати більш комплексні конфігурації систем.
- Удосконалено алгоритм запуску конфігураційних скриптів, який, на відміну від аналогічних алгоритмів, дозволяє використовувати інтерпретовані скрипти й модулі з можливістю динамічно змінювати контекст виконання, що дозволило підвищити гнучкість процесу керування конфігураціями.
- Практична цінність одержаних результатів полягає в тому, що на основі отриманих в бакалаврській дипломній роботі теоретичних положень запропоновано алгоритми та розроблено програмні засоби для підвищення ефективності та гнучкості процесу управління конфігураціями.

Рисунок Г.5 – Новизна і практична цінність одержаних результатів

Порівняльний аналіз аналогів

КРИТЕРІЙ	ANSIBLE	CHEF	PUPPET	SALTSTACK	DETRINT
CLI в базовій редакції	1	1	1	1	1
GUI в базовій редакції	0	0	0	0	1
Підтримка ОС Windows в якості контролера	0	0	0	0	1
Відсутність потреби встановлення агентів на керовані сервери	0,5	0	0	1	0
Підтримка конфігурацій з комплексною синхронізацією	0,5	0,5	0	0	1
Підсумок	2	1,5	1	2	4

Рисунок Г.6 – Порівняльний аналіз аналогів

Архітектура додатку

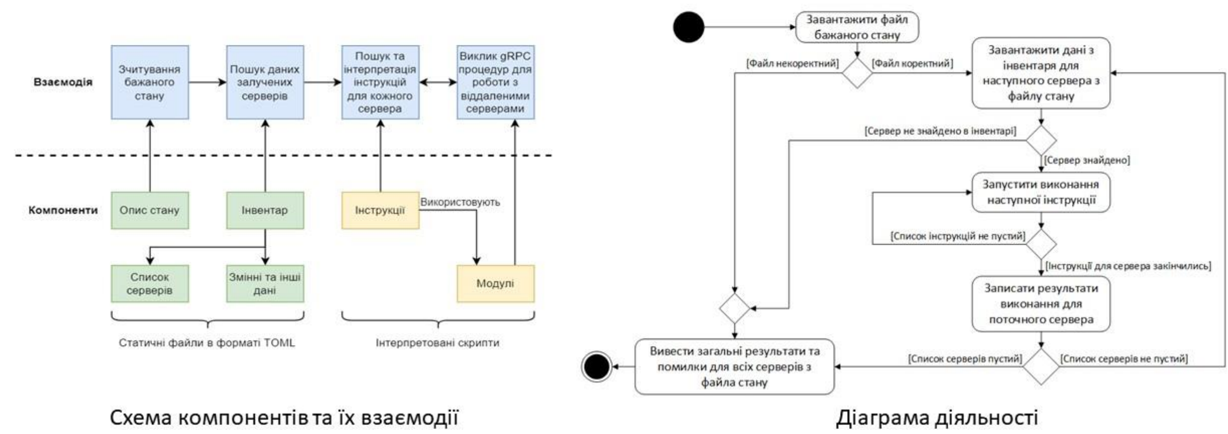


Рисунок Г.7 – Архітектура додатку

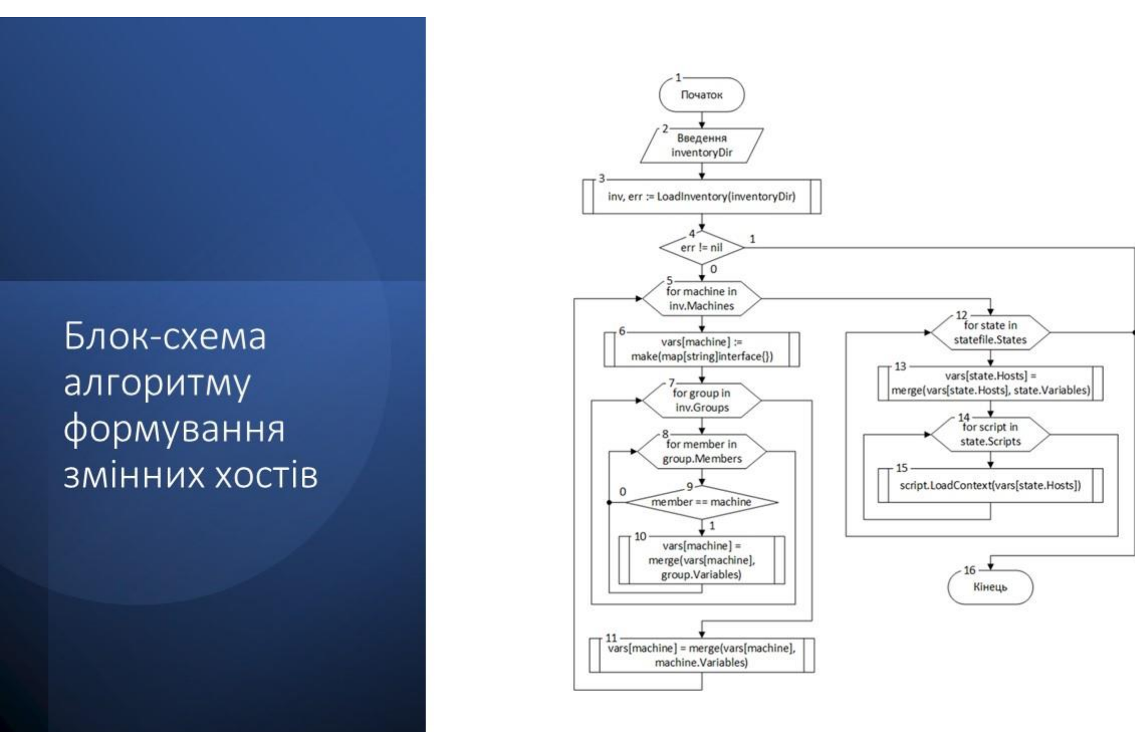
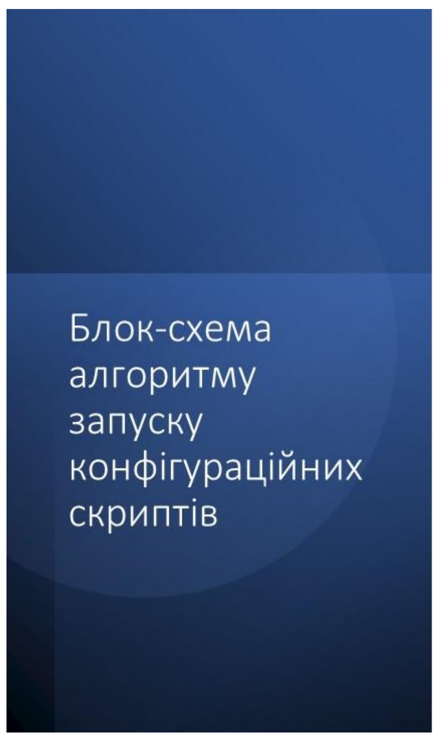


Рисунок Г.8 – Блок-схема алгоритму формування змінних хостів



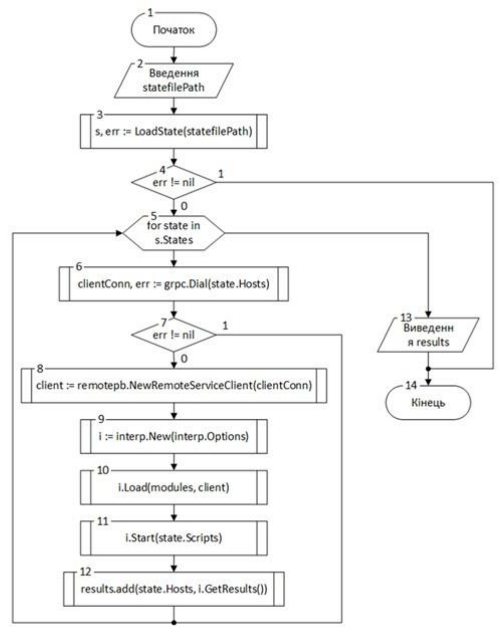


Рисунок Г.9 – Блок-схема алгоритму запуску конфігураційних скриптів

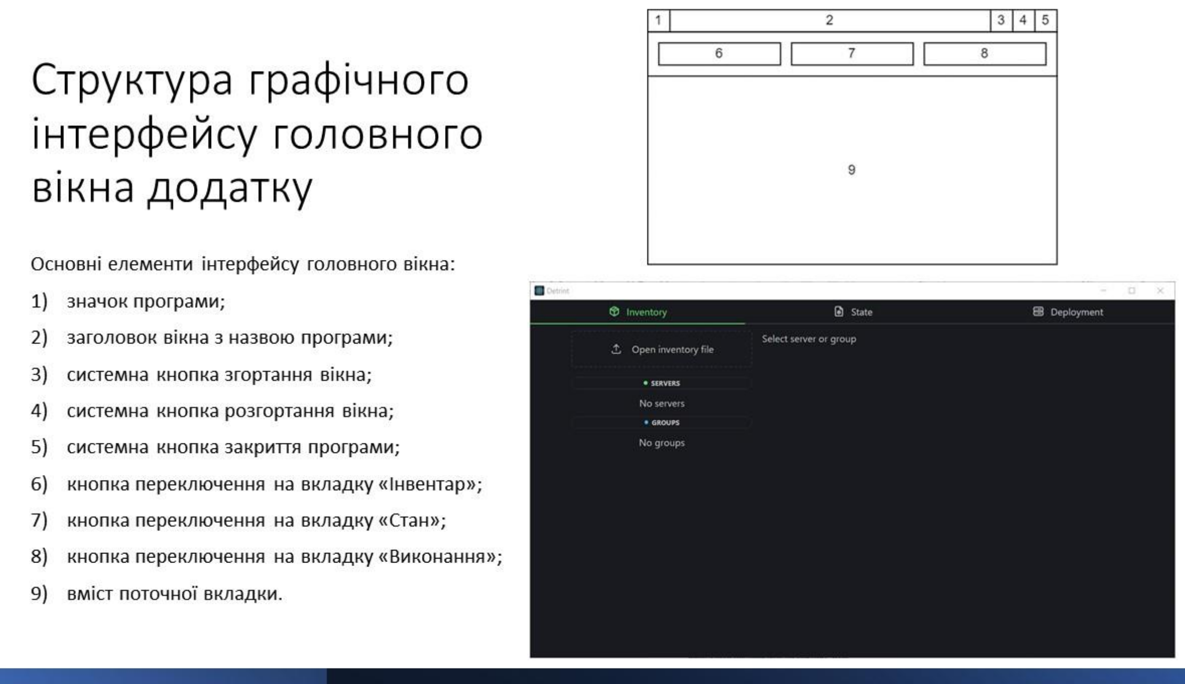
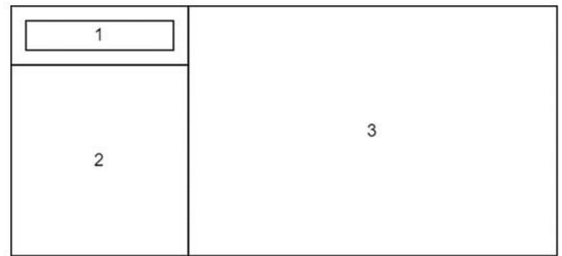


Рисунок Г.10 – Структура графічного інтерфейсу головного вікна додатку

Структура графічного інтерфейсу вкладки «Інвентар»

Вкладка «Інвентар» включає в себе лише три елементи інтерфейсу:

1) кнопка відкриття файлу інвентаря;
2) меню зі списком серверів та груп;
3) область з додатковою інформацією по обраному серверу або групі.



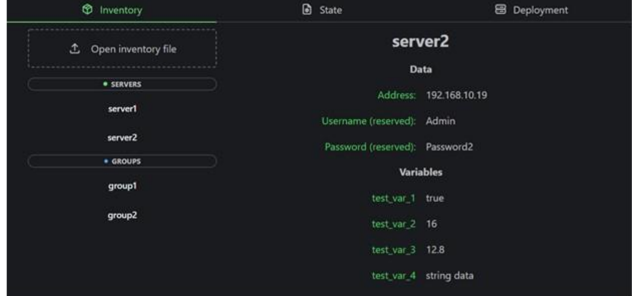
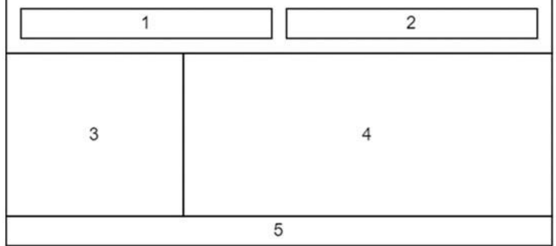


Рисунок Г.11 – Структура графічного інтерфейсу вкладки «Інвентар»

Структура графічного інтерфейсу вкладки «Стан»

Вкладка «Стан» включає в себе наступні елементи інтерфейсу:

1) кнопка відкриття файлу стану;
2) кнопка запуску виконання скриптів;
3) меню зі списком завантажених станів;
4) область з додатковою інформацією по обраному стану;
5) рядок статусу поточного файлу стану.



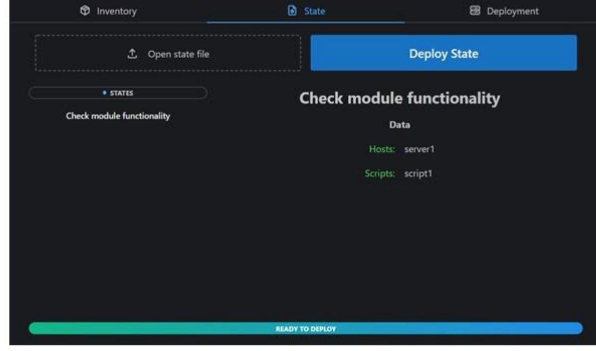
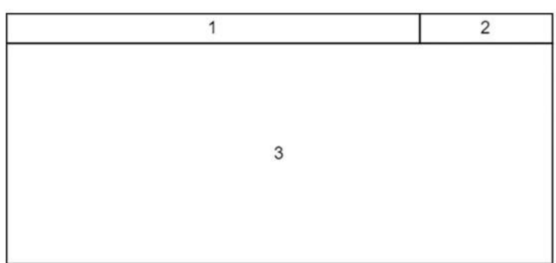


Рисунок Г.12 – Структура графічного інтерфейсу вкладки «Стан»

Структура графічного інтерфейсу вкладки «Виконання»

Елементи інтерфейсу, що знаходяться у цій вкладці, наступні:

1) текст-індикатор прогресу виконання;
2) загальний статус виконання;
3) логи скриптів.



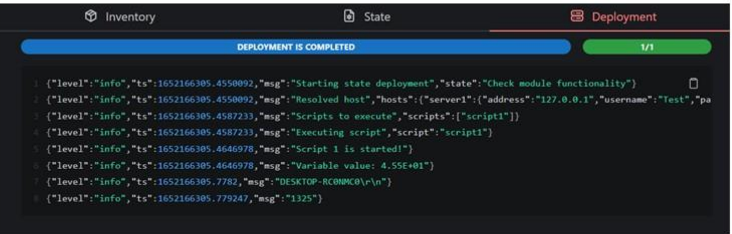
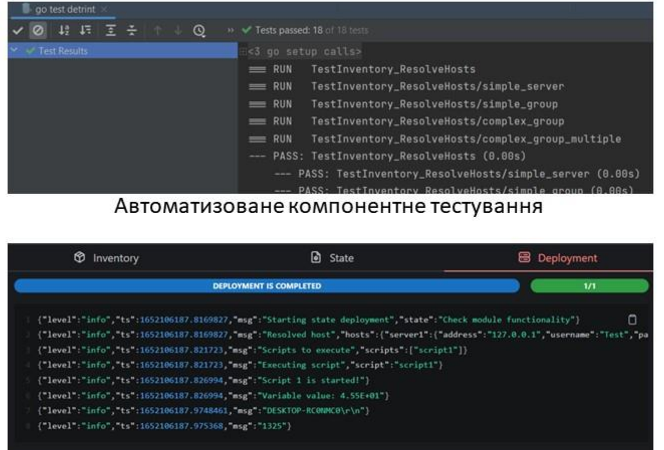
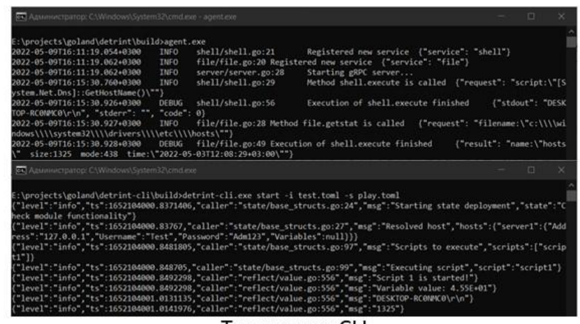


Рисунок Г.13 – Структура графічного інтерфейсу 3 вкладки «Виконання»



Автоматизоване компонентне тестування

Тестування GUI



Тестування CLI

Тестування програми

Рисунок Г.14 – Тестування програми

Результати роботи доповідалися на:

- Міжнародній науково-практичній інтернет-конференції «Електронні інформаційні ресурси: створення, використання, доступ» (2021 р., м. Вінниця);
- LI Науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії (2022 р., м. Вінниця);
- XXII Всеукраїнській науково-технічній конференції молодих вчених, аспірантів та студентів «Стан, досягнення та перспективи інформаційних систем і технологій» (2022 р., м. Одеса).

За тематикою дослідження опубліковано 3 наукових праці у збірниках матеріалів конференцій.

Рисунок Г.15 – Апробація матеріалів бакалаврської дипломної роботи

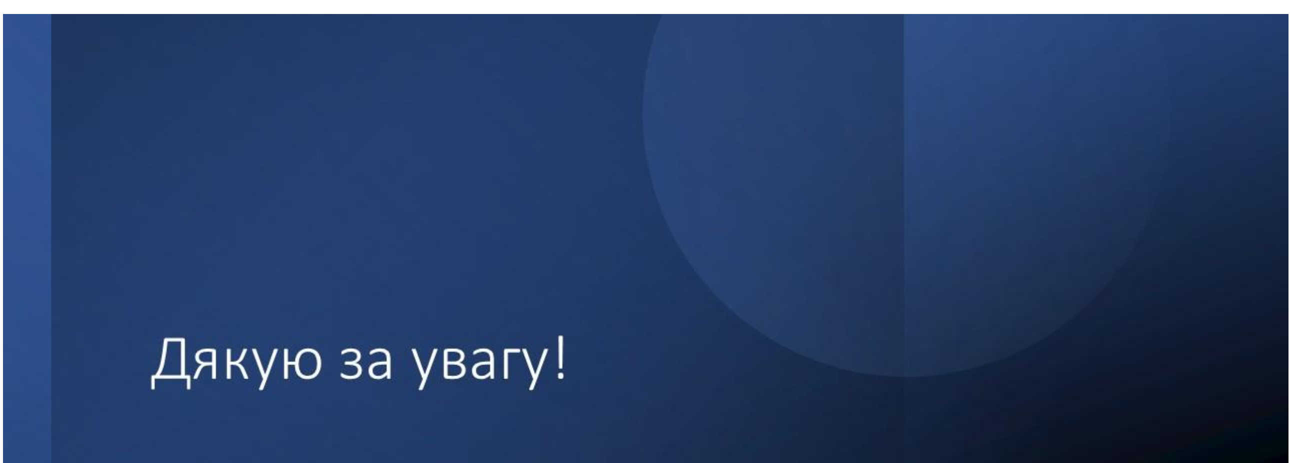


Рисунок Г.16 – Фінальний слайд