НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

«На правах рукопису»
УДК _____

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
«___»_____2023 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Фреймворк тестування навантаження веб-додатку»

Виконала:
студентка ІІ курсу, групи ІО-з21мп
Шпак Діана Михайлівна _____

Науковий керівник
доцент, к.т.н., с.н.с.
Долголенко О.М _____

Консультант:
проф., д.т.н., проф.,
Кулаков Юрій Олексійович _____

Рецензент:
доц. каф. РТС РТФ, к.т.н., доц.,
Катін П.В _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою

Спеціальність *123. Комп'ютерна інженерія*
(код і назва)

Спеціалізація *123. Комп'ютерні системи та мережі*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
«_____» _____ 2023 р.

ЗАВДАННЯ
на магістерську дисертацію студентці
Шпак Діані Михайлівні
(прізвище, ім'я, по батькові)

1. Тема дисертації Фреймворк тестування навантаження веб додатку

Науковий керівник дисертації доцент, к.т.н., с.н.с. Долголенко О.М
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «27» жовтня 2023 р. № 3587-с

2. Строк подання студентом дисертації 24.11.2023

3. Об'єкт дослідження процес тестування навантаження веб додатку

4. Предмет дослідження методи та засоби створення сценаріїв для побудови фреймфорку тестування навантаження веб додатку

5. Перелік завдань, які потрібно розробити: дослідити спосіб тестування навантаження веб додатку

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Кулаков Ю. О.		
1.	Куц В.Ю.		
2.	Куц В.Ю.		
3.	Куц В.Ю.		
4.	Куц В.Ю.		

7. Дата видачі завдання 1.09.2023

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1.	Затвердження теми дослідження. Визначення предмету дослідження	1.09.23-7.09.23	
2.	Дослідження існуючих рішень та проблем конструювання трафіка в досліджуваній області	8.09.23-21.09.23	
3.	Побудова та обґрунтування архітектурного рішення	22.09.23-28.09.23	
4.	Моделювання генерації трафіка в SDN мережі	29.09.23-05.10.23	
5.	Реалізація системи безперервної потокової передачі повідомлень	06.10.23-19.10.23	
5.	Розробка процесу обробки, агрегації та акумуляції мережевих даних	20.10.23-09.11.23	
6.	Тестування моделі та аналіз її ефективності.	10.11.23-13.11.23	
7.	Розробка стартап-проекту	14.11.23-18.11.23	
8.	Оформлення магістерської дисертації.	19.11.23-24.11.23	

Студентка Діана ШПАК _____
(підпис)

Науковий керівник дисертації Долголенко О. М. _____
(підпис)

РЕФЕРАТ

на магістерську дисертацію

виконану на тему: Фреймворк тестування навантаження веб додатку

студенткою: Шпак Діаною Михайлівною

Робота складається із вступу та п’яти розділів. Загальний обсяг роботи: 178 аркушів основного тексту, 123 ілюстрацій, 23 таблиць. При підготовці використовувалася література з 22 різних джерел.

Актуальність. Тестування навантаження дозволяє звести до мінімуму ризики низької продуктивності IT-систем під високим навантаженням. Також дає можливість оцінити якість роботи застосунку при нормальному навантаженні.

Мета і завдання дослідження. Розробити фреймфорк який емулюватиме дії справжніх коритсувачів додатку, тим самим створюючи первну нагрузку в залежності від підтипу тестування навантаженності. Це дозволятиме отримати оцінку стану додатку та його слабкі місця

Об’єкт дослідження – готові фреймфорки та веб додатки з відкртим вихідним кодом та апі.

Предмет дослідження – процес аналізу існуючих фреймворків для тестування навантаження веб додатків, веб додатки, покращення розуміння даного процесу та імплементація певних елементів до фреймворку які призведуть до підвищення точності результатів тестування та оцінювання якості продукту.

Методи досліджень.

Наукова новизна одержаних результатів роботи полягає у розробці власного фреймфорку що:

- дозволятиме створювати нагрузку на додаток з особисто регульованими параметрами аби протестувати всі можливі слабкі місця системи
- буде також безкоштовним рішенням з відкритим вихідним кодом, але лише для спеціалістів які вже мають технічні знання на певному рівні

Особистий внесок здобувача. Магістерське дослідження є самостійно виконаною роботою, в якій відображено особистий авторський підхід та особисто отримані теоретичні та прикладні результати. Формулювання мети та завдань дослідження проводилось спільно з науковим керівником.

Практична цінність. Отримані результати можуть використовуватися на реальних проектах в айті компаніях.

Ключові слова

Тестування навантаження, проуктивність, веб додаток, апі тестування, інфраструктура як код, Gatling, Terraform, Ansible, Grafana, Docker, Consul, Prometheus, performance testing

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

Реліз – додавання нової версії веб сайту на середовище з реальними користувачами

Стрес тест - одна з форм тестування, яка використовується для визначення стійкості комп'ютерної системи або ж юридичної особи в умовах перевищення меж нормального функціонування.

Демон - (англ. daemon) — сервіс Unix та Unix-подібних операційних систем, що працює у фоновому режимі без прямого спілкування з користувачем.

СУБД - Систе́ма управління ба́зами да́них. набір взаємопов'язаних даних (база даних) і програм для доступу до цих даних. Надає можливості створення, збереження, оновлення та пошуку інформації в базах даних з контролем доступу до даних.

Мутабельна, мутабельність (від англ. Mutable — «той, що може змінюватися») — поширений термін, котрий описує здатність предмету (суб'єкту) змінювати свої основні характеристики, властивості.

Скейлинг - налаштування масштабування баз даних, сервісів, тощоу відповідно до потреби

Нода - це будь-який комп'ютер, підключений до блокчейн-мережі. Ноди децентралізованої мережі контактують за допомогою P2P-протоколів для обміну інформацією про блоки та транзакції.

Юай - (англ. user interface, UI), засіб зручної взаємодії користувача (людини) з інформаційною системою.

Креди - (англ. creadentials), комбінація логіна та пароля для будь-якого сервісу

Примаунтити (англ. mount)- Монтування в обчислювальній техніці використовується для визначення процесу додавання додаткового сховища чи інших

пристроїв до обчислювальної системи. Щоразу, коли пристрій масової інформації, як-от дисковод або флеш-накопичувач, стає доступним для наявного сховища, його потрібно підключити до системи

Кластер - група серверів, об'єднаних логічно, здатних обробляти ідентичні запити, і що використовуються як єдиний ресурс.

Датасоурс - (англ. DataSource) - це ім'я, дане з'єднанню, встановленому із сервером до бази даних

Баг - (англ. Bug) - це сленгове позначення програмної помилки. У свою чергу програмна помилка — це помилка у програмі або системі, через яку програма виявляє неочікувану поведінку і, як наслідок, видає результат, який не відповідає плану. Тобто, bug — це розбіжність між очікуваним і фактичним результатом.

Фікс - (англ. Fix) - виправлення багів програмного коду.

Ос - (англ. Operation System) - операційна система.

Степ - (англ. Step) - крок у виконанні сценарію тестування.

Продакшн - (англ. Production) - середовище додатку з реальними користувачами

Демо - (англ. Demonstration) - депонстрація готового продукту, або ж проміжного результату виконання.

ВСТУП

В сучасні дні складно уявити життєвий цикл розробки програмного забезпечення (SDLC) без етапу тестування навантаження додатку. Тестування навантаження оцінює продуктивність і надійність системи, такої як додаток, API або веб-сайт, під певним навантаженням. Додаток працює по-різному при різних типах трафіку.

Тестування навантаження імітує реальний трафік до вашої системи, щоб оцінити її продуктивність, даючи змогу виявити і запобігти потенційним проблемам, перш ніж вони видикнуть.

Коли йдеться про тестування навантаження, ми мається на увазі тестування системи за допомогою сучасних інструментів і практик, відходячи від застарілих рішень. Сучасне тестування навантаження приносить користь організації багатьма способами:

-Швидкі та якісні релізи

Якщо інтегрувати тестування навантаження з безперервними та автоматизованими процесами тестування, можна уникнути затяжних регресій програмного забезпечення та випускати більш якісні версії продукту ще швидше.

-Надає можливість правильно оцінити нову або змінену інфраструктуру

Зазвичай компанії хочуть змінити або масштабувати певний компонент інфраструктури аби підготуватися до зростаючого попиту або тому, що існуюча інфраструктура є малогабаритною. Інфраструктури на етапі А буде недостатньо для інфраструктури на етапі В бізнесу. Наприклад, Netflix почав з монолітної архітектури, але пізніше перейшов в архітектуру мікросервісів для забезпечення швидкого зростання користувачів - в даний час компанія має 231 мільйон платних членів. Netflix робить багато тестування, щоб забезпечити швидкість і надійність своєї платформи в міру зростання бізнесу, включаючи навантаження і стрес-тести. Мета такого тестування полягає в

тому, щоб оцінити продуктивність, надійність і масштабованість нової або оновленої інфраструктури.

-Мінімізувати вартість помилки. Найкращий спосіб знизити вартість помилки додатків - це відловити і полагодити дефекти на початку життєвого циклу розробки програмного забезпечення (SDLC).

Кент Бек, відомий інженер-програміст і автор, сказав в одній із своїх робіт «Extreme Programming Explaned: Embrace Change»: «Більшість дефектів в кінцевому підсумку коштують більше, ніж це коштувало б, щоб запобігти їм.»

За допомогою тестування наавантаження ми отримуємо кращі продукти, це факт. Адже коли ваш веб додаток працює справно, ваші користувачі будуть задоволені послугою, що є в кінцевому варіанті прибудковим для вашого бізнесу і всієї організації.

Наразі на просторах інтернету складно знайти готове якісне рішення яке б допомогло у проведенні тестування навантаження. Саме тому суть задачі, що вирішується в рамках магістерської дисертації, полягає в розробці фреймворку, орієнтованого на тестування навантаження веб додатків..

1 РОЗДІЛ. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Тестування продуктивності

Тестування продуктивності – визначає стабільність тестованого додатка, а також перевіряє показники того, наскільки швидко програма реагує на зовнішні впливи при різних типах і інтенсивності навантаження.. Метою тестування є виявлення помилок і вразливостей в системі, визначення швидкості завантаження та обробки даних, а також надійності програми.

Перевірити налаштування системи можна за допомогою:

- визначення кількості дозволених користувачів програми;
- вимірюванн часу виконання тієї чи іншої системної операції;
- визначення межі прийнятної продуктивності програми при різних рівнях навантаження.

Тестування продуктивності можна розділити на такі підкатегорії:

- навантажувальне;
- стресове;
- стабільності;
- конфігураційне.

1.2 Тестування навантаження

Навантажувальне тестування - дозволяє зрозуміти, чи може додаток працювати стабільно з навантаженням в межах дозволених і трохи перевищує ці межі. Цей тест перевіряє, як програма поводиться під високим навантаженням. Наприклад, навантаженням може бути певна кількість одночасних користувачів програми протягом певного періоду часу. Цей вид тестування дозволяє визначити час реакції важливих бізнес-операцій. Спостерігаючи за базою даних, серверами додатків і мережею, ви можете визначити слабкі місця в додатку.

Нижче розглянемо приклади навантажувального тестування.

Припустимо, у нас є діючий інтернет-магазин і наближається всіма улюблене свято – Новий Рік. В цьому випадку потрібно, щоб сайт витримав навантаження від численних користувачів та їх замовлень. Для цього виду тестування можна використовувати API, а також проаналізувати таймінги завантаження сайту при високій кількості користувачів. Для цього можна взяти значення завантаження в звичайний день і порівняти його зі значенням при великій кількості користувачів.

Розглянемо приклад навантажувального тестування консолі карти, де користувач розміщує великий потік інформації, навантажуючи систему. Під час тестування потрібно було відтворити цих користувачів і відправити великі обсяги даних і запитів на сервер.

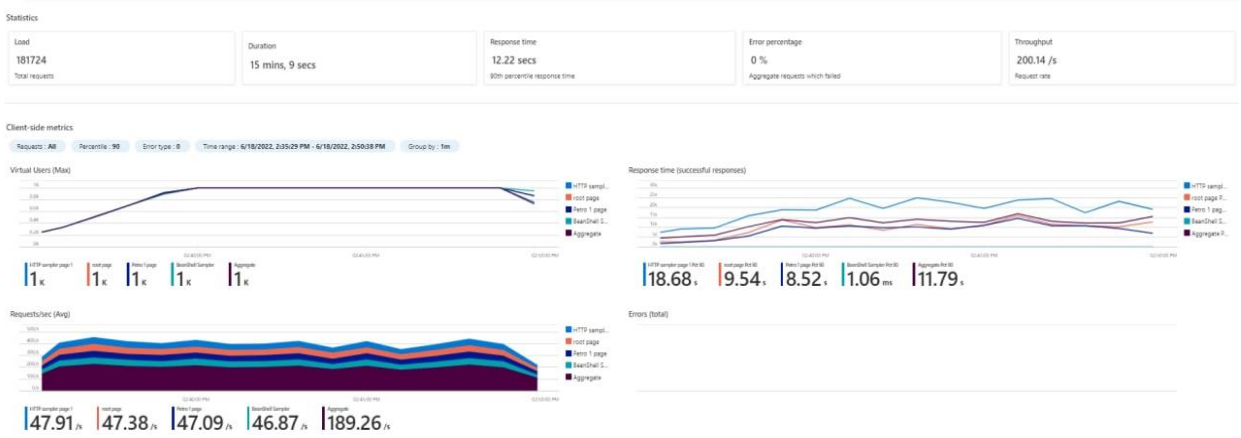


Рис. 1.1. Приклад графіків тестування навантаження

1.3 Конфігураційне тестування

Конфігураційне тестування перевіряє роботу програмного забезпечення в різних операційних системах, конфігурації програмного та апаратного забезпечення та їх сумісність. Під час цього етапу тестування параметри продуктивності системи вимірюються за допомогою середніх і максимальних значень навантаження. Цей тип тестування продуктивності гарантує, що система виконує однакові операції в різних конфігураціях і операційних системах.

Прикладом тестування конфігурації може бути перевірка того, як сайт запускається на пристроях із флагманською підкладкою (iPhone 12, Samsung Galaxy

S21) і п’ятирічних аксесуарах (Xiaomi Note 3, Nexus 5), а також порівняння продуктивності різних пристроїв. Веб-сайти на цих платформах. Завдяки цим перевіркам ви можете визначити найкращу конфігурацію свого пристрою для комфортної роботи у вашій програмі чи на сайті. Ви також можете перевірити веб-сайт на мобільних (iOS, Android) і настільних (Linux, MacOS, Windows) операційних системах.

1.4 Стрес тестування

Стрес тестування — це дослідження поведінки програми під час неочікуваних змін навантаження, які перевищують обчислювальні рівні. Цей тест визначає, як програми та системи працюють у нестандартних умовах, а також дозволяє оцінити, чи здатна система продовжувати роботу після нестандартних умов. Стресом може бути збільшення інтенсивності роботи (досягнення досить високих значень) або екстрена зміна конфігурації сервера.

Для стрес-тестування ви берете звичайний ПК і тестуєте його компоненти: процесор, оперативну пам’ять тощо. Тестування ЦП використовує консольні команди, які завантажують усі доступні ядра, наприклад:

- stress ng --class cpu --sequential 8 --timeout 60s --metrics-brief

Буде виконувати математичні операції, сортування, шифрування, стиснення, пошук, операції з рядками.

Для тестування оперативної пам’яті ви можете використовувати команди консолі, такі як:

- stress-ng --class memory --sequential 8 --timeout 60s --metrics-brief

1.5 Тестування стабільності

Тестування стабільності перевіряє, як довго програма може працювати стабільно при середньому навантаженні. При цьому також необхідно виявити витоки пам'яті, некоректні налаштування програмного забезпечення, перевірити перезавантаження сервера під навантаженням та інші аспекти, що впливають на

стабільну роботу програми. Але в цьому типі тесту час, необхідний для виконання дії, стоїть на другому місці.

Для перевірки стабільності можна використовувати програму Apache Bench, яка надсилає вказану кількість запитів і кількість потоків для цих запитів. У цьому тесті можна виділити основні компоненти: кількість запитів за секунду, середній час виконання запиту за секунду (Time per request) і середній час виконання запиту (Time per request). Використовуючи утиліту Httperf та її запити, ви можете визначити, як швидко створюються нові з'єднання (швидкість з'єднань), скільки часу потрібно з'єднанню від ініціалізації до закриття (час з'єднання) і як швидко обробляються запити сервера за секунду (швидкість запитів).

1.6 Сценарії тестування

Тестування продуктивності також можна розділити на тестування за сценаріями.

При тестуванні веб сайту перевіряється на:

- Кількість користувачів – кількість запитів, виконаних за певний період часу;
- Час відповіді – скільки часу потрібно для виконання запиту користувача;
- реквестів за секунду – кількість запитів, надісланих користувачем на сервер;
- Транзакцій за секунду – вимірює кількість транзакцій, які користувач надсилає на сервер;
- Відсоток помилок – перевірте відсоток помилок від загальної кількості відповідей користувачеві за одиницю часу;
- Процесор – перевірити навантаження на процесор під час виконання завдання;
- RAM – Перевірте використану фізичну пам'ять у МБ;
- Жорсткий диск - Перевірте використання дискового простору.

1.7 Інструменти для тестування продуктивності

Засоби тестування продуктивності.

Тестування продуктивності використовує різні інструменти для полегшення роботи тестувальника. Нижче наведено кілька прикладів таких інструментів.

- LoadView є одним із найефективніших інструментів тестування продуктивності. Найпопулярнішими функціями цього інструменту є сценарії «вкажи та клацни», глобальна хмарна інфраструктура та тестування браузера в реальному часі.
- Apache JMeter — це інструмент навантажувального тестування, розроблений Apache Software Foundation. Ця програма реалізує механізм віртуальної автентифікації користувача та підтримує сесії користувачів. Систематично записуйте результати тестування та візуалізуйте результати в різних форматах, таких як діаграми та таблиці.
- MSI Afterburner – надає можливість переглядати різні властивості за допомогою накладень і відстежувати споживання ресурсів вашою грою. Безкоштовні інструменти, багаті налаштування та простий у використанні. Завантажити його можна за посиланням. Ви можете контролювати FPS, використання процесора, використання графічного процесора, використання оперативної пам’яті, швидкість охолодження тощо.
- FPS Meter (Android) – FPS Meter відображає FPS (кадри в секунду) у всіх вікнах вашої гри чи програми. Це дозволяє виміряти фактичну продуктивність вашого пристрою.
- GameBench (Android та iOS) – цей інструмент чудовий, оскільки дозволяє запускати будь-яку гру на вашому пристрої та запускати її у фоновому режимі, а також відстежувати та записувати продуктивність і використання акумулятора. А в цей час користувач спокійно грає.

ВИСНОВКИ ДО РОЗДІЛУ 1

У даному розділі було розглянуто визначення терміну тестування продуктивності, виділено підвиди цього підходу, та наведені інструменти за допомогою яких можна провести тестування навантаження.

Розглянуто теоретичні та практичні аспекти тестування навантаження, досліджено сценарії перевірок та основні атрибути верифікації веб додатків.

Спираючись на дані аналізу, сформульовано основні задачі для розробки власного фреймворку для тестування навантаження веб додатків, що дозволятиме оцінювати якість продукту та масштабувати його ж або модифікувати на основі результатів тестування.

2 РОЗДІЛ. ІНСТРУМЕНТИ РЕАЛІЗАЦІЇ

2.1 Інструменти для проекту з тестами

2.1.1 Gatling.

Для написання наших лоадтестів буде використана мова програмування Gatling.

Gatling — це потужний інструмент навантажувального тестування з відкритим вихідним кодом, розроблений, щоб допомогти розробникам і інженерам продуктивності перевірити продуктивність і масштабованість своїх веб-додатків. Він написаний на Scala і заснований на наборі інструментів Akka , що означає, що він є розширюваним і легко інтегрується з іншими інструментами.

Однією з ключових особливостей Gatling є його інтуїтивно зрозумілий і простий у використанні DSL (domain-specific language) для створення тестових сценаріїв. Це дозволяє розробникам писати навантажувальні тести природною та зрозумілою мовою, що полегшує розуміння та підтримку. Gatling також підтримує кілька протоколів, таких як HTTP, HTTPS, WebSockets і JMS, і його можна використовувати для тестування програм, написаних різними мовами та запущених на різних платформах.

Gatling також надає детальні та точні результати з можливістю моніторингу та звітності в реальному часі. Він створює HTML-звіти, які надають чіткий і стислий огляд результатів тестування, включаючи деталі запитів і відповідей, розподіл часу відповіді та статистику помилок. Gatling також дозволяє налаштовувати звіти, додаючи власні діаграми та статистику.

Ще однією перевагою Gatling є висока продуктивність. Він розроблений для роботи з великою кількістю одночасно працюючих користувачів, що робить його ідеальним для тестування веб-додатків із високим трафіком. Архітектура Gatling заснована на Akka, який є високопродуктивним набором інструментів для створення паралельних і розподілених систем. Це дозволяє Gatling обробляти велику кількість запитів з мінімальним використанням ресурсів.

Gatling — це потужний і гнучкий інструмент тестування навантаження, який простий у використанні та надає детальні та точні результати. Його інтуїтивно зрозумілий DSL, підтримка кількох протоколів і висока продуктивність роблять його ідеальним вибором для тестування веб-додатків будь-якого розміру та складності.

Важливо відзначити, що Gatling також має комерційну версію під назвою Gatling Frontline, яка пропонує додаткові функції, такі як розподілене тестування, розширена аналітика та звітність у реальному часі.

2.1.2 IntelliJ IDEA

IntelliJ IDEA - провідне середовище розробки для Java та Kotlin від JetBrains.

Виділяється наступним:

- Найдосконаліший контекстно-залежний редактор, який надає підказки щодо завершення коду під час введення, забезпечує виправлення помилок та попереджень і може знайти будь-що, від класів до вікон інструментів.
- Завдяки глибокому розумінню коду, IDE може миттєво виявляти помилки та надавати відповідні підказки в будь-якому контексті.
- Готовий до роботи досвід. Це означає, що ви можете почати кодувати з першого запуску. Важливі інструменти та широкий спектр підтримуваних мов доступні одразу, і немає проблем з плагінами.
- Інструменти для спільної та віддаленої розробки.

Наразі існує дві версії IntelliJ IDEA:

IntelliJ IDEA Community Edition - безкоштовна версія, яка має достатній функціонал, щоб задовольнити більшість базових потреб розробки на Java та Kotlin.

IntelliJ IDEA Ultimate - повнофункціональна комерційна версія, яка підтримує різноманітні фреймворки та технології для розробки серверних та фронтенд інтерфейсів. Вона надає розширений набір інструментів для оптимізації робочих процесів. Профілювання, інструменти для роботи з базами даних та HTTP-клієнт доступні з коробки.

2.2 Інструменти для інфраструктури

2.2.1 Hetzner

Для розгортання нашої інфраструктури нам буде потрібен інтернет хостінг, який нам дозволить динамічно створювати сервери і динамічно деплоїти наші проекти, і для цього ми обрали **Hetzner**.

Hetzner є Південноафриканським веб-хостингом, який надає свої послуги протягом 20 років та має більш ніж 40 000 клієнтів. Компанія виграла цілий ряд призів та нагород, включаючи вихід у фінал в конкурсі «National Business Awards» у 2016 та 2018 роках, та срібну нагороду в конкурсі «Stevie Awards» у 2019 році за високу якість служби технічної підтримки.

Hetzner пропонує широкий асортимент хостингових пакетів: від віртуального хостингу та серверів із самостійним управлінням до спільного розташування (колокації) та оренди виділених серверів.

Ми будемо використовувати один із серверів Hetzner з самостійним управлінням, на Hetzner ми можемо обрати операційну систему, якою будемо користуватись, включаючи обидві, Linux та Windows. Крім того, нам буде надано кореневий/адміністраторський доступ до нашого серверу та до 64 ГБ оперативної пам’яті, однак трафік буде обмежено до 6 ТБ. Більшість планів Hetzner, так чи інакше, містять обмеження трафіку

Особливості Hetzner:

- Надає оренду виділених серверів, стойки colocation, віртуальний і VPS хостинг, реєстрацію доменів і оренду диска для зберігання даних.
- Тестовий період не передбачений ні для однієї з послуг. При реєстрації необхідно внести оплату згідно з вибраним тарифом.
- Віртуальний хостинг підійде для персональних сторінок, блогів, форумів, інтернет-магазинів. Він надається з DNS-адмініструванням, FTP-доступом, щоденним резервним копіюванням і захищеним паролем. Доступ до статистики здійснюється за допомогою інструменту Report Magic.

- Можливе підключення CMS (наприклад WordPress, Joomla). Хостинг VPS називається «vServer» і надається з операційними системами на вибір: Debian, Ubuntu, CentOS, openSuSE. Для управління можна підключити Plesk або cPanel.
- Провайдер надає бази даних на вибір (MySQL або PostgreSQL), захист від DDos-атак і 24-годинний моніторинг. Доступні домени, включені в тарифи віртуального хостингу: .de, .com, .net, .org, .info, .biz, .eu, .name або .nl.
- SSL-сертифікати надаються тільки на платній основі. Підтримка виявляється на російській мові тільки для технічних консультацій по виділеним серверам, по інших питаннях — на англійській.

2.2.2 Docker

Всі наші аплікейшини будуть запаковані в докери. **Docker** – це програмне забезпечення, яке дає можливість на певній ділянці пам'яті ізольовано встановити необхідну ОС (операційну систему), версію Java, налаштувати змінні оточення, встановити різні залежності та надати доступ лише за певних умов. При цьому цю програму зовсім не хвилюватиме, що відбувається довкола. Docker вирішує безліч завдань, пов'язаних зі створенням контейнерів, розміщенням в них додатків, управлінням процесами, а також тестуванням ПЗ і його окремих компонентів.

Він потрібен для більш ефективного використання системи і ресурсів, швидкого розгортання готових програмних продуктів, а також для їх масштабування і перенесення в інші середовища з гарантованим збереженням стабільної роботи.

Docker допомагає:

- мінімально використовувати ресурси;
- зручно приховати фонові процеси;
- просто масштабовувати;
- зменшити час між написанням і запуском коду;
- швидше тестувати;
- швидко розгортати;
- швидше створювати додатки.

Docker робить це за допомогою легкої платформи контейнерної віртуалізації.

Для початку необхідно встановити Docker на наші сервери, щоб можна було створювати, налаштовувати та запускати контейнери. Схема створення контейнера виглядає так:

Рис. 2.1. Схема створення контейнера

1) Створюєте 'Dockerfile' — файл, у якому необхідно описати, як створюватиметься образ. Простими словами - це опис того, як виглядатиме ваша кімната;
2) Image - це образ, на основі якого надалі буде запущено контейнер. Це дизайнерський проект вашої кімнати, чітка схема того, що і де стоятиме;
3) Container - це запущений образ, в якому працює ваша програма з описаними залежностями згідно з інструкцією. Тобто це вже готова кімната, в якій ви можете жити.

2.2.3 Terraform

Щоб створювати ресурси без ризику щось забути і робити це швидко при повторенні – вигадали Infrastructure as Code. Це підхід до управління інфраструктурою через набори конфігураційних файлів (коду), у яких декларативно (тобто специфікації фінального результату) описується бажаний стан цієї інфраструктури.

Такий підхід вирішує багато завдань та проблем, наприклад:

- Вона часто є і документацією.

- Повторне застосування цієї конфігурації (наприклад, вам потрібно кілька оточень) гарантовано призводять до того ж результату.
- Усувається людський фактор (коли забули щось додати чи налаштувати)

Terraform – це один із інструментів реалізації Infrastructure as Code.

Terraform (утиліта командного рядка) працює з конфігураційними файлами (кодом) та інтерпретує їх усередині cloud або on-premise платформи – створює, змінює чи видаляє ресурси.

Terraform підтримує велику кількість платформ: від основних хмарних провайдерів (AWS, Azure, GCP) до скромніших платформ, наприклад Hetzner або 1&1. Крім того, він підтримує роботу з таким програмним забезпеченням, як Docker, Kubernetes, Chef, що розширює функціонал і дозволяє використовувати їх у зв'язці. Тому Terraform такий популярний – це справжній швейцарський ніж у світі управління інфраструктурою.

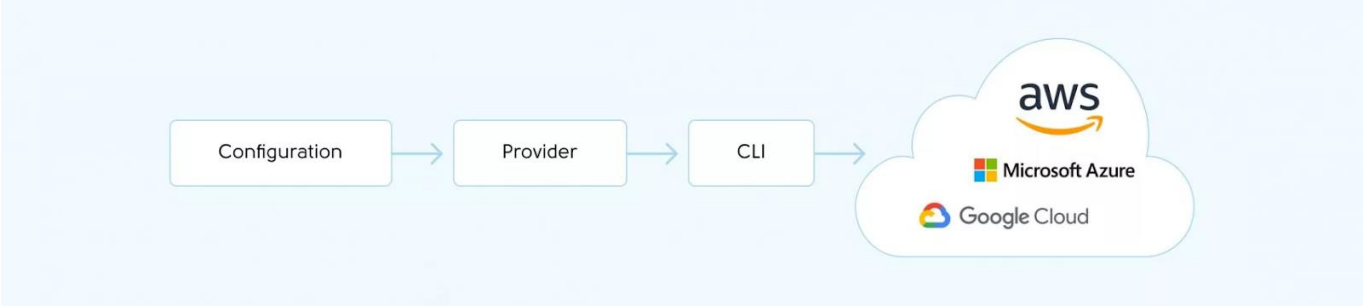


Рис. 2.2. Terraform (утиліта командного рядка)

Код Terraform використовує свій власний синтаксис, але виглядає дуже доброзичливо. Наприклад, так виглядає мінімальний блок коду, який створює EC2 instance в AWS.

```
ресурс "aws_instance" “web_server” {
ami = "ami-a1b2c3d4"
instance_type = "t3.micro"}
```

Terraform підтримує безліч так званих провайдерів, які дозволяють інтерпретувати код для тієї чи іншої платформи хостингу.

Щоб застосувати конфігурацію, використовується CLI-додаток, який запускає код на обробку та застосування провайдером по відношенню до тієї чи іншої платформи.

При першому використанні Terraform відбувається ініціалізація проекту - каталогу, де знаходяться файли конфігурації (ваш код). Для цього використовується команда terraform init. Ця команда дозволяє завантажити необхідні версії провайдерів та інших залежностей, які використовуються у конфігурації інфраструктури.

Далі під час першого застосування команди terraform apply Terraform створить описані ресурси на хостинг-платформі та створить спеціальний файл – state file – у каталозі проекту.

Саме цей файл буде використовуватися Terraform для порівняння поточного стану вашої інфраструктури з новою версією вашої конфігурації щоразу при виконанні terraform apply. У такий спосіб він зрозуміє: які ресурси змінити, видалити чи додати.

У синтаксису Terraform є безліч інструментів для організації коду та розширення можливостей роботи з ним, наприклад:

- Modules – дозволяють створювати логічні блоки (абстракції) з різних наборів ресурсів та знову використовувати такі блоки надалі.
- Variables – дозволяють параметризувати код, роблячи його універсальнішим.
- Expressions — дозволяє обчислювати або звертатися до різних типів даних у коді. Наприклад, взяти елемент зі списку або вибрати якесь значення на підставі умови.
- Functions — вбудовані функції, які ще більше розширюють можливості роботи з даними всередині конфігурації. Наприклад, перетворення рядків, математичні обчислення, робота зі списками тощо.

Це та багато іншого робить Terraform потужним інструментом для управління інфраструктурою: як усередині однієї хостинг-платформи, так і між кількома одночасно.

2.2.4 Ansible

Для автоматизації конфігурації й управління комп’ютерними системами ми будемо використовувати **Ansible**. Він забезпечує можливість описувати задачі (playbooks) у простому для розуміння YAML-синтаксисі, що дозволяє автоматизувати налаштування серверів, застосунків, баз даних та інших системних компонентів. Код Ansible написаний мовою Python і розповсюджується під ліцензією GPLv3.

Ansible може працювати з багатьма типами систем, включно з Linux, macOS і Windows. Він не потребує встановлення жодного агента на кожному вузлі (на відміну від конкурентів, як-от Chef), що спрощує його використання. Замість цього Ansible використовує SSH-з’єднання для взаємодії з системами.

Перше, що нам потрібно зробити після установки, налаштувати файл hosts. Туди повинні бути занесені всі наші хости, з якими ми будемо працювати. Хост файл може бути написаний у кількох форматах, як-от ini або yaml.

Playbook – це сценарій, у якому ви вказуєте дії, які повинен виконати Ansible на наших хостах.

В Ansible є купа модулів, які можна використовувати для роботи з установкою пакетів і закінчуючи створенням баз даних.

За допомогою Ansible можна виконувати різні завдання, включно з такими:

- налаштування серверів і застосунків;
- деплоймент програмного забезпечення;
- керування конфігураціями;
- автоматизоване тестування;
- моніторинг систем;
- і багато іншого.

Загалом Ansible дозволяє зменшити витрати на управління системами та збільшити швидкість виконання задач. Як інструмент для автоматизації конфігурації, розгортання й управління інфраструктурою, що базується на моделі з описом стану

системи. Вони дають змогу ефективно перевикористовувати код і ділитися ним. Ролі дають нам чітко встановлену структуру для виконання завдань.

Роль Ansible – набір завдань або обробник змінних, файлів та інших артефактів, які поширюються і підключаються як єдине ціле до плейбуку.

У міру того, як ви додаєте у плейбуки функціональність, вони стають все більш громіздкими й складними в обслуговуванні та читанні коду. Тут на допомогу приходять ролі – вони розбивають складні плейбуки на окремі дрібніші фрагменти, які можуть координуватися центральною точкою входу.

Основна ідея ролей полягає у тому, щоб дозволити повторно використовувати спільні кроки налаштування між різними типами серверів.

Стандартна структура ролі в Ansible виглядає так:

```
roles/
common/              # ця ієрархія показує роль
tasks/           #    <-- файли із задачами, які буде виконувати Ansible
main.yml      #
handlers/        #   <-- хендлери для Ansible* (примітка, нижче пояснюю, що це таке)
main.yml      #
templates/       #  <-- темплейти, які ролі буде деплоїти

ntp.conf.j2   #

files/           #      <-- файли, які ролі буде використовувати

bar.txt       #

foo.sh        #

vars/            #     <-- змінні, які буде використовувати роль

main.yml      #

defaults/        #   <-- також змінні, але з найнижчим пріоритетом
```

main.yml #

meta/ # <-- метадані для ролі, містять залежності ролі

main.yml #

2.2.5 Git Hub

Життя програміста - це не просто сидіння на самоті в темній кімнаті, освітленій лише світлом монітора. Йому також потрібно спілкуватися з іншими програмістами та брати соціальну участь у різних спільнотах. Саме тому були створені та популяризовані такі речі, як Git та Github.

Git - це програмне забезпечення для контролю версій, яке дуже спрощує співпрацю з колегами по команді.

Git - це програмне забезпечення для контролю версій. Його створив Лінус Торвальдс (який винайшов Linux); вам не обов'язково мати Linux, щоб використовувати його на Windows або Mac.

За допомогою Git'у ви можете відстежувати всі ревізії (зміни), які ви та ваша команда зробили під час розробки програмного забезпечення. Використовуючи єдине сховище коду, кожна людина працює незалежно, а потім об'єднується. І вам не потрібно постійно бути підключеним до репозиторію, оскільки проекти зберігаються як локально, так і віддалено на кожній машині (і на Github).

Що робить Git особливим (і дуже важливим), так це те, що ви можете повернутися до будь-якої попередньої версії коду, перейти на іншу гілку або розробити певну функцію, не впливаючи ні на що і ні на кого, що виключає можливість пошкодження даних через спільний доступ.

Він також дозволяє легко реалізувати шифрування і забезпечує асинхронні, нелінійні робочі процеси, так що ви можете працювати над потрібною частиною проекту, де б ви не знаходилися.

2.3 Інструменти для системи моніторингу

2.3.1 Prometheus

Також нам потрібно буде налаштувати моніторинг який дозволяє дізнатися, коли сервер\додаток виходить з ладу, і коли з'являються до цього передумови, також дозволить нам моніторити різні параметри наших серверів. Це все ми будемо налаштовувати використовуючи наступний стек Prometheus, Grafana та Node Exporter.

Prometheus – серце моніторингу. Це екосистема, що дозволяє збирати метрики з фізичних серверів, VPS, окремих додатків, різних баз даних, контейнерів та пристроїв IoT. Prometheus включає сотні готових рішень для збору інформації і дозволяє одночасно моніторити тисячі служб. При необхідності, в міру вивчення, досвідчений адміністратор може йому написати свої сценарії.

2.3.2 Node exporter

Node exporter - являє собою готову програму, експортер метрик.

Prometheus для збору даних про стан сервера в середу візуалізації.

2.3.3 Grafana

Grafana – середовище візуалізації. Використовує метрики, які зібрав Prometheus, і відображає їх у вигляді інформативних графіків та діаграм, організованих у панелі, що настроюються.

Розглянемо основні переваги Grafana:

- Новий редактор панелей. Редизайн на основі відгуків спільноти.
- Новий інтерфейс трасування та підтримка візуалізації трасувань Jaeger та Zipkin.
- Аналіз корпоративного використання, індикатор присутності користувачів, покращення автентифікації.
- Дані, які ви хочете візуалізувати, можуть надходити з різних місць, і вони не завжди правильні. Тепер користувачі можуть перетворювати дані на таблиці, що не належать до тимчасових рядів (наприклад, файли JSON або навіть прості таблиці пошуку).

- Поліпшений дашборд. Він підтримує горизонтальне прокручування та зміна розміру стовпців. Ви можете змінювати порядок, приховувати та перейменовувати стовпці. Ця нова панель також підтримує нові режими відображення осередків.
- Нові платформи плагіни.
- Оновлений tutorials.
- Підтримка Amazon CloudWatch для журналів Cloudwatch.
- Вилучено PhantomJS.
- Підтримка часових поясів.

ВИСНОВКИ ДО РОЗДІЛУ 2

У даному розділі було розглянуто інструментарій необхідний для реалізації фреймфорку тестування навантаження.

Розписана детальна характеристика кожної із технологій та наведені приклади використання, аби мати розуміння складових фреймворку. Також описані альтернативні інструменти для реалізації проекту, їх переваги та недоліки, базуючись на чому можна обрати зручний для себе шлях створення фреймворку.

Запропоновано спосіб імплементації що враховує особливості роботи веб додатків та використання найоптимальніших технологій.

Розглянуто проміжні етапи побудови фреймворку для вирішення задачі тестування навантаження. Надано вирішення можливих проблем під час розробки та можливості масштабування.

3 РОЗДІЛ РОЗРОБКА ФРЕЙМВОРКУ НАВАНТАЖЕННЯ І СИСТЕМИ МОНІТОРИНГУ

3.1 Загальна архітектура фреймворку навантаження

3.1.1 Архітектура системи моніторингу

Наш фреймворк для тестування продуктивності складається з трьох основних частин:

1) Самого застосунку який нам потрібно тестувати.
2) Системи моніторингу.
3) Системи навантаження.

Загальна структура фреймворку навантаження показана на рис. 3.1.

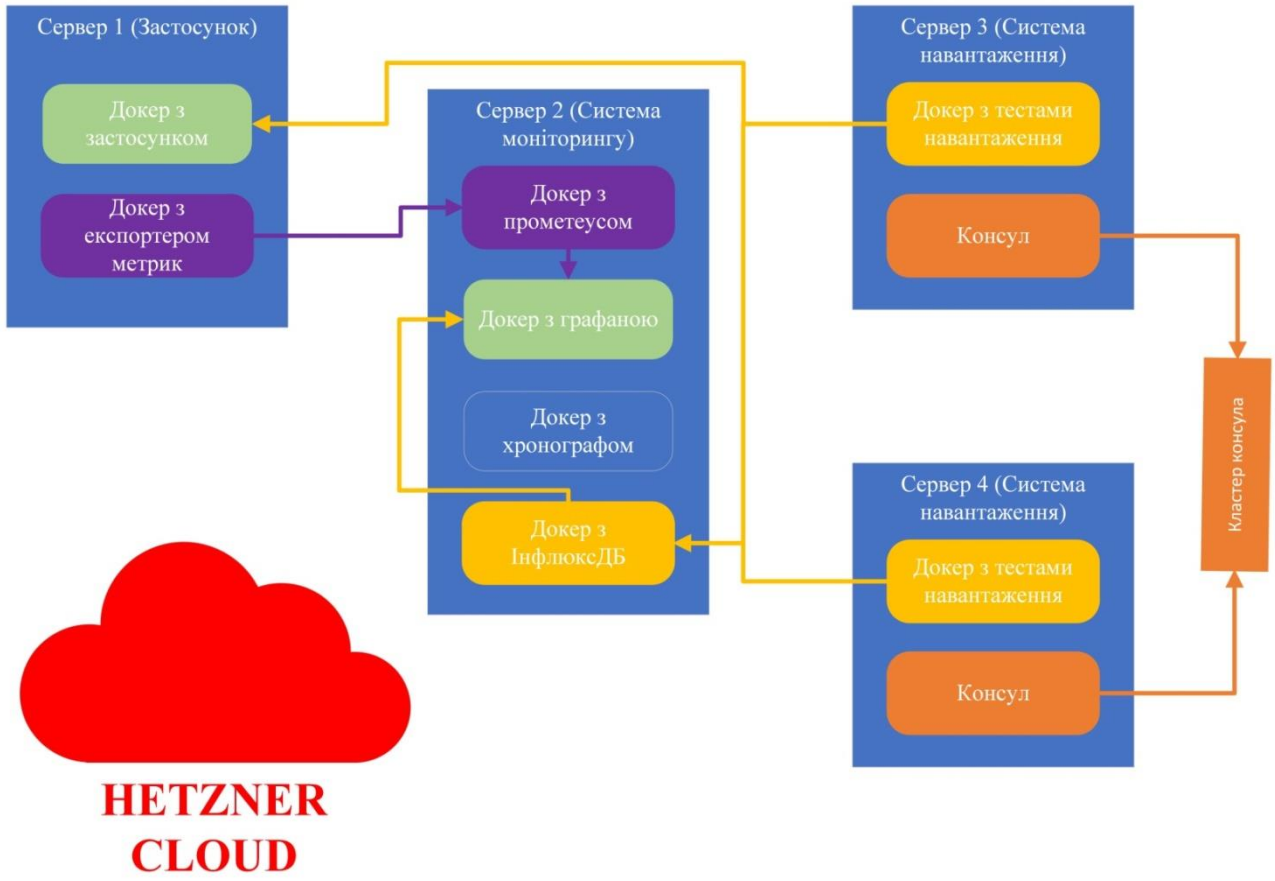


Рис. 3.1. Загальна схема фреймворку тестування навантаження

Застосунок являє собою клієнт-серверний додаток, який буде запакований в докер, що дасть нам можлжливість запускати наш застосунок в ізольованому контейнері на віддаленому сервері. Всі наші сервери будемо запускати на Hezner Cloud, завдяки чому можна змінювати конфігурацію нашого сервера відповідно навантаження який він повинен витримати. На рис.3.2. показано характеристики які будуть мати наші сервери.

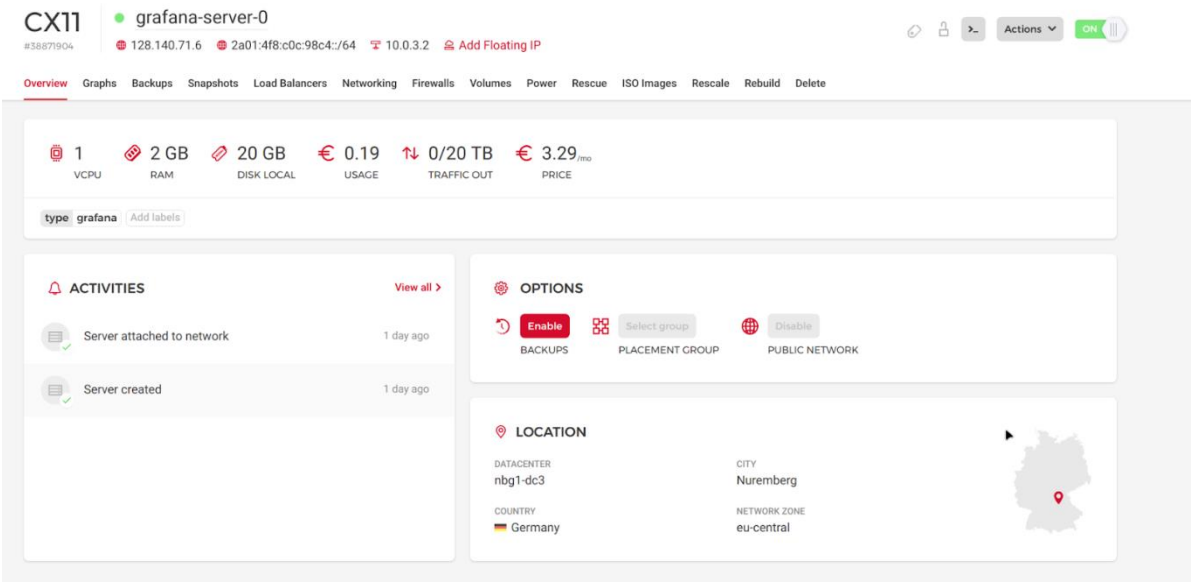


Рис. 3.2. Основні параметри сервера

Також на даному сервері потрібно буде задеплоїти експортер метрик. Це програма яка буде збирати метрики на хості та надаватиме до них доступ по певному HTTP порту. Ми будемо використовувати найбільш поширений експортер - node exporter, який збиратиме метрики з ОС і сервера. Всі метрики будуть надаватися в форматі plain-text, це зроблено для зручності парсингу. Також можна забирати дані за допомогою самописних скриптів на bash, python та інших. Самі метрики складаються з назви, значень та інколи додаткових даних, вкладених у фігурні дужки. Такі метрики називають складовими чи складними:

node_sockstat_sockets_used 5

 # HELP node_softnet_backlog_len Softnet backlog status

 # TYPE node_softnet_backlog_len gauge

node_softnet_backlog_len{cpu="0"} 0

 # HELP node_softnet_cpu_collision_total Number of collision occur while obtaining device lock while transmitting

 # TYPE node_softnet_cpu_collision_total counter

node_softnet_cpu_collision_total{cpu="0"} 0

 # HELP node_softnet_dropped_total Number of dropped packets

 # TYPE node_softnet_dropped_total counter

node_softnet_dropped_total{cpu="0"} 0

 # HELP node_softnet_flow_limit_count_total Number of times flow limit has been reached

 # TYPE node_softnet_flow_limit_count_total counter

node_softnet_flow_limit_count_total{cpu="0"} 0

 # HELP node_softnet_processed_total Number of processed packets

 # TYPE node_softnet_processed_total counter

node_softnet_processed_total{cpu="0"} 30103

 # HELP node_softnet_received_rps_total Number of times cpu woken up received_rps

 # TYPE node_softnet_received_rps_total counter

node_softnet_received_rps_total{cpu="0"} 0

 # HELP node_softnet_times_squeezed_total Number of times processing packets ran out of quota

 # TYPE node_softnet_times_squeezed_total counter

node_softnet_times_squeezed_total{cpu="0"} 0

 # HELP node_textfile_scrape_error 1 if there was an error opening or reading a file, 0 otherwise

 # TYPE node_textfile_scrape_error gauge

node_textfile_scrape_error 0

 # HELP node_time_clocksource_available_info Available clocksources read from '/sys/devices/system/clocksource'.

 # TYPE node_time_clocksource_available_info gauge

node_time_clocksource_available_info{clocksource="acpi_pm",device="0"} 1

node_time_clocksource_available_info{clocksource="hpet",device="0"} 1

node_time_clocksource_available_info{clocksource="kvm-clock",device="0"} 1

node_time_clocksource_available_info{clocksource="tsc",device="0"} 1

 # HELP node_time_clocksource_current_info Current clocksource read from '/sys/devices/system/clocksource'.

 # TYPE node_time_clocksource_current_info gauge

node_time_clocksource_current_info{clocksource="kvm-clock",device="0"} 1

Метрики збираються так званими колекторами, які можна включати і вимикати за допомогою директив при запуску node exporter. Це буває дуже корисно, коли у вас багато серверів, з яких потрібно збирати інформацію. Дані, що збираються і транслюються на http-порт забирає Prometheus, але завдяки простому text-plain формату дані легко парсяться і можуть бути перевикористані іншими програмами.

Система моніторингу за додатком у нас складається з експортера, який у нас буде розгорнутий на одному сервері з додатком, і сервера з графаною і прометеусом. Основні частини системи моніторингу показані на рис. 3.3.

Рис. 3.3. Структура системи моніторингу

Prometheus - це система моніторингу та оповіщення про події, що зберігає дані у вигляді часових рядів. Prometheus так само як і node exporter — це програма, один бінарний файл, який запускається як демон і збирає дані, отримані від node exporter, у задане в конфігураційному файлі місце. На відміну від багатьох інших СУБД, наприклад, PostgreSQL, Prometheus сам опитує зазначені в конфігураційному файлі сервери та порти рис.3.4.

Рис. 3.4. Схема роботи прометеусу

За замовчуванням web-інтерфейс Prometheus працює на порті 9100, але ми його змінимо на 9090. Відкривши цей інструмент ви побачите наступний інтерфейс рис.3.5.

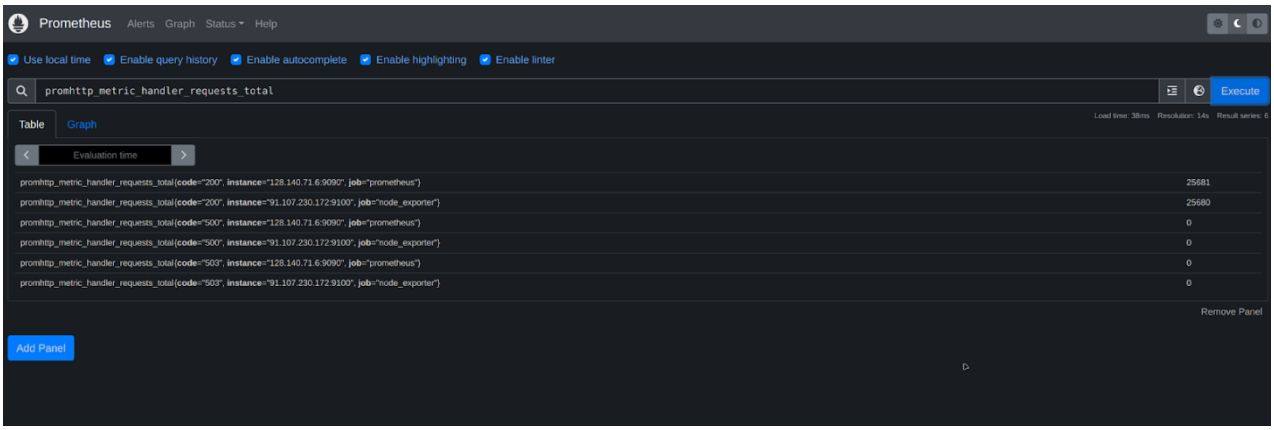


Рис.3.5. Інтерфейс Prometheus

Якщо розглянути висновок метрик node exporter уважніше, можна помітити, що кожна секція колектора починається з двох рядків вступної інформації: перший рядок — це короткий опис статистики що збирається, а другий рядок — опис типу метрики. Є 4 типи метрик (2 прості метрики та 2 складові):

- Counter це просто лінійний лічильник, який може тільки збільшуватися. Наприклад, кількість відвідувачів сайту;
- Gauge – це двонаправлений лічильник, значення якого може як збільшуватись, так і зменшуватись. Наприклад, кількість потоків у ОС;

- Histogram – це комбінація різних лічильників, що використовується для відстеження розмірних показників та їх тривалості, таких як тривалість запитів;
- Summary (Зведення) працює як гістограма, але також розраховує квантил.

Типи метрик gauge, histogram і summary підходять для побудови графіків. Ось наприклад ми будуємо графік за метрикою go_gc_duration_seconds, яка відображає час тривалості виклику GC (Go client) для frontend-сервера зі значенням квантилю в 0.25 , приклад метрики показано на рис.3.6.

Рис.3.6 Приклад метрики з Prometheus

Для формування запитів у Prometheus використовується мова запитів для TSDB - PromQL. Він використовується і в Grafana. PromQL потужний, функціональний та відносно простий мову запитів, що дозволяє використовувати агрегатори та оператори. Це означає, що ви можете робити складні вибірки за багатьма параметрами та проводити з ними математичні операції.

Ось наприклад, запит, який повертає обсяг вільної пам'яті на сервері, що спостерігається в байтах, які ми відразу перевели в мегабайти за допомогою оператора поділу:

*node_memory_MemFree_bytes / (1024 * 1024)*

Так ми можемо дізнатися відсоток вільної пам'яті на сервері і створити на базі цього запиту повідомлення при критичному рівні вільної пам'яті менше 5%:

*round((node_memory_MemFree_bytes / node_memory_MemTotal_bytes) * 100) #Дізнатися залишок вільної пам'яті*

*round((node_memory_MemFree_bytes / node_memory_MemTotal_bytes) * 100) < 5 #Умова оповіщення при критичному залишку вільної пам' яти меньше 5%*

Звичайно можна будувати і складніші запити для кількох метрик, порівнювати їх між собою, знаходити середнє і багато іншого, але зазвичай такі обчислення не проводяться використовуючи Prometheus, тому що важливіше бути в курсі поточної ситуації по всій інфраструктурі і оперативно реагувати на інциденти.

Як правило, системи моніторингу навантаження складаються із трьох і більше компонентів. У найпростішому варіанті система може складатися з агента, що передає дані в сховище та системи візуалізації даних.

У випадку з класичною схемою роботи Grafana, має три елементи:

- Агент, який передає дані - node explorer;
- Додаток для моніторингу подій та оповіщень - Prometheus;
- Система візуалізації даних – Grafana.

Інша не менш популярна система моніторингу навантаження Zabbix використовує дещо іншу модель роботи. У ній чотири елементи:

- Ядро, яке керує всіма процесами системи моніторингу;
- Проксі сервер – зберігає статистику у вибраному форматі (SQL);
- Агент - програма, що збирає статистику на серверах, що спостерігаються;
- Web-інтерфейс, що відбиває отримані дані як графіків.

Незалежно від кількості елементів у системі моніторингу навантаження, вона працює за одним загальним принципом: збір метрик агентом → передача їх у БД → візуалізація даних у web. Така схема може бути локальною та розподіленою. Розгляньмо кожен тип систем моніторингу.

Локальна система моніторингу навантаження - Як випливає з назви, локальна система моніторингу розташовується на тому сервері, який потрібно спостерігати. Така схема добре підходить, якщо у вас єдиний робочий сервер. Проте, така схема має очевидний істотний мінус — якщо сервер «впаде», то система моніторингу не зможе вам про це повідомити, оскільки теж працювати не буде.

Проблему стабільності роботи сервера можна вирішити просто – замовити надійний сервер у перевіреного провайдера. Наприклад, сервери 1cloud працюють під керуванням відмовостійкої віртуалізації Enterprise-рівня - це гарантує високий рівень надійності та відмовостійкості серверів.

Основною перевагою локальної системи моніторингу є простота її встановлення та налаштування. Сьогодні є багато готових рішень на базі Docker, де буквально в кілька команд можна розгорнути всю систему моніторингу навантаження.

Розподілена система моніторингу навантаження - як випливає з назви, система моніторингу навантаження — це система, елементи якої рознесені по різних серверах і пов'язані між собою мережею. Говорячи, простими словами: Grafana може бути на одному сервері, Prometheus на іншому сервері, а кількість спостережуваних серверів взагалі може бути якою завгодно.

Очевидно, що така система завдяки своїй розподіленості стійкіша до збоїв і високих навантажень, а також більш мутабельна — кожен її елемент може бути точно налаштований, однак є, і зворотний бік медалі — великі витрати на розгортання та обслуговування такої системи. На рис 3.7 показано роботу розподіленої системи моніторингу Grafana.

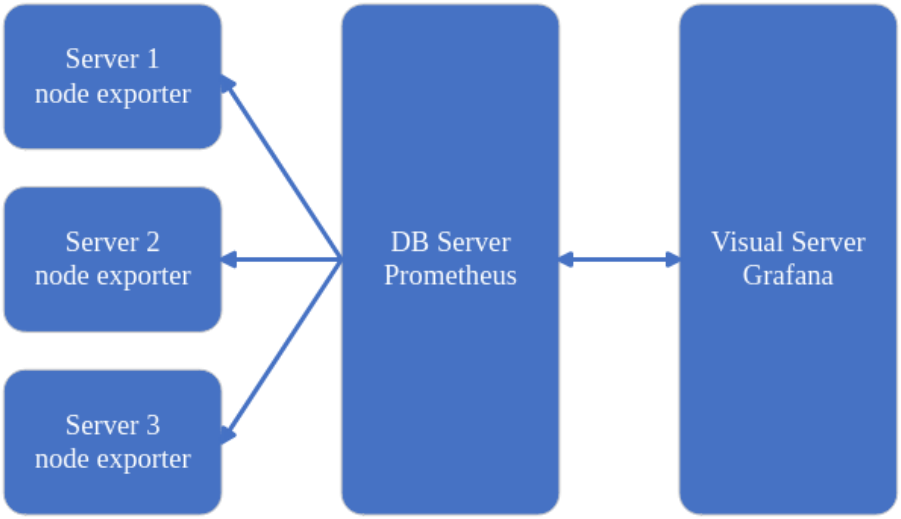


Рис.3.7. Розподілена система моніторингу

Як ми говорили раніше, така схема дозволяє змінювати кожен компонент системи окремо, а також доповнювати цю схему дублюючими елементами. На рис.3.8, наприклад, схема, де Grafana і Zabbix як джерело даних використовує один BD сервер - Prometheus.

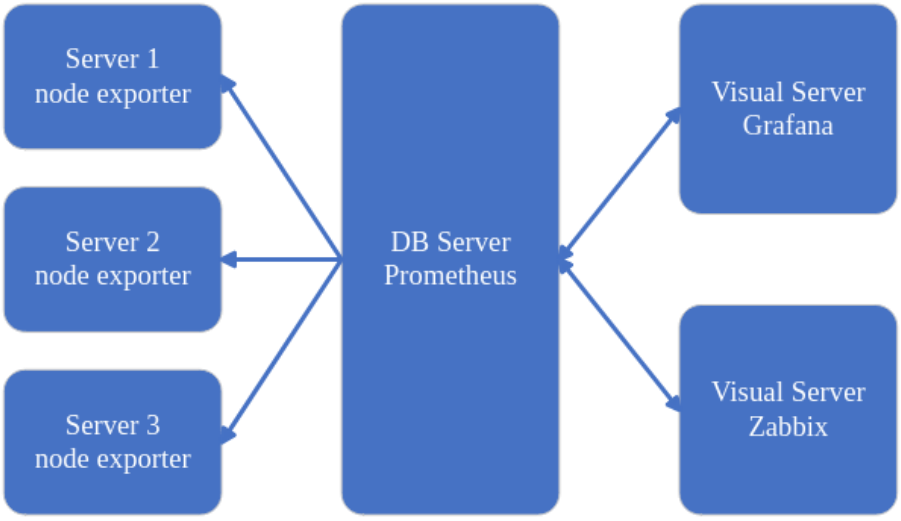


Рис.3.8 Розподілена система моніторингу з двома візуальними серверами

Такий підхід може бути дуже зручним, якщо вам потрібно віддавати різні дані різним людям. Наприклад, така система може віддавати клієнтам дані про ресурси,

що споживаються на серверах, а мережевим адміністраторам ця ж система може надавати дані про мережі і кількість споживаного трафіку.

Одним словом, якщо вам потрібно отримувати дані про навантаження з одного сервера, можна використовувати локальну установку системи моніторингу з Docker, а якщо треба моніторити безліч серверів, то краще застосувати розподілену систему моніторингу.

Система моніторингу за тестами навантаження буде складатися з наступних частин:

- InfluxDB: база даних часових рядів, яка зберігає результати тестування продуктивності.
- Grafana: Інструмент інформаційної панелі, який використовується для візуалізації результатів тестування продуктивності.
- Хронограф: використовується для швидкого перегляду даних, які ви зберегли в InfluxDB

На відміну від системи моніторингу за застосунком де прометеус робить запити на експортери метрик щоб їх отримати, гатлінг сам відправляє метрики в InfluxDB після чого вони виводяться на графіки графаною. Схема роботи показана на рис.3.9.

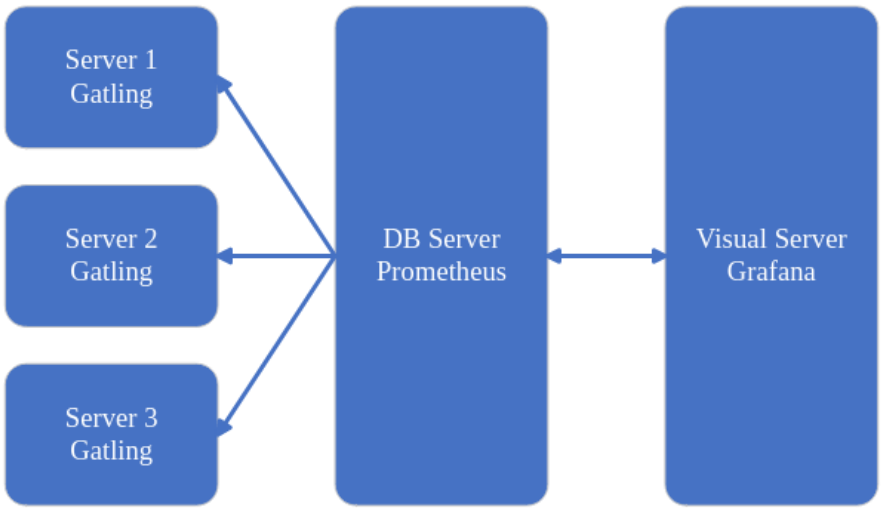


Рис. 3.9. Схема роботи системи моніторингу за Gatling

3.1.2 Архітектура системи навантаження

Наступна частина це фреймворк тестування продуктивності. Для написання якого ми будемо використовувати інструмент Gatling. Gatling має наступні особливості:

- Автономний HTTP-проксі-записувач,
- Сценарії на основі Scala,
- Виразний DSL для розробки тестів, що не потребує пояснень ,
- асинхронний неблокуючий двигун для максимальної продуктивності,
- Відмінна підтримка протоколів HTTP(S), а також може використовуватися для навантажувального тестування JDBC і JMS,
- Перевірки та твердження,
- і вичерпний звіт HTML.

На відміну від JMeter, Gatling Tool не має графічного інтерфейсу . Для редагування симуляції потрібно написати код. Інструмент Gatling явно створений для розробників, хоча вони стверджують, що ним можуть користуватися й не програмісти. Синтаксис симуляції Gatling Scala досить простий для розуміння. На рис.3.10. показано симуляцію з використанням фреймворку Gatling.

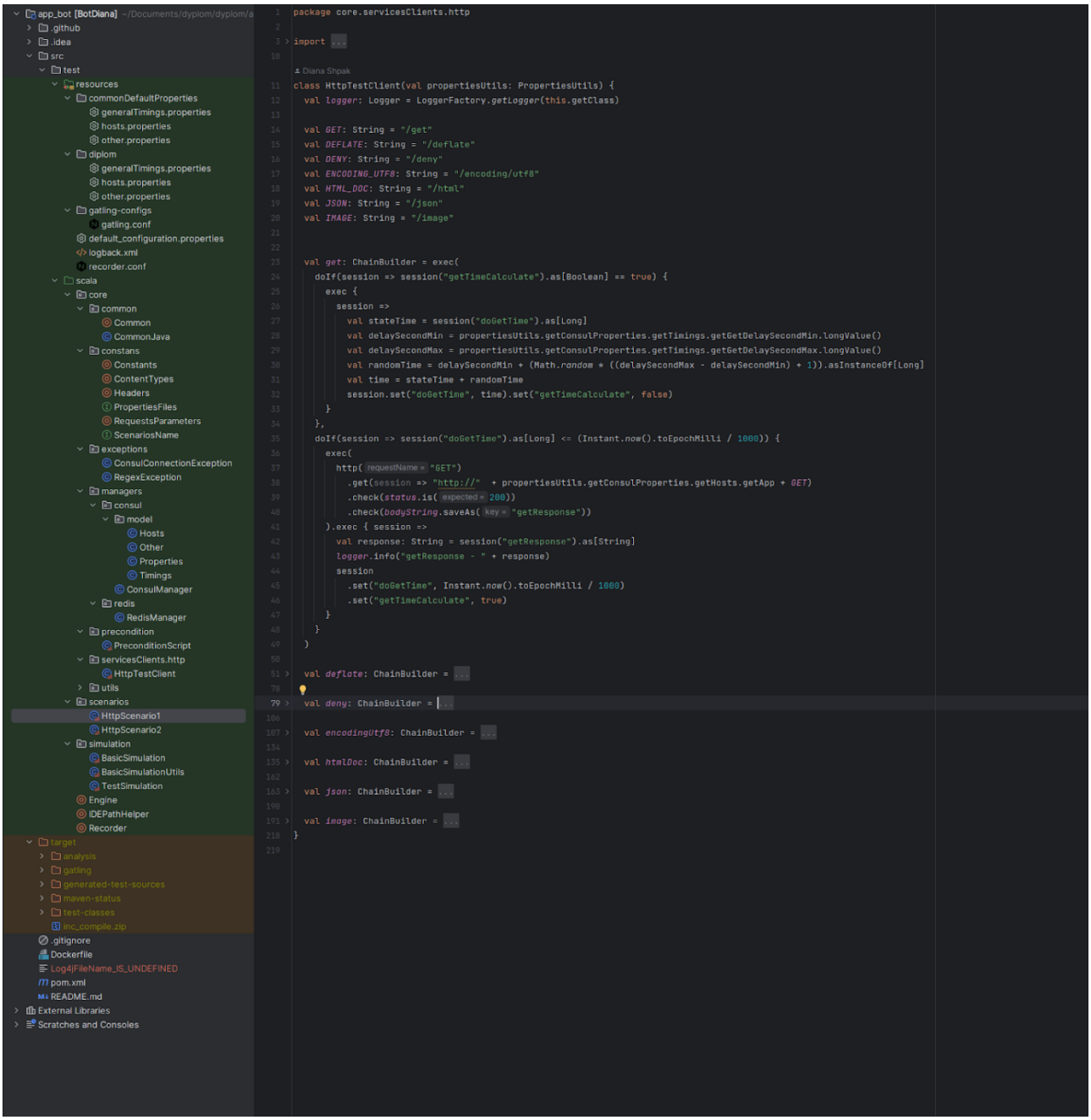


Рис.3.10.Симуляцію з використанням фреймворку Gatling.

Gatling використовує більш сучасний механізм на основі Akka. Akka — це розподілена структура, заснована на моделі актора. Це дозволяє повністю асинхронні обчислення. Актори — це малі сутності, які спілкуються з іншими акторами за допомогою обміну повідомленнями. Він може імітувати кілька віртуальних користувачів за допомогою одного потоку. Gatling також використовує Async HTTP Client.

Хоча Gatling базується на Akka, який є структурою розподіленого актора, Gatling не є розподіленим. Він не може масштабуватися по горизонталі. Ви, звичайно, можете запустити кілька екземплярів Gatling на різних машинах одночасно, і тоді це потрібно буде нам дотатково автоматизувати за допомогою Ansible. Оскільки під час навантажувального тестування HTTP-програми, коли вам потрібно імітувати кілька тисяч користувачів, однієї машини може бути недостатньо, навантаження має бути розподілене між кількома машинами.

Розподілене тестування - це можливість використовувати кілька підлеглих комп'ютерів як генератори навантаження. Ідея полягає в тому, щоб розподілити навантаження на кілька машин, щоб мати можливість симулювати більше одночасних користувачів. Приклад розподіленої системи тестування показано на рис.3.11.

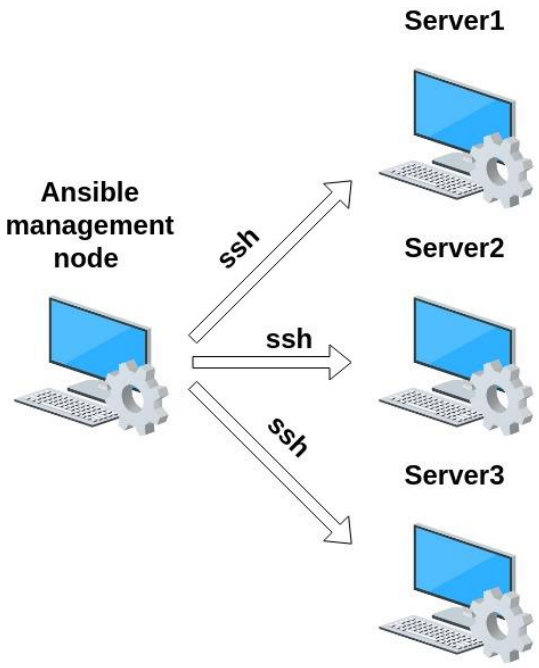


Рис.3.11 Розподілена система тестування

Таким чином ми зможемо зменьшувати чи збільшувати кількість потоків на кожному сервері, а 1 потік емулює роботу 1 реального користувача нашого застосунку. Також ми зможемо змінювати кількість самих серверів якщо нам буде їх недостатньо щоб згенерувати відповідне навантаження.

Так як у нас можливий динамічний скейлінг серверів з тестами навантаження то нам потрібне якесь централізоване рішення для збереження параметрів самих тестів навантаження, для цього дуже добре підходить Consul. Схема кластеру Consul показана на рис.3.12.

Рис.3.12 Кластер консул

Перший сервер для тестів навантаження буде майстер нода консула, всі інші приєднаються до неї як агенти, і таким чином з кожного інстанса по локалхосту ми отримуємо доступ до консула. Відповідно у нас буде централізоване сховище для параметрів самих тестів, які ми також зможемо редагувати під час виконання автотесту. Ці параметри відразу будуть примінятися до всіх серверів, в результаті в самому консулі ми побачимо всі сервіси які знаходяться у кластері, показано на рис.3.13

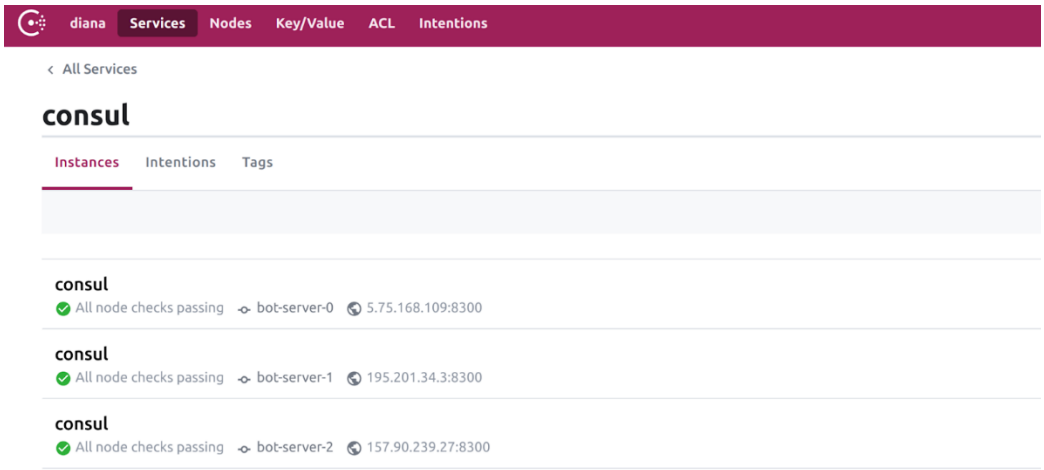


Рис.3.13 Сервіси кластера консул

У нас в консулі зареєстровано 3 сервера, показано на рис.3.14, і відповідно якщо перейдемо на список нод то у маємо 3 ноди, одна з яких з зірочкою - це майстер нода консула.

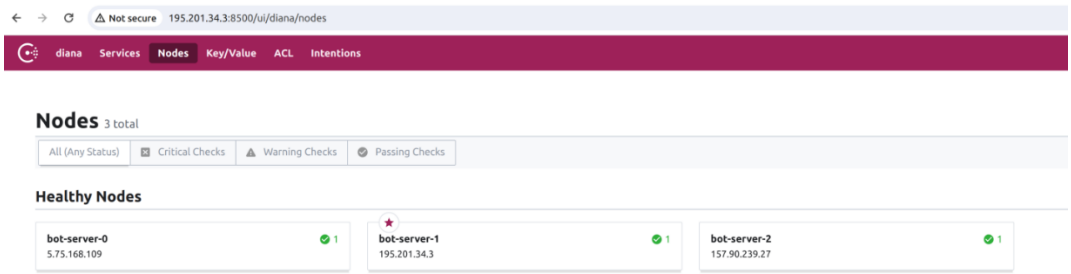


Рис.3.14 Сервіси зареєстровані у консулі

3.2 Розробка системи управління конфігураціями та інфраструктурою

3.2.1 Розробка Infrastructure-as-code

Terraform — це декларативна мова конфігурації інфраструктури, яка дозволяє визначати обчислювальні ресурси, правила брандмауера, облікові записи користувачів і будь-які інші концепції хмарних інфраструктур.

У реєстрі Terraform є офіційний плагін провайдера Hetzner, класифікований як партнер і тому розроблений самим Hetzner. Переглядаючи документацію, ми бачимо, що такими концепціями Hetzner Cloud можна керувати:

- hcloud_firewall: конфігурація брандмауера
- hcloud_floating_ip: Віртуальний IP, незалежний від конкретного екземпляра сервера. Можна використовувати, наприклад, як IP-адресу запису балансувальника навантаження.
- hcloud_load_balancer: реалізація балансувальника навантаження в хмарі Hetzner надає підресурси для network, serviceitarget
- hcloud_managed_certificate: Сертифікати під керуванням Hetzner
- hcloud_network: Налаштуйте мережі як частину інсталяції з допоміжними ресурсами для route,subnet
- hcloud_placement_group: Абстракція, що визначає, як пов’язані ресурси розміщуються в хмарі Terraform
- hcloud_primary_ip: загальнодоступна IP-адреса, яка відображається на конкретному сервері
- hcloud_rdns: Керуйте зворотними записами DNS у хмарі Hetzner
- hcloud_server: конкретний екземпляр сервера, яким слід керувати за допомогою Hetzner.
- hcloud_server_network: приватна мережа, яка з’єднує сервери
- hcloud_snapshot: резервна копія файлової системи сервера
- hcloud_ssh_key: Ключі SSH, які використовуються для доступу до серверів
- hcloud_uploaded_certificate: представляє сертифікати, керовані вручну
- hcloud_volume: обсяги зберігання, які можна приєднати до серверів із attachment допоміжним ресурсом.

Відповідно до офіційної документації ми налаштовуємо провайдера наступним чином, показано на рис.3.15:

```
# Define provider
terraform {
  required_providers {
    hcloud = {
      source = "hetznercloud/hcloud"
    }
  }
}
```

Рис.3.15 Приклад налаштування провайдера

Тепер ми можемо ініціалізувати проект виконавши команду - terraform init, приклад якої показано на рис.3.16

```
Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hetznercloud/hcloud from the dependency lock file
- Installing hetznercloud/hcloud v1.44.1...
- Installed hetznercloud/hcloud v1.44.1 (signed by a HashiCorp partner, key ID 5219EACB3A77198B)

Partner and community providers are signed by their developers.
If you'd like to know more about provider signing, you can read about it here:
https://www.terraform.io/docs/cli/plugins/signing.html

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Рис.3.16. Результат роботи команди terraform init

Ресурс hcloud_ssh_key налаштовано за допомогою відкритого ключа в рядковому форматі. Існує кілька провайдерів Terraform для створення ключів SSH. Я вибрала hashicorp/tls провайдера - він не вимагає налаштувань. Ключовий ресурс SSH створюється з такою конфігурацією, показано на рис.3.17.

```
# Define Hetzner provider token
provider "hcloud" {
  token = var.hcloud_token
}

# Obtain ssh key data
data "hcloud_ssh_key" "ssh_key" {
  fingerprint = "c5:e7:80:f0:ed:07:3d:e3:97:19:fa:30:b0:7f:d5:e6"
}
```

Рис.3.17 Створення ресурсу SSH

Виконуємо налаштування мережі за допомогою ресурсу hcloud_server_network і hcloud_network, створення ресурсу мережа показано на рис.3.18.

```
resource "hcloud_server_network" "grafana_network" {
  count     = var.instances_grafana
  server_id = hcloud_server.grafana[count.index].id
  subnet_id = hcloud_network_subnet.hc_private_subnet_grafana.id
}

resource "hcloud_network_subnet" "hc_private_subnet_grafana" {
  network_id   = hcloud_network.hc_private_grafana.id
  type         = "cloud"
  network_zone = "eu-central"
  ip_range     = var.ip_range_grafana
}

resource "hcloud_network" "hc_private_grafana" {
  name     = "hc_private_grafana"
  ip_range = var.ip_range_grafana
}
```

Рис.3.18 Створення ресурсу мережа

Після сворення ресурсів ми побачимо нашу мережу в списку мереж на Hetzner рис.3.19.

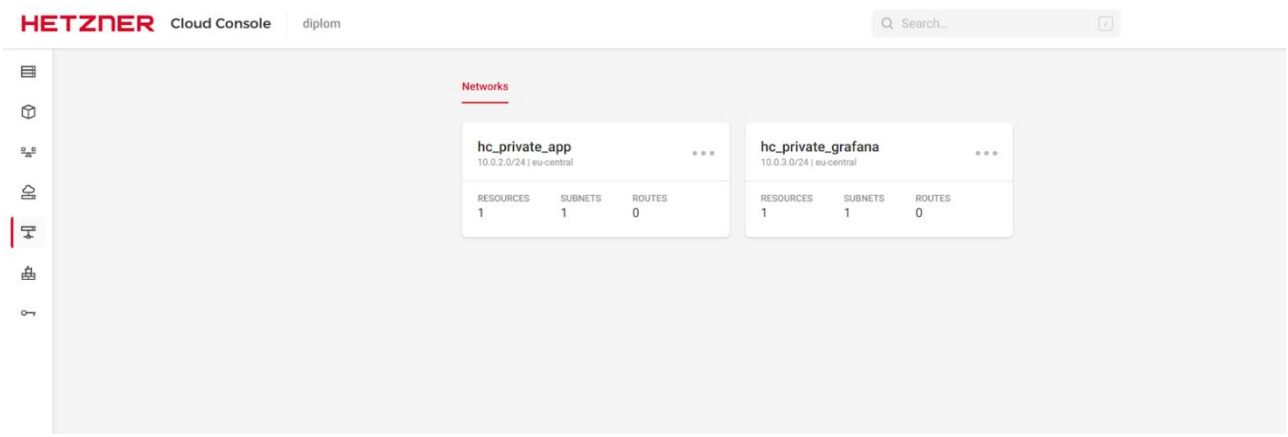


Рис.3.19 Список мереж

Більш детальну інформацію і про сервери які знаходяться у нашій мережі ми можемо побачити натиснувши на цю мережу, так як ми створюємо її для сервера графана. На рис.3.20 ми бачимо що в нашій мережі знаходиться один сервер з відповідною йому IP адресою.

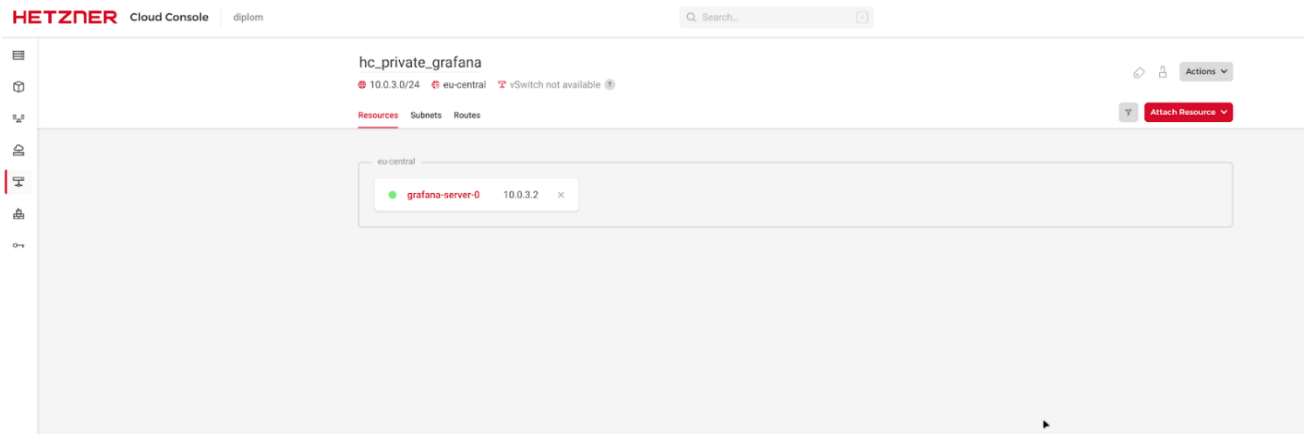


Рис. 3.20 Список серверів в мережі

Далі виконуємо налаштування ресурсу серверу. Ресурс сервера вимагає встановлення кількох атрибутів. Нам потрібно з’ясувати, які образи та типи серверів існують, а також дізнатися їхні конкретні назви, щоб посилатися на них у конфігурації ресурсу Terraform. Приклад отримання всіх доступних образів показано на рис.3.21.

```
curl \
 -H "Authorization: Bearer $TF_HETZNER_TOKEN" \
 'https://api.hetzner.cloud/v1/images'

{
  "images": [
    {
      "id": 3,
      "type": "system",
      "status": "available",
      "name": "centos-7",
  ...
```

Рис.3.21 Приклад отримання всіх образів з клауду

Далі нам потрібно отримати ідентифікатори для серверів, приклад отримання яких показано на рис.3.22

```
curl \
 -H "Authorization: Bearer $TF_HETZNER_TOKEN" \
 'https://api.hetzner.cloud/v1/server_types'

"server_types": [
    {
      "id": 1,
      "name": "cx11",
      "description": "CX11",
      "cores": 1,
      "memory": 2.0,
      "disk": 20,
   ...
```

Рис.3.22 Приклад отримання ідентифікаторів

Ми обрали образ "ubuntu-22.04" і тип сервера cx11. Остаточне hcloud_server визначення ресурсу містить цю інформацію та посилається на hcloud_ssh_key ресурс, який ми створили раніше. Створення ресурсу сервера в тераформ показано на рис.3.23

```
resource "hcloud_server" "grafana" {
  count       = var.instances_grafana
  name        = "grafana-server-${count.index}"
  image       = var.os_type
  server_type = var.server_type
  location    = var.location
  ssh_keys  = ["${data.hcloud_ssh_key.ssh_key.id}"]
  labels = {
    type = "grafana"
  }
  user_data = file("user_data.yml")
}
```

Рис.3.23 Ресурс сервера Grafana

Всі параметри сервера ми винесемо в окремий файл variables.tf, створений файл зі змінними показано на рис.3.24

```
variable "location" {
  default = "nbg1"
}

variable "http_protocol" {
  default = "http"
}

variable "http_port" {
  default = "80"
}


variable "instances_grafana" {
  default = "1"
}

variable "server_type" {
  default = "cx11"
}

variable "os_type" {
  default = "ubuntu-22.04"
}

variable "disk_size" {
  default = "20"
}

variable "ip_range_grafana" {
  default = "10.0.3.0/24"
}
```

Рис.3.24. Файл зі змінними тераформ

Останнім етапом налаштовуємо виведення даних в консоль, а саме айпі адрес і статусів серверів які ми створюємо на Hezner, показано на рис.3.25.

```
output "grafana_servers_status" {
  value = {
    for server in hcloud_server.grafana :
    server.name => server.status
  }
}

output "grafana_servers_ips" {
  value = {
    for server in hcloud_server.grafana :
    server.name => server.ipv4_address
  }
}
```

Рис.3.25 Приклад налаштування виведення даних в консоль

Після застосування сценарію, а саме виконання команди terraform apply маємо отримати створений сервер на Hezner, показано на рис.3.26.

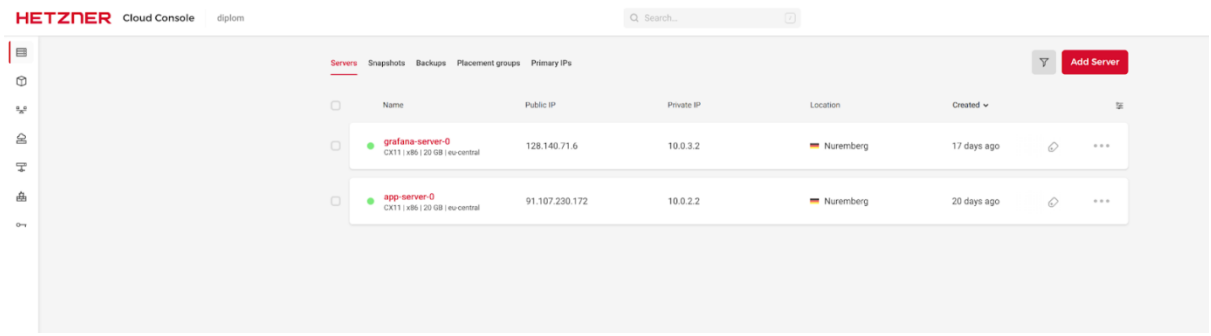


Рис.3.26 Список серверів

3.2.2 Деплой і конфігурація веб додатку

Сервер, який ми підготували, є звичайним сервером Ubuntu без спеціальної конфігурації, тому нам його ще необхідно налаштувати. Саме для цього будемо використовувати Ansible.

Конфігурація інвентаризації Ansible розташована за адресою *inventory/hcloud.yml*

Першою командою яку ми виконаємо буде ansible-inventory --list аби перевірити, чи має Ansible тепер доступ до наших ресурсів Hetzner, і чи правильно читає їхні мітки для формування інвенторі. Якщо у нас сервер створений правильно і з усіма параметрами то ми побачимо його в інвенторі яке показано на рис.3.27.

```
{
    "_meta": {
        "hostvars": {
            "app-server-0": {
                "ansible_host": "{{ labels.ssh_ip | default(ipv4) }}",
                "architecture": "x86",
                "datacenter": "nbg1-dc3",
                "id": "38871367",
                "image_id": "67794396",
                "image_name": "ubuntu-22.04",
                "image_os_flavor": "ubuntu",
                "ipv4": "91.107.230.172",
                "ipv6": "2a01:4f8:1c1e:9c0d::1",
                "ipv6_network": "2a01:4f8:1c1e:9c0d::",
                "ipv6_network_mask": "64",
                "labels": {
                    "type": "http"
                },
                "location": "nbg1",
                "name": "app-server-0",
                "private_networks": [
                    {
                        "id": 3514055,
                        "ip": "10.0.2.2",
                        "name": "hc_private_app"
                    }
                ],
                "server_type": "cx11",
                "status": "running",
                "type": "cx11"
            },
            "grafana-server-0": {
                "ansible_host": "{{ labels.ssh_ip | default(ipv4) }}",
                "architecture": "x86",
                "datacenter": "nbg1-dc3",
                "id": "38984887",
                "image_id": "67794396",
                "image_name": "ubuntu-22.04",
                "image_os_flavor": "ubuntu",
                "ipv4": "128.140.71.6",
                "ipv6": "2a01:4f8:c0c:98c4::1",
                "ipv6_network": "2a01:4f8:c0c:98c4::",
                "ipv6_network_mask": "64",
                "labels": {
                    "type": "grafana"
                },
                "location": "nbg1",
                "name": "grafana-server-0",
                "private_networks": [
                    {
                        "id": 3524889,
                        "ip": "10.0.3.2",
                        "name": "hc_private_grafana"
                    }
                ],
                "server_type": "cx11",
                "status": "running",
                "type": "cx11"
            }
        }
    },
    "all": {
        "children": [
            "ungrouped",
            "hcloud",
            "app_http",
            "app_grafana"
        ]
    },
    "app_grafana": {
        "hosts": [
            "grafana-server-0"
        ]
    },
    "app_http": {
        "hosts": [
            "app-server-0"
        ]
    },
    "hcloud": {
        "hosts": [
            "app-server-0",
            "grafana-server-0"
        ]
    }
}
```


Рис.3.27 Інвенторі

Ми також можемо пінгувати наш сервер за допомогою Ansible запустивши цю команду, щоб переконатися, що з’єднання налаштовано належним чином: - ansible --become-user=root -m ping all -u root. Виконання команди пінг показано на рис.3.28.

```
app-server-0 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
grafana-server-0 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

Рис.3.28. Виконання команди пінг

З’єднання працює і ми готові до налаштування сервера. На нашому сервері потрібно проінсталювати докер клієнт для того щоб ми могли запускати докер контейнери і після цього запустити докер контейнер з нашим застосунком з відповідною конфігурацією. Таким чином наш плейбук буде складатися з 2 ролей, першу проінсталює докер, а другу розгорне наш застосунок. Відповідно даний плейбук показано на рис.3.29

```
---
- hosts:
    - app_http
  user: root
  become: true
  become_method: sudo
  roles:
    - docker_install

- hosts:
    - app_http
  user: root
  become: true
  become_method: sudo
  roles:
    - start_app
```

Рис. 3.29 Плейбук для деплоя застосунку

Роль докер інстал буде мати в своєму складі декілька модулів для встановлення залежностей для докер клієнта, для скачування самого докер клієнта, та для його розгортання. Відповідно роль для інсталяції докеру показана на рис. 3.30:

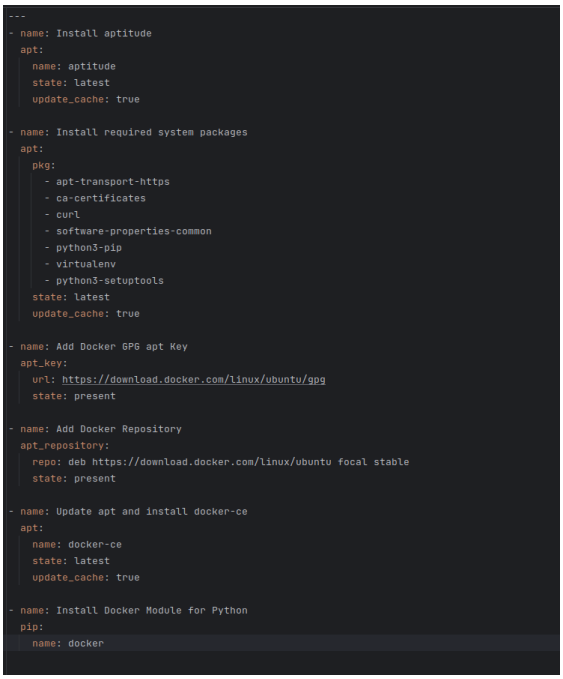


Рис. 3.30 роль для інсталяції докеру

Роль для запуску контейнера з аплікейшином містить один модуль який запускає контейнер на нашому сервері на 8000 порті. Роль для деплоя застосунку показана на рис.3.31.

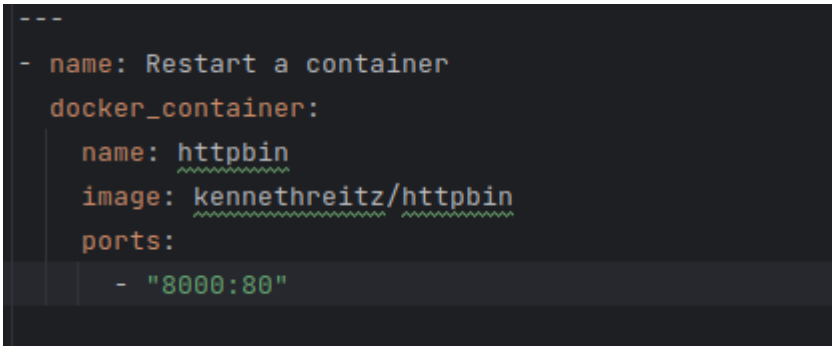


Рис.3.31. Роль для деплоя застосунку

Після того як у нас готові всі плейбуки застосовуємо наші зміни ansible-playbook inventory/hcloud.yml deploy_app.yml. Як тільки все пробіжить то наші

сервери будуть мати потрібну нам конфігурацію і ми зможемо відкрити юай нашого застосунку на нашому сервері який запущений на 8000 порті що показано на рис.3.32.

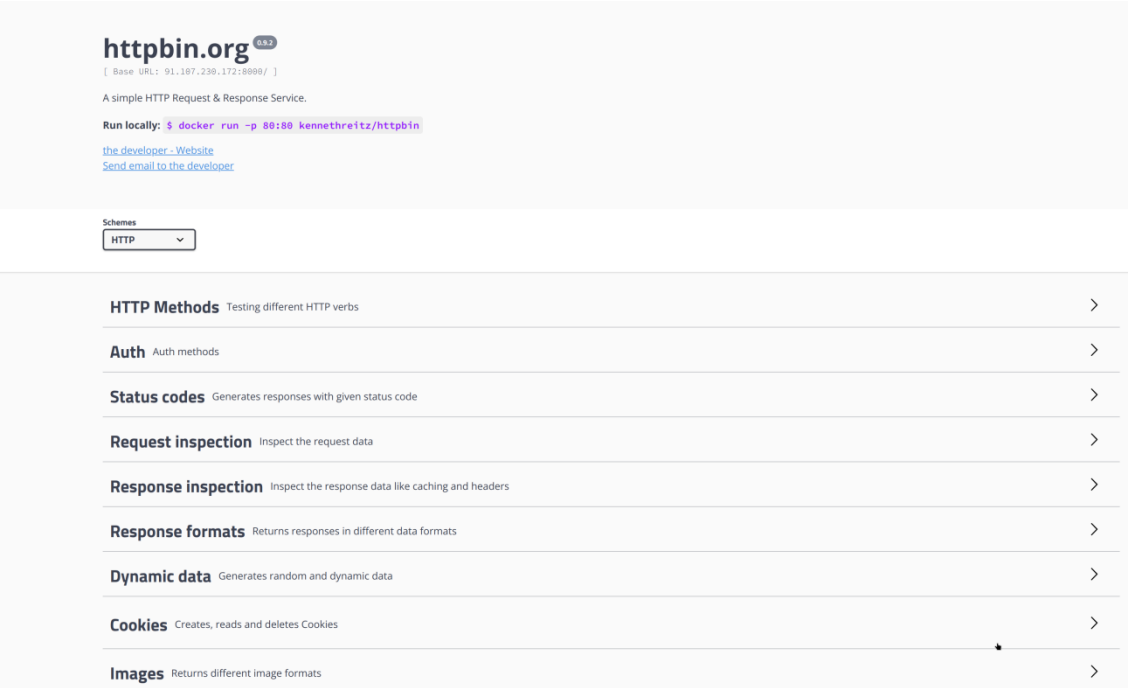


Рис.3.32 Юай застосунку

Якщо наш застосунок запустився, то значить все працює і ми можемо переходити до наступного пункту і налаштовувати систему моніторингу за застосунком.

3.2.3 Деплой і конфігурація моніторингу за веб додатком

Плейбук для розгортання і конфігурації системи моніторингу буде складатися з 5 ролей які будуть запускатися на різних хост групах:

deploy_exporter - дана роль буде розгортати експортер метрик, який збиратиме метрики з сервера на якому розгорнутий застосунок і буде відправляти їх у прометей;

docker_install - дана роль встановлюватиме докер і всі потрібні йому залежності на сервер на якому буде розгорнута графана з базами даних

deploy_influxdb - дана роль запускає контейнер з influxdb на сервері на якому встановлена графана, в даній базі ми будемо зберігати метрики з гатлінгу

deploy_grafana - дана роль запускає контейнер з прометеусом і графаною яка відповідає за відображення метрик з прометеуса і інфлюксдб

configure_grafana_dashboards - дана роль запускається на локалхості, з якого ми запускаємо ансібл плейбуки, налаштовує графану і створює відповідні дашбоарди для відображення наших метрик.

Система моніторингу за застосунком буде складатися з наступних частин: графана, прометеус і експортер метрик, що є показано на рис.3.33.

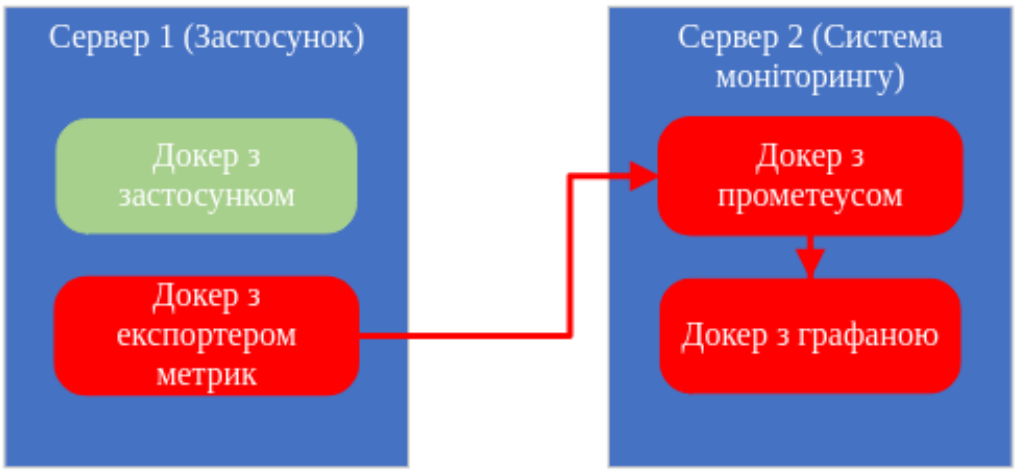


Рис.3.33 Система моніторингу за застосунком

Плейбук для розгортання системи моніторингу за застосунком, в якому застосовуються всі вище перераховані ролі показано на рис.3.34.

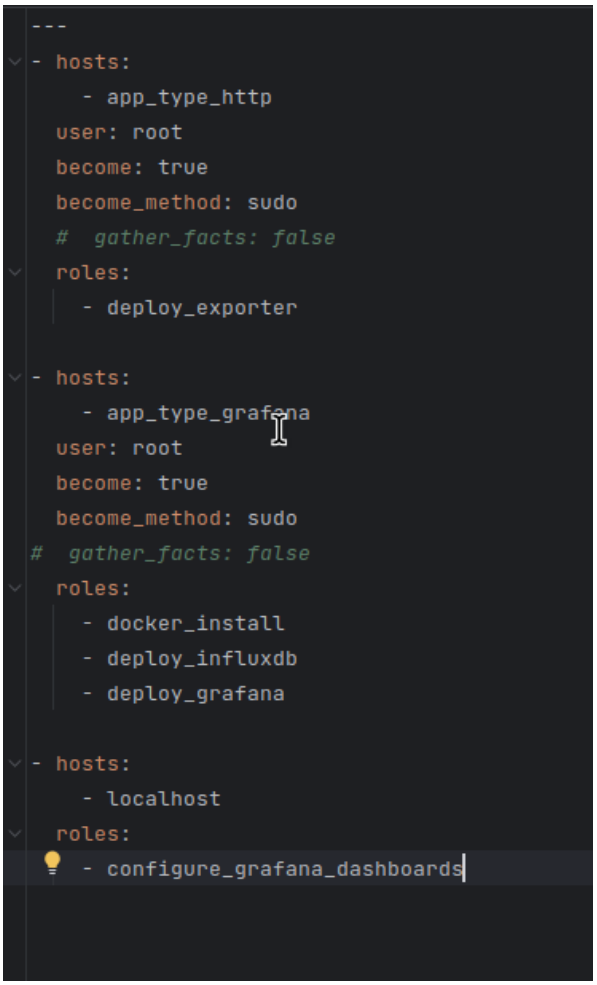


Рис.3.34. Плейбук для розгортання системи моніторингу

Розглянемо кожну роль окремо, на сервер з застосунком нам вже не потрібно інсталювати докер так як ми це зробили перед запуском контейнера з застосунком, а значить відразу можемо перейти до запуску контейнера з експортером метрик. Роль deploy_exporter показана на рис.3.35, який буде запускатися на віртуальній машині з нашим застосунком і потрібен нам в свою чергу для збору метрик з даного сервера і відправки їх в прометеус.

Рис.3.35 Роль deploy_exporter

Як ми бачимо у нас запускається докер контейнер з імеджа prom/node-exporter на 9100 порті з наступними вольюмами:

- /proc:/host/proc:ro

- /sys:/host/sys:ro

- /:/rootfs:ro

і наступним списком команд:

"--path.procfs=/host/proc",

"--path.sysfs=/host/sys",

"--collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc|rootfs/var/lib/docker/containers|rootfs/var/lib/docker/overlay2|rootfs/run/docker/netns|rootfs/var/lib/docker/aufs)($$|/)"

Даний node_exporter застосовується для моніторингу хост-системи. Так як ми використовуємо докер то потрібно використовувати деякі додаткові позначки щоб дозволити доступ node_exporter до простору імен хоста.

Для цього ми вказуємо path.rootfs аргумент. Цей аргумент має відповідати шляху в монтуванні зв’язування кореневого вузла. Node_exporter використовуватиме path.rootfs як префікс для доступу до файлової системи хоста.

Існує різна підтримка колекціонерів у кожній операційній системі. У таблицях нижче перераховано всі існуючі колектори та підтримувані системи. Колекціонери вмикаються за допомогою --collector.<name>прапора. Колекціонери, увімкнені за замовчуванням, можна вимкнути, вказавши прапорець --no-collector.<name>. Щоб увімкнути лише деякі певні колекціонери, використовуйте --collector.disable-defaults --collector.<name> …

Кілька колекціонерів можна налаштувати для включення або виключення певних шаблонів за допомогою спеціальних прапорців. Прапорці виключення використовуються для позначення «всіх, крім», тоді як прапорці включення використовуються, щоб сказати «жодного, крім». Зверніть увагу, що ці позначки є взаємовиключними для колекціонерів, які підтримують обидва.

Ми маємо наступне налаштування колекціонерів:

"--collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc|rootfs/var/lib/docker/containers|rootfs/var/lib/docker/overlay2|rootfs/run/docker/netns|rootfs/var/lib/docker/aufs)($$|/)"

Повний список колекціонерів відображено у таблиці 3.1

Таблиця 3.1. Список колекціонерів

Collector	**Scope**	**Include Flag**	**Exclude Flag**
arp	device	--collector.arp.device-include	--collector.arp.device-exclude
cpu	bugs	--collector.cpu.info.bugs-include	N/A
cpu	flags	--collector.cpu.info.flags-include	N/A
diskstats	device	--collector.diskstats.device-include	--collector.diskstats.device-exclude
ethtool	device	--collector.ethtool.device-include	--collector.ethtool.device-exclude
ethtool	metrics	--collector.ethtool.metrics-include	N/A
filesystem	fs-types	N/A	--collector.filesystem.fs-types-exclude

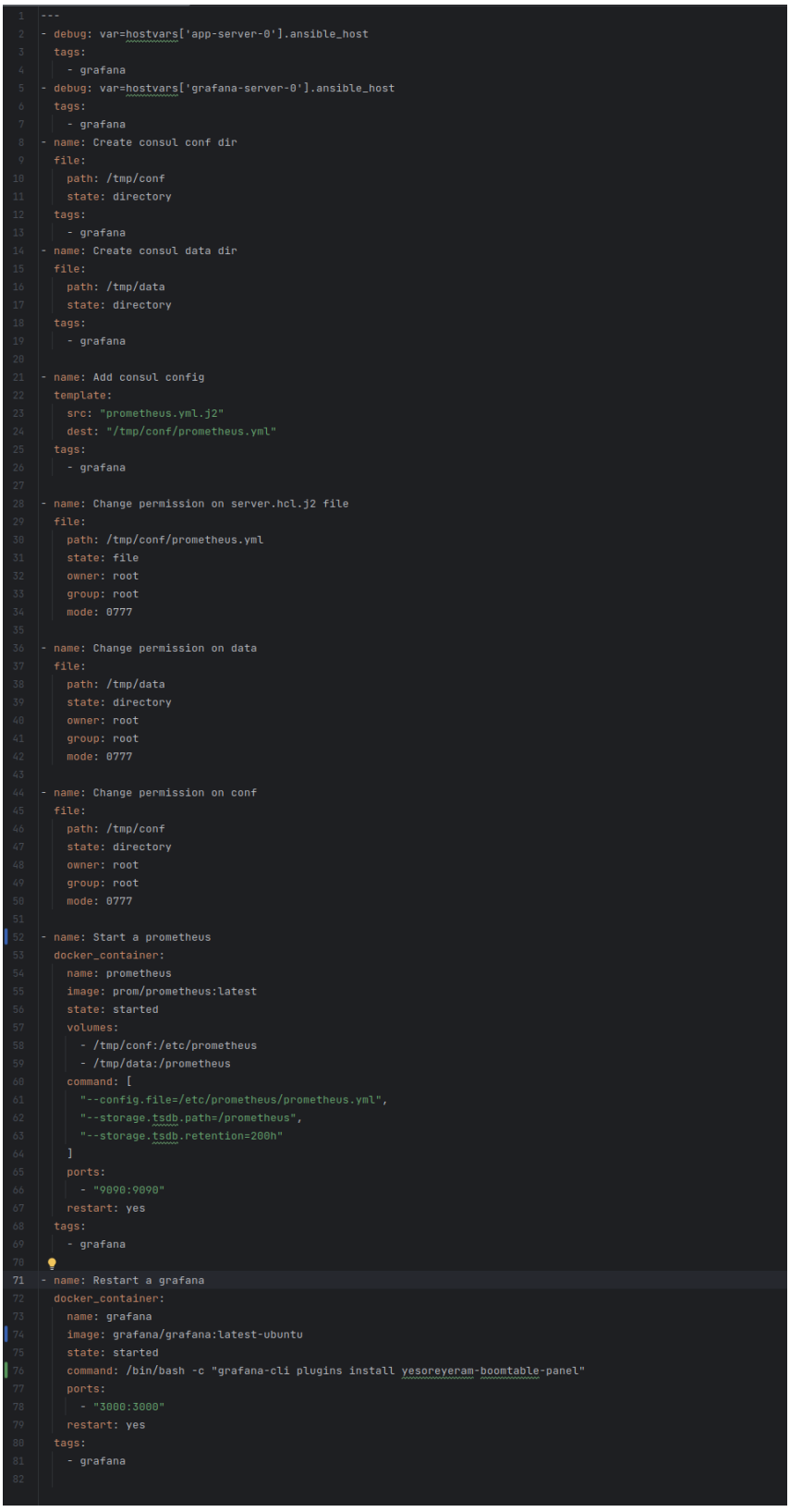
Collector	Scope	Include Flag	Exclude Flag
filesystem	mount-points	N/A	--collector.filesystem.mount-points-exclude
hwmon	chip	--collector.hwmon.chip-include	--collector.hwmon.chip-exclude
netdev	device	--collector.netdev.device-include	--collector.netdev.device-exclude
qdisk	device	--collector.qdisk.device-include	--collector.qdisk.device-exclude
sysctl	all	--collector.sysctl.include	N/A
systemd	unit	--collector.systemd.unit-include	--collector.systemd.unit-exclude

Після того як ми встановили експортер метрик на сервер з застосунком, можемо перейти до запуску контейнера з прометеусом. Але перед цим нам потрібно проінсталювати на даний сервер докер клієнт. Це у нас виконує роль docker_install, що є показана на рис.3.36. Команда буде мати в своєму складі декілька модулів: для встановлення залежностей для докер клієнта, для скачування самого докер клієнта і для його розгортання.

```yaml
---
- name: Install aptitude
  apt:
    name: aptitude
    state: latest
    update_cache: true

- name: Install required system packages
  apt:
    pkg:
      - apt-transport-https
      - ca-certificates
      - curl
      - software-properties-common
      - python3-pip
      - virtualenv
      - python3-setuptools
    state: latest
    update_cache: true

- name: Add Docker GPG apt Key
  apt_key:
    url: https://download.docker.com/linux/ubuntu/gpg
    state: present

- name: Add Docker Repository
  apt_repository:
    repo: deb https://download.docker.com/linux/ubuntu focal stable
    state: present

- name: Update apt and install docker-ce
  apt:
    name: docker-ce
    state: latest
    update_cache: true

- name: Install Docker Module for Python
  pip:
    name: docker
```

Рис.3.36 Роль docker_install

Після того як ми встановили докер клієнт переходимо до запуску контейнера з прометеусом і графаною, прометеус буде сам збирати метрики з експортера метрик по відповідному ендпоінту. Прометей і графана у нас запускаються роллю deploy_grafana, що є показано на рис.3.37.



```yaml
---
- debug: var=hostvars['app-server-0'].ansible_host
  tags:
    - grafana
- debug: var=hostvars['grafana-server-0'].ansible_host
  tags:
    - grafana
- name: Create consul conf dir
  file:
    path: /tmp/conf
    state: directory
  tags:
    - grafana
- name: Create consul data dir
  file:
    path: /tmp/data
    state: directory
  tags:
    - grafana

- name: Add consul config
  template:
    src: "prometheus.yml.j2"
    dest: "/tmp/conf/prometheus.yml"
  tags:
    - grafana

- name: Change permission on server.hcl.j2 file
  file:
    path: /tmp/conf/prometheus.yml
    state: file
    owner: root
    group: root
    mode: 0777

- name: Change permission on data
  file:
    path: /tmp/data
    state: directory
    owner: root
    group: root
    mode: 0777

- name: Change permission on conf
  file:
    path: /tmp/conf
    state: directory
    owner: root
    group: root
    mode: 0777

- name: Start a prometheus
  docker_container:
    name: prometheus
    image: prom/prometheus:latest
    state: started
    volumes:
      - /tmp/conf:/etc/prometheus
      - /tmp/data:/prometheus
    command: [
      "--config.file=/etc/prometheus/prometheus.yml",
      "--storage.tsdb.path=/prometheus",
      "--storage.tsdb.retention=200h"
    ]
    ports:
      - "9090:9090"
    restart: yes
  tags:
    - grafana

- name: Restart a grafana
  docker_container:
    name: grafana
    image: grafana/grafana:latest-ubuntu
    state: started
    command: /bin/bash -c "grafana-cli plugins install yesoreyeram-boomtable-panel"
    ports:
      - "3000:3000"
    restart: yes
  tags:
    - grafana
```

Рис.3.37 Роль deploy_grafana

Як бачимо в даній ролі спочатку створюються директорії, потім встановлюються для них права, після чого ми можемо їх примайнтити у вольюмі до

контейнера з прометеусом. Також нам потрібно буде згенерити конфігураційний файл для мрометеуса, який в себе буде включати всі айпі адреси серверів з аплікейшином та айпі адресу з графаною. Даний конфіг динамічний і згенерується джинджою, що є показано на рис.3.38.

```
scrape_configs:
{% for h in groups['app_type_grafana'] %}
  - job_name: '{{h}}'
    scrape_interval: 5s
    static_configs:
      - targets: ['{{hostvars[h].ansible_host}}:9100']
{% endfor %}
{% for h in groups['app_type_http'] %}
  - job_name: '{{h}}'
    scrape_interval: 5s
    static_configs:
      - targets: ['{{hostvars[h].ansible_host}}:9100']
{% endfor %}
```

Рис.3.38 Конфігураційний файл для прометеуса

Після того як всі каталоги і конфігураційний файл створені з відповідними правами, можемо переходити до запуску прометея і графани.

При старті контейнера з прометеусом нам потрібно прокинути наступні вольюми:

volumes:

- /tmp/conf:/etc/prometheus

- /tmp/data:/prometheus

Перший з конфігураційним файлом який ми згенерували раніше, інший для збереження даних прометеуса.

Також нам потрібно виконати запуск контейнера з наступним списком прапорців:

command:[

"--config.file=/etc/prometheus/prometheus.yml",

"--storage.tsdb.path=/prometheus",

"--storage.tsdb.retention=200h"]

Тут ми налаштовуємо шлях до конфігураційного файлу, шлях до каталогу в якому прометеус буде зберігати свої дані, і час протягом якого вони будуть зберігатися.

Далі щоб була можливість доступу до прометеуса нам потрібно буде прокинути порти в контейнер:

ports:

- "9090:9090"

На цьому конфігурація прометеуса завершена, далі у нас йде модуль для запуску контейнера з графаною в якому нам потрібно лише прокинути порт в контейнер. Для налаштування графани і дашбоардів у нас є окрема роль, тут нам потрібно лише запустити контейнер.

Після того як ми запустили дані ролі ми можемо відкрити стартову сторінку прометеуса, див рис.3.39, за наступним посиланням http://128.140.71.6:9090/, і графани, див рис.3.40, за посиланням http://128.140.71.6:3000/. Оскільки сервер у нас один то в даному випадку зміняться тільки порти.

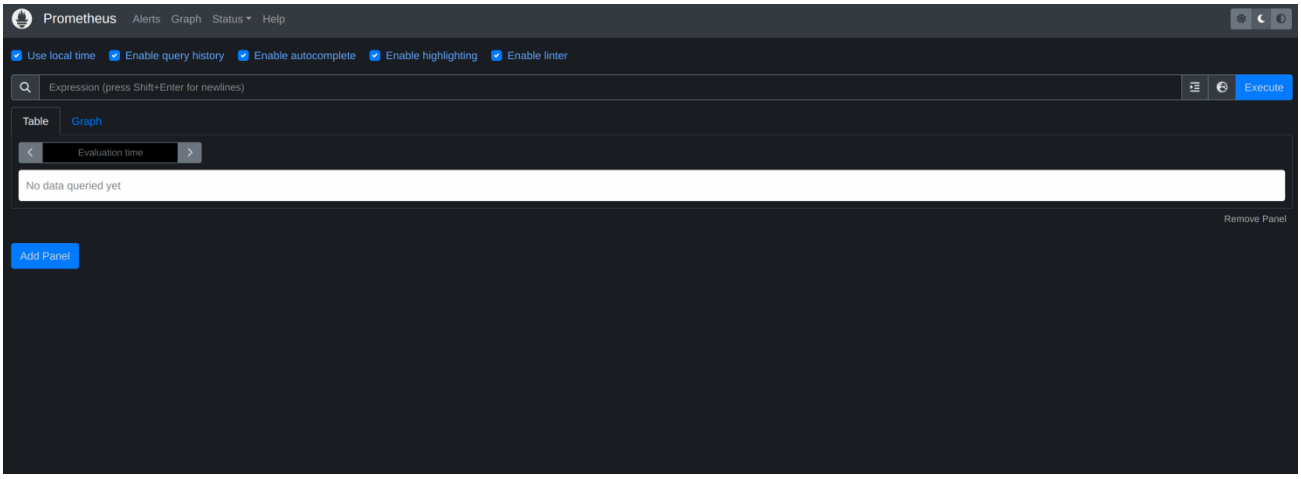


Рис.3.39 Стартова сторінка прометеуса

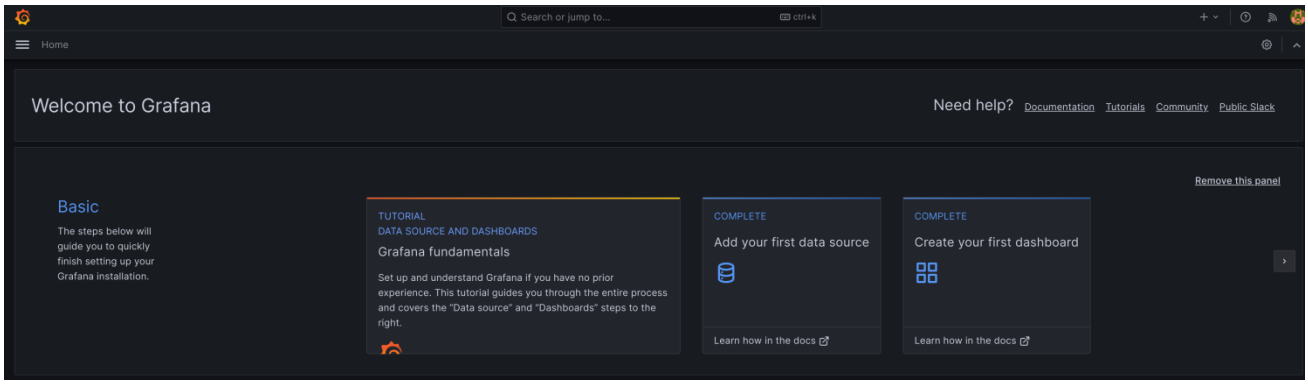


Рис.3.40 Стартова сторінка графани

Якщо у нас відкрились стартові сторінки значить ми все зробили вірно і можна переходити до налаштування системи моніторингу за тестами навантаження.

3.2.4 Деплой і конфігурація моніторингу за тестами навантаження

Система моніторингу за тестами навантаження складається з наступних елементів, показано на рис.3.41.:

- Графана - буде відображати наші метрики
- Інфлюксдб - база в якій будуть зберігатися наші метрики
- Хронографа - застосунок який знадобиться для тестування метрик які ми зберігаємо в інфлюксдб
- Тестів навантаження

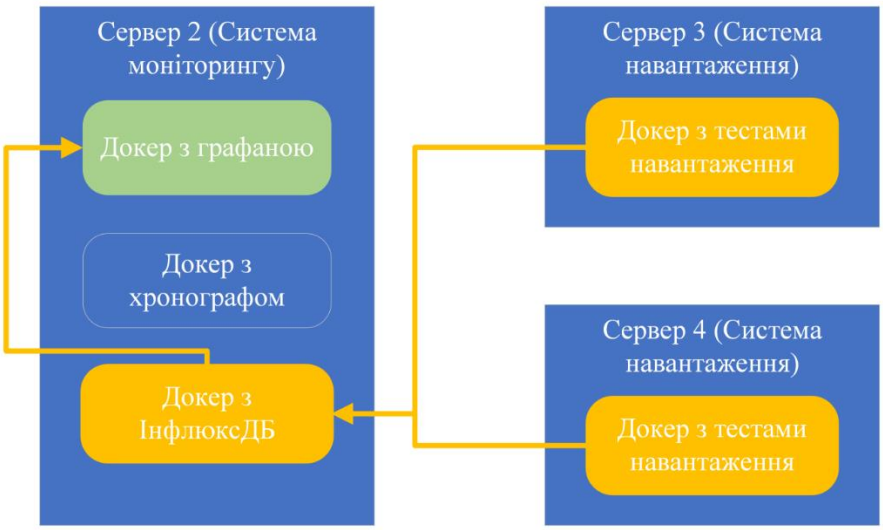
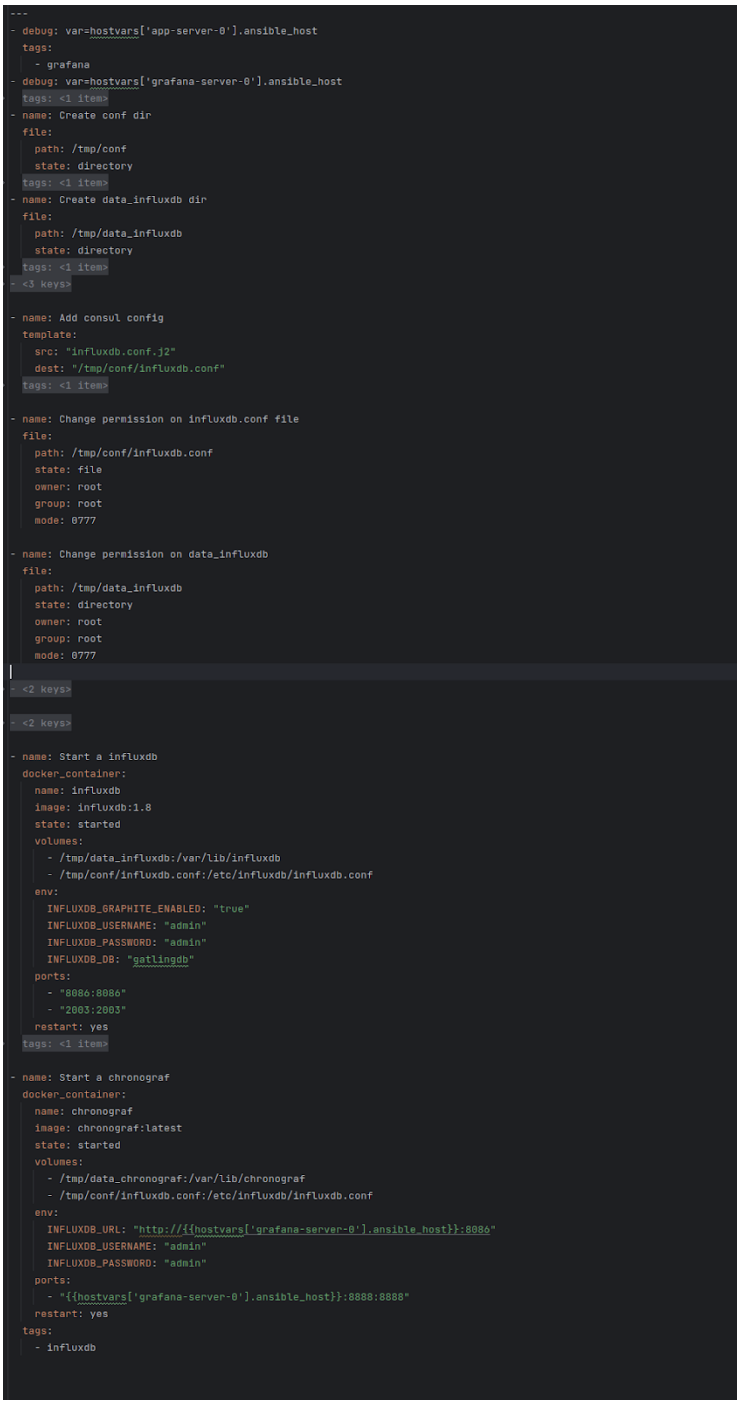


Рис.3.41 Система моніторингу за тестами навантаження

Так як у нас гатлінг сам відправляє метрики в інфлюкдб, то нам потрібно тут налаштувати тільки базу і протестувати самі метрики і переконатися що вони приходять з усіх персентилів які ми вказали у конфігураційному файлі гатлінка.

Щоб налаштувати базу нам потрібно додати наступну роль - deploy_influxdb, яка показана на рис.3.42. На рисунку виконується створення конфігураційних файлів для influxdb, підняття контейнера з influxdb з відповідною конфігурацією, і підняття контейнера chronograf, за допомогою якого ми можемо протестувати які метрики відправляються експортером в інфлюкс influxdb. Також в даній ролі є модулі для сворення потрібних нам каталогів, конфігураційних файлів та надання їм відповідних прав доступу. Після того як ми їх створили на сервері ми можемо примаунтити ці файли і каналоги до контейнера з influxdb



```
---
- debug: var=hostvars['app-server-0'].ansible_host
  tags:
    - grafana
- debug: var=hostvars['grafana-server-0'].ansible_host
  tags: <1 item>
- name: Create conf dir
  file:
    path: /tmp/conf
    state: directory
  tags: <1 item>
- name: Create data_influxdb dir
  file:
    path: /tmp/data_influxdb
    state: directory
  tags: <1 item>
- <3 keys>

- name: Add consul config
  template:
    src: "influxdb.conf.j2"
    dest: "/tmp/conf/influxdb.conf"
  tags: <1 item>

- name: Change permission on influxdb.conf file
  file:
    path: /tmp/conf/influxdb.conf
    state: file
    owner: root
    group: root
    mode: 0777

- name: Change permission on data_influxdb
  file:
    path: /tmp/data_influxdb
    state: directory
    owner: root
    group: root
    mode: 0777

- <2 keys>

- <2 keys>

- name: Start a influxdb
  docker_container:
    name: influxdb
    image: influxdb:1.8
    state: started
    volumes:
      - /tmp/data_influxdb:/var/lib/influxdb
      - /tmp/conf/influxdb.conf:/etc/influxdb/influxdb.conf
    env:
      INFLUXDB_GRAPHITE_ENABLED: "true"
      INFLUXDB_USERNAME: "admin"
      INFLUXDB_PASSWORD: "admin"
      INFLUXDB_DB: "gatlingdb"
    ports:
      - "8086:8086"
      - "2003:2003"
    restart: yes
  tags: <1 item>

- name: Start a chronograf
  docker_container:
    name: chronograf
    image: chronograf:latest
    state: started
    volumes:
      - /tmp/data_chronograf:/var/lib/chronograf
      - /tmp/conf/influxdb.conf:/etc/influxdb/influxdb.conf
    env:
      INFLUXDB_URL: "http://{{hostvars['grafana-server-0'].ansible_host}}:8086"
      INFLUXDB_USERNAME: "admin"
      INFLUXDB_PASSWORD: "admin"
    ports:
      - "{{hostvars['grafana-server-0'].ansible_host}}:8888:8888"
    restart: yes
  tags:
    - influxdb
```

Рис.3.42 Роль deploy_influxdb

В модулі який запускає контейнер з ІнфлюксДБ нам потрібно вказати наступний список вольюмів і змінних середовища з назвою бази, юзером і паролем:

volumes:

- /tmp/data_influxdb:/var/lib/influxdb

- /tmp/conf/influxdb.conf:/etc/influxdb/influxdb.conf

env:

INFLUXDB_GRAPHITE_ENABLED: "true"

INFLUXDB_USERNAME: "admin"

INFLUXDB_PASSWORD: "admin"

INFLUXDB_DB: "gatlingdb"

Також нам потрібно прокинути наступний список портів:

ports:

- "8086:8086"

- "2003:2003"

По цих портах і за допомогою цих кредів, наші експортери і гатлінг зможуть мати доступ до бази данних і відправляти в неї метрики.

Також нам потрідно буде встановити Chronograf - це окремий web-додаток, який ми можемо розгорнути для дослідження тимчасових рядів та тестування наших метрик в графана. Також Chronograf дозволяє швидко переглядати дані, які ми зберегли в InfluxDB, що дозволить нам створювати надійні запити та сповіщення. Він простий у використанні та містить шаблони та бібліотеки, які дозволяють швидко створювати інформаційні панелі з візуалізацією ваших даних у реальному часі.

Для налаштування хронографу нам потрібно прокинути 2 вольюма в контейнер. Перший - це паку в якій хронограф буде зберігати свої дані, і другий з конфігом ІнфлюксДБ. Окрім цього ми маємо передати параметри для підключення до ІнфлюксДБ, а саме: юрл адресу, логін і пароль. Всі ці параметри повинні бути збережені у змінних середовища контейнера з хронографом:

volumes:

- /tmp/data_chronograf:/var/lib/chronograf

- /tmp/conf/influxdb.conf:/etc/influxdb/influxdb.conf

env:

INFLUXDB_URL: "http://{{hostvars['grafana-server-0'].ansible_host}}:8086"

INFLUXDB_USERNAME: "admin"

INFLUXDB_PASSWORD: "admin"

Також нам потрібно прокинути порт в контейнері з хронографом, що дасть нам можливість відкрити юай:

ports:

- "{{hostvars['grafana-server-0'].ansible_host}}:8888:8888"

Після чого ми можемо відкрити за даним портом сам хронограф http://128.140.71.6:8888/ і перевірити дані які ми зберігаємо в ІнфлюксДБ. Показано на рис.3.43.

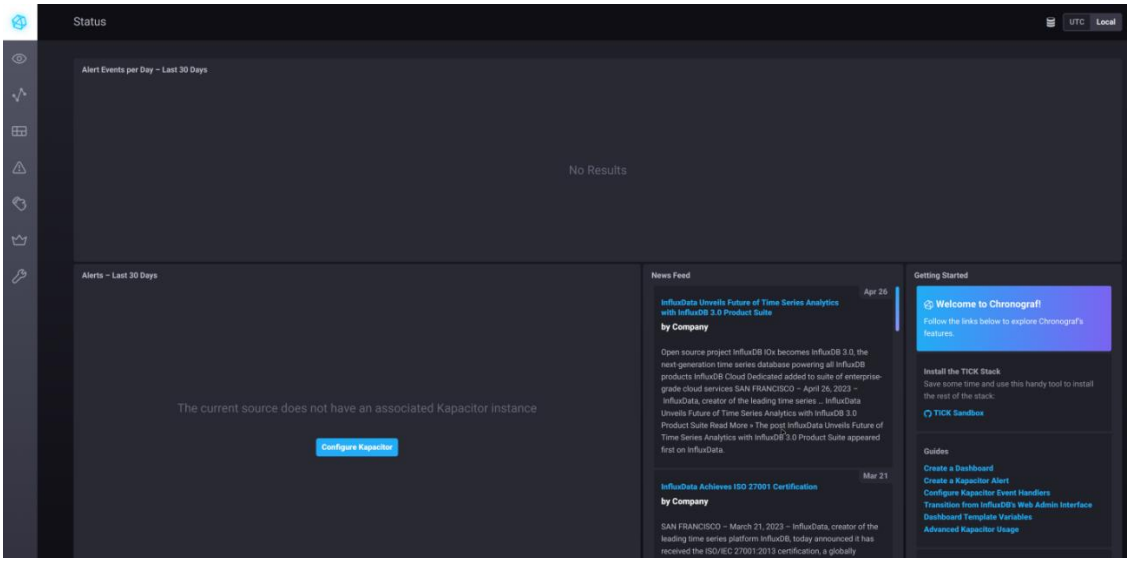


Рис.3.43 Головний дашбоард хронографа

Далі перейдемо на вкладку досліджень для перевірки наших метрик. Виберемо базу гатлінгДб, після цього потрібно обрати ‘gatling’ тег, який ми вказуємо в налаштуваннях гатлінга, і відповідні поля по яким нам потрібно буде будувати наші метрики. Показано на рис.3.44. Перш за все перевіримо чи виводяться метрики по персентилях.

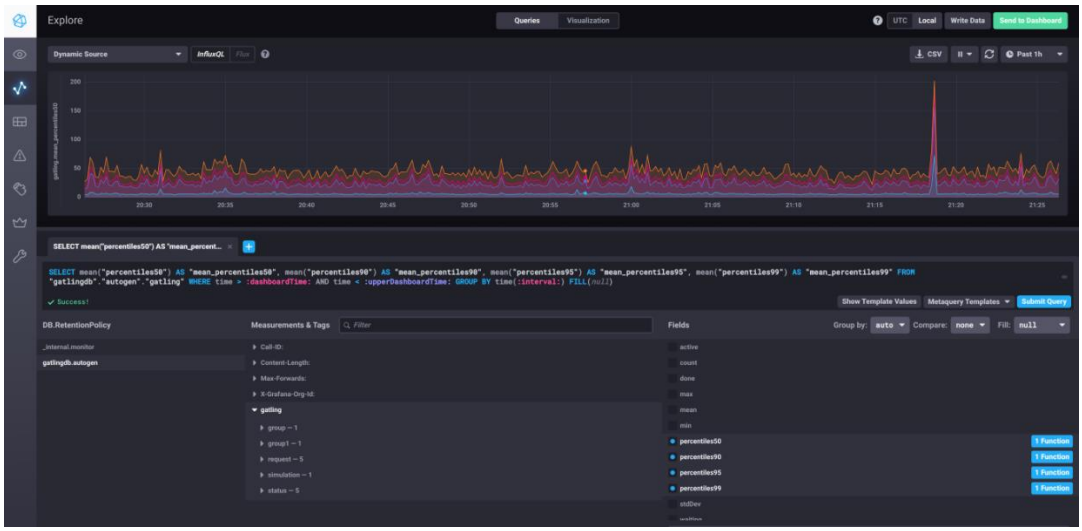


Рис.3.44 Тестування метрик хронографом

Якщо у нас з’явилися метрики і по ним будуються графіки, значить конфігурацію ІнфлюксДБ і гатлінга ми виконали правильно, і далі можна переходити до написання тестів навантаження.

3.2.5 Деплой і конфігурація тестів навантаження

Збільшувати навантаження на наш застосунок ми можемо двома способами. Перший це збільшувати кількість потоків на одному сервері на якому у нас запущені тести, але таким чином ми рано чи пізно досягнемо ліміту по ресурсах сервера які витримують якусь можливу максимальну кількість потоків, і другим способом - збільшити кількість серверів на яких ми запускаємо тести навантаження. Тому нам потрібно передбачити два способи підвищення навантаження. Перший реалізується в самому гатлінгу, а другий за допомогою тераформа. Також нам потрібно якесь централізоване сховище для параметрів тестів навантаження. Для цього ми можемо використати консул. На першому сервері який ми запустимо буде запускатися майстер нода консула, всі інші приєднаються до майстер ноди як консул агенти. Таким чином у нас буде кластер консула для всіх серверів з тестами навантаження, що дасть нам змогу обмінюватися даними між усіма серверами.

Коли ми розробляли нашу інфраструкруру як код ми заздалегіть передбачили можливість збільшення кількості інстансів і винесли це в файл зі змінними для тераформа. Для всіх наших серверів буде використовуватися один і той же скрипт,

тільки на серверах будуть ставитися різні мітки, по яких будуть формуватися свої хост групи. Схема роботи тестів навантаження показана на рис.3.45.

Рис.3.45 Схема роботи тестів навантаження

Після того яки ми підняли наші сервери, виконаємо команду ansible-inventory -i inventory/hcloud.yml --graph, яка нам виведе граф всіх доступних серверів і хост груп рис.3.46.

```
@all:
  |--@ungrouped:
  |--@hcloud:
  |  |--app-server-0
  |  |--grafana-server-0
  |  |--bot-server-2
  |  |--bot-server-0
  |  |--bot-server-1
  |--@host_http:
  |  |--app-server-0
  |--@host_grafana:
  |  |--grafana-server-0
  |--@host_bot:
  |  |--bot-server-2
  |  |--bot-server-0
  |  |--bot-server-1
```

Рис.3.46 Граф інвенторі

Як бачимо зявилась нова група серверів host_bot в яку у нас входять 3 сервери. Далі можемо переходити до написання плейбуків, які нам потрібні для розгортання тестів навантаження.

Для цього нам необхідно мати три плейбука:

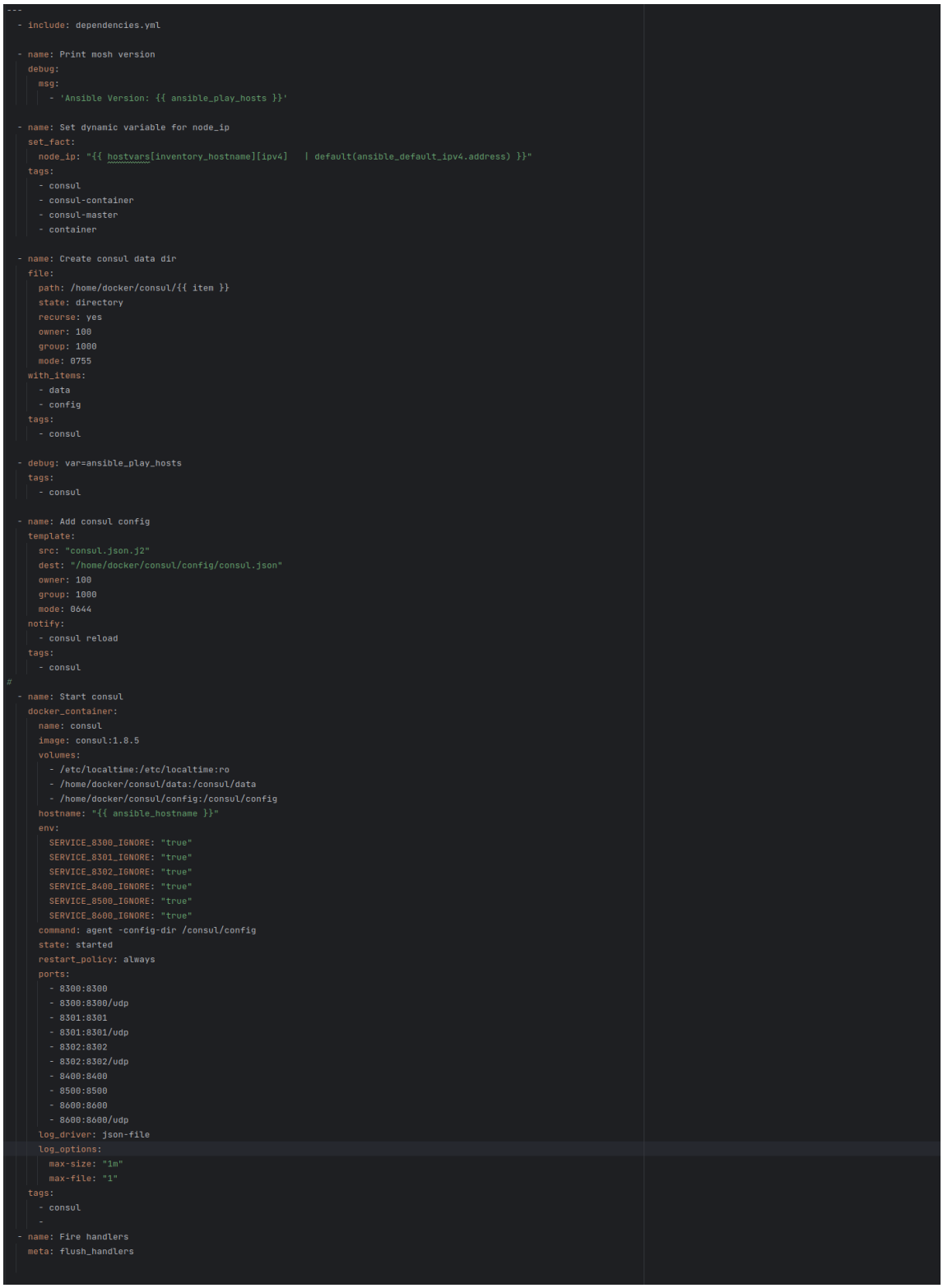
- Precondition.yml - виконує інсталяцію докера і запускає контейнер консула, після чого виконує його налаштування
- bot_start.yml - виконує запуск контейнера з тестами навантаження з відповідними параметрами
- bot_stop.yml - виконує зупинку контейнера з тестами навантаження
- Розглянемо плейбук Precondition, показано на рис.3.47.

```
---
- hosts:
    - host_bot
  user: root
  become: true
  become_method: sudo
#  gather_facts: false
  roles:
    - docker_install
    - consul
```

Рис.3.47 Плейбук Precondition

Даний плейбук запускається тільки на серверах з тестами навантаження. Роль docker_install, як і в попередніх випадках, виконує інсталяцію докера і всіх його залежностей. Це нам потрібно для того щоб запустити контейнер з консулом і тестами навантаження.

Роль consul, показану на рис.3.48, виконує створення конфіга для консула, який генерується динамічно для кожного сервера. Для першого це конфіг майстер ноди, а для всіх наступних це конфіг консул агентів, які підключаються до майстер ноди.



```
---
- include: dependencies.yml

- name: Print mosh version
  debug:
    msg:
      - 'Ansible Version: {{ ansible_play_hosts }}'

- name: Set dynamic variable for node_ip
  set_fact:
    node_ip: "{{ hostvars[inventory_hostname][ipv4]   | default(ansible_default_ipv4.address) }}"
  tags:
    - consul
    - consul-container
    - consul-master
    - container

- name: Create consul data dir
  file:
    path: /home/docker/consul/{{ item }}
    state: directory
    recurse: yes
    owner: 100
    group: 1000
    mode: 0755
  with_items:
    - data
    - config
  tags:
    - consul

- debug: var=ansible_play_hosts
  tags:
    - consul

- name: Add consul config
  template:
    src: "consul.json.j2"
    dest: "/home/docker/consul/config/consul.json"
    owner: 100
    group: 1000
    mode: 0644
  notify:
    - consul reload
  tags:
    - consul
#
- name: Start consul
  docker_container:
    name: consul
    image: consul:1.8.5
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /home/docker/consul/data:/consul/data
      - /home/docker/consul/config:/consul/config
    hostname: "{{ ansible_hostname }}"
    env:
      SERVICE_8300_IGNORE: "true"
      SERVICE_8301_IGNORE: "true"
      SERVICE_8302_IGNORE: "true"
      SERVICE_8400_IGNORE: "true"
      SERVICE_8500_IGNORE: "true"
      SERVICE_8600_IGNORE: "true"
    command: agent -config-dir /consul/config
    state: started
    restart_policy: always
    ports:
      - 8300:8300
      - 8300:8300/udp
      - 8301:8301
      - 8301:8301/udp
      - 8302:8302
      - 8302:8302/udp
      - 8400:8400
      - 8500:8500
      - 8600:8600
      - 8600:8600/udp
    log_driver: json-file
    log_options:
      max-size: "1m"
      max-file: "1"
  tags:
    - consul
    -
- name: Fire handlers
  meta: flush_handlers
```

Рис.3.48 Роль Consul

Перш за все ми створюємо потрібні нам каталоги для збереження даних консула і для конфігураційних файлів, після цього створюємо сам конфігураційний файл який згенериться автоматично для кожного сервера свій. Див рис.3.49.

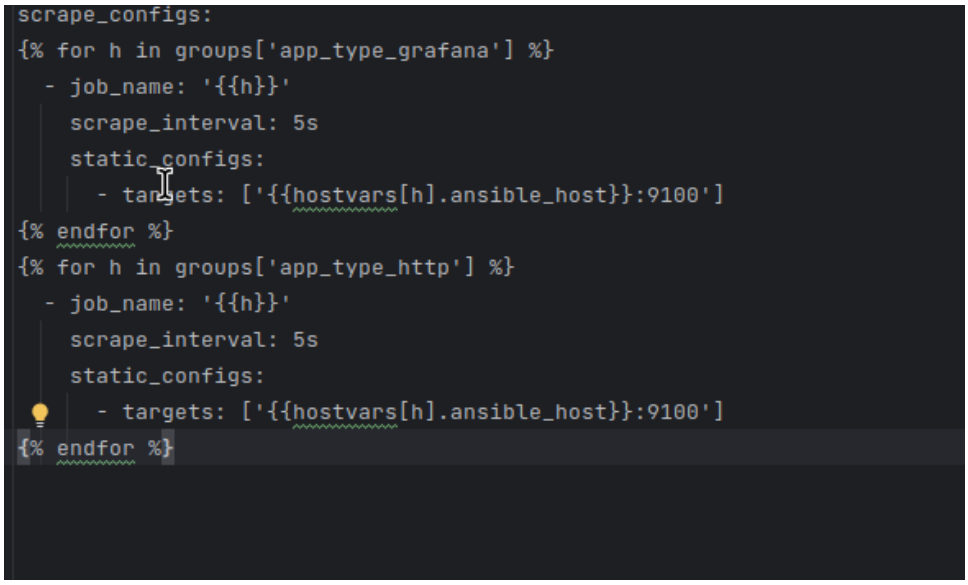


Рис.3.49. Конфігураційний файл консула

Після того як ми згенерували конфігураційний файл, можемо запускати контейнер з консулом, прокинувши вольюми з каталогами для збереження даний і конфігами:

volumes:

- /etc/localtime:/etc/localtime:ro
- /home/docker/consul/data:/consul/data
- /home/docker/consul/config:/consul/config

Також нам потрібно виконати команду для запуску консула в контейнері:

command: agent -config-dir /consul/config

Далі нам потрібно відкрити список портів:

Після того як пробіжить плейбук precondition.yml, ми можемо відкрити консул на будь якому з серверів з тестами навантаження по порту 8500, і якщо у нас відкрився консул, і всі сервери зареєстровані, то значить все працює вірно. Кластер консула показано на рис.3.43.

ports:

- 8300:8300
- 8300:8300/udp
- 8301:8301

- 8301:8301/udp
- 8302:8302
- 8302:8302/udp
- 8400:8400
- 8500:8500
- 8600:8600
- 8600:8600/udp

Після того як пробіжить плейбук precondition.yml, ми можемо відкрити консул на будь якому з серверів з тестами навантаження по порту 8500, і якщо у нас відкрився консул, і всі сервери зареєстровані, то значить все працює вірно. Кластер консула показано на рис.3.50.

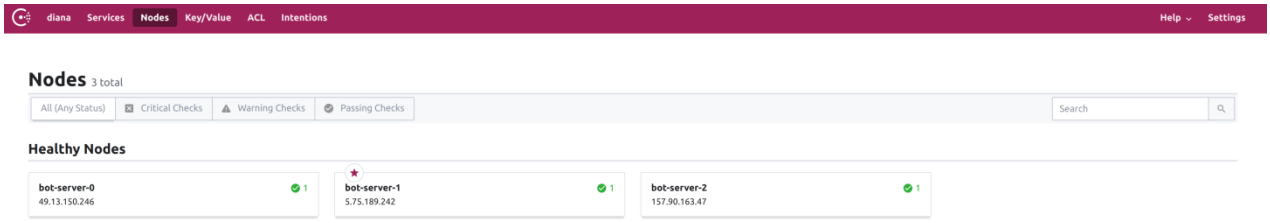
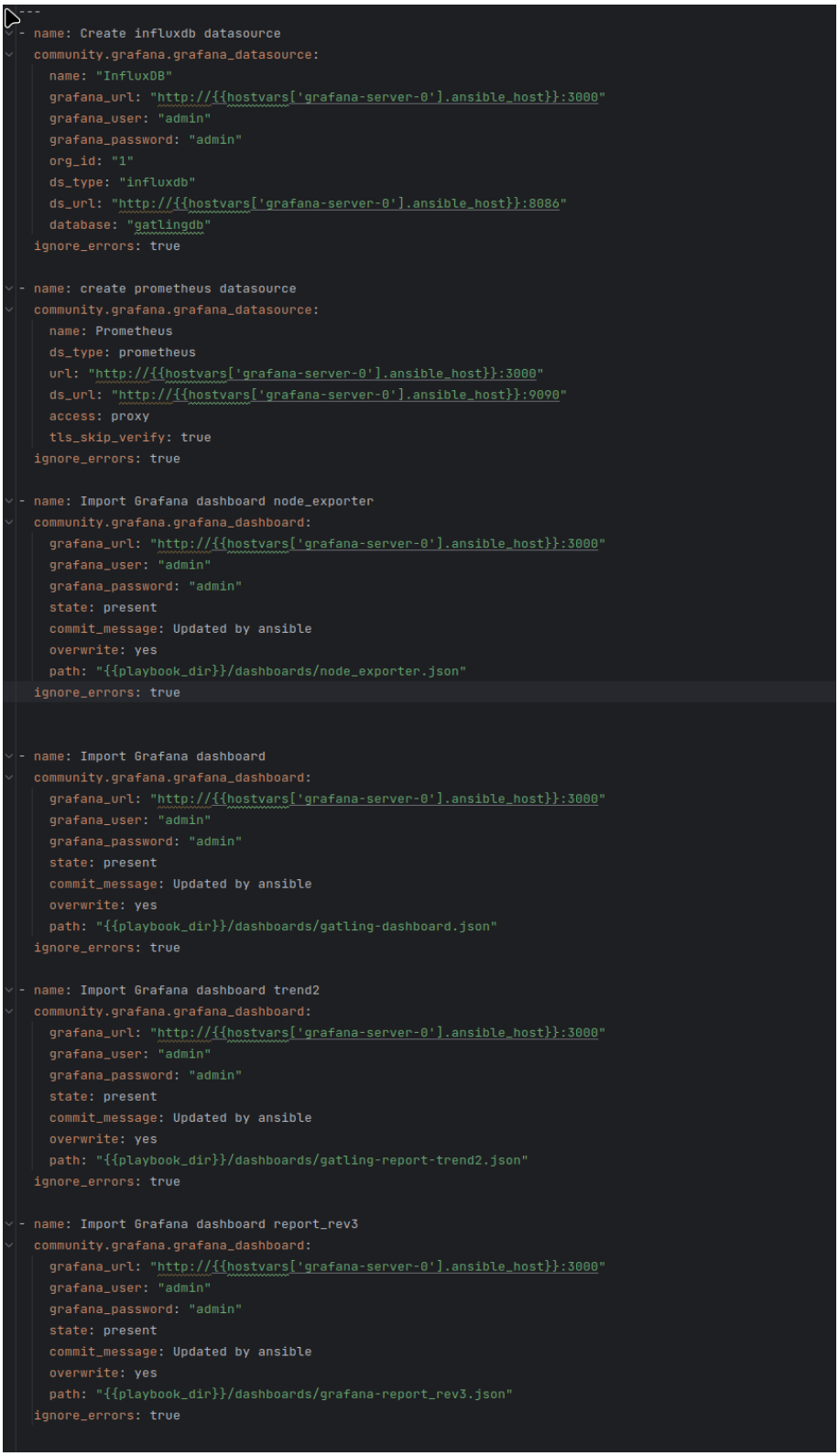


Рис.3.50 Кластер консула

Як бачимо все працює вірно, майстер нода позначена зірочкою. Інші дві ноди то консул агенти, які підключилися до майстра, що означить консул налаштований без попилок.

3.2.6 Налаштування і створення дашбоардів графани

Після того як ми розгорнули всю інфраструктуру можемо переходити до налаштування самої графани і дашбоардів для відображення наших метрик. Все це виконується в ролі configure_grafana_dashboards, показано на рис.3.51. яка запускається на локалхості, з якого ми запускаємо ансібл плейбуки.



```yaml
---
- name: Create influxdb datasource
  community.grafana.grafana_datasource:
    name: "InfluxDB"
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    org_id: "1"
    ds_type: "influxdb"
    ds_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:8086"
    database: "gatlingdb"
  ignore_errors: true

- name: create prometheus datasource
  community.grafana.grafana_datasource:
    name: Prometheus
    ds_type: prometheus
    url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    ds_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:9090"
    access: proxy
    tls_skip_verify: true
  ignore_errors: true

- name: Import Grafana dashboard node_exporter
  community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/node_exporter.json"
  ignore_errors: true


- name: Import Grafana dashboard
  community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/gatling-dashboard.json"
  ignore_errors: true

- name: Import Grafana dashboard trend2
  community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/gatling-report-trend2.json"
  ignore_errors: true

- name: Import Grafana dashboard report_rev3
  community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/grafana-report_rev3.json"
  ignore_errors: true
```

Рис.3.51 Роль configure_grafana_dashboards

Графана конфігурується модулем community.grafana.grafana_datasource. Перш за все нам потрібно створити датасоурси у графані. У нас є дві бази данних: прометей

і інфлюксДб. Відповідно нам потрібно налаштувати два датасоурса, які будуть мати наступну конфігурацію:

```
- name: Create influxdb datasource
 community.grafana.grafana_datasource:
 name: "InfluxDB"
 grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
 grafana_user: "admin"
 grafana_password: "admin"
 org_id: "1"
 ds_type: "influxdb"
 ds_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:8086"
 database: "gatlingdb"
 ignore_errors: true
- name: create prometheus datasource
 community.grafana.grafana_datasource:
 name: Prometheus
 ds_type: prometheus
 url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
 ds_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:9090"
 access: proxy
 tls_skip_verify: true
 ignore_errors: true
```

Після того як відпрацювала дана роль, ми можемо перейти в графані на вкладку коннекти, де у нас мають з’явитися 2 датасоурси, що є показано на рис.3.52.

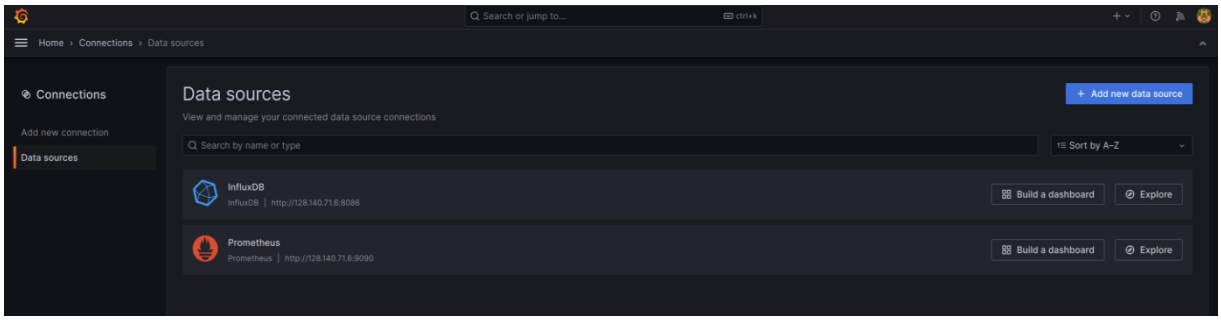


Рис.3.52 Датасоурси в графані

Відкриваєм тепер кожен з датасоурсів і дивимось на їх конфігурацію. На рис.3.53 показано датасоурс інфлюксдб, на рис.3.54 показано датасоурс до прометеуса.

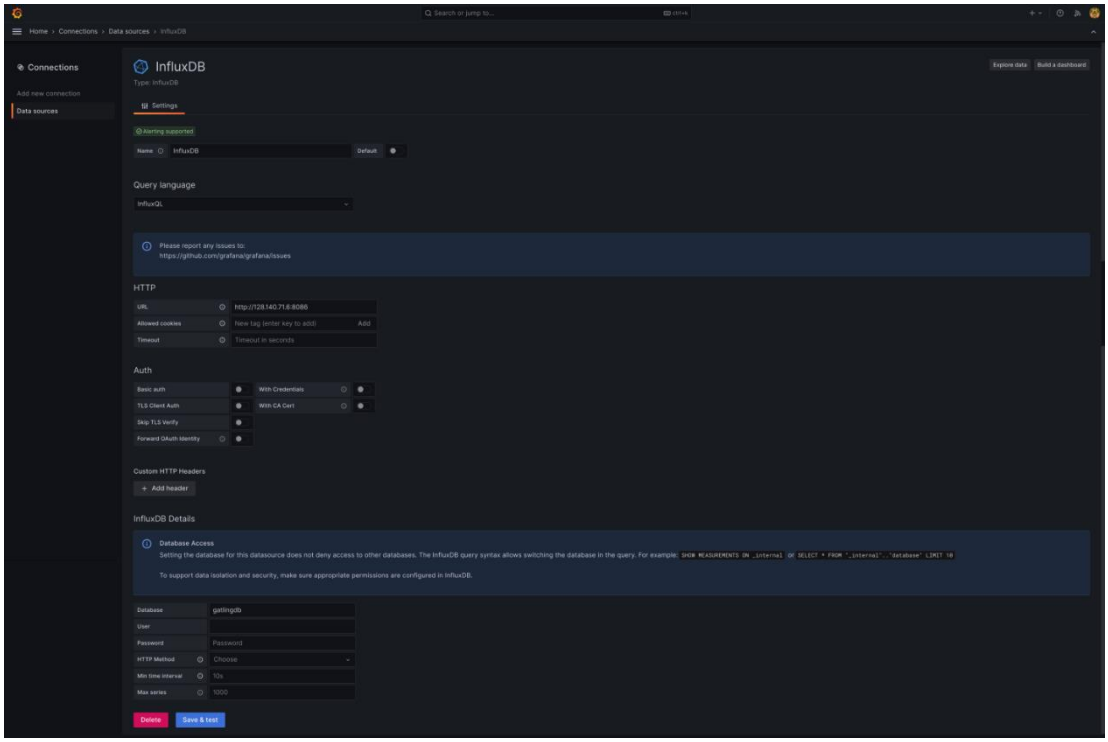


Рис.3.53 Датасоурс ІнфлюксДБ

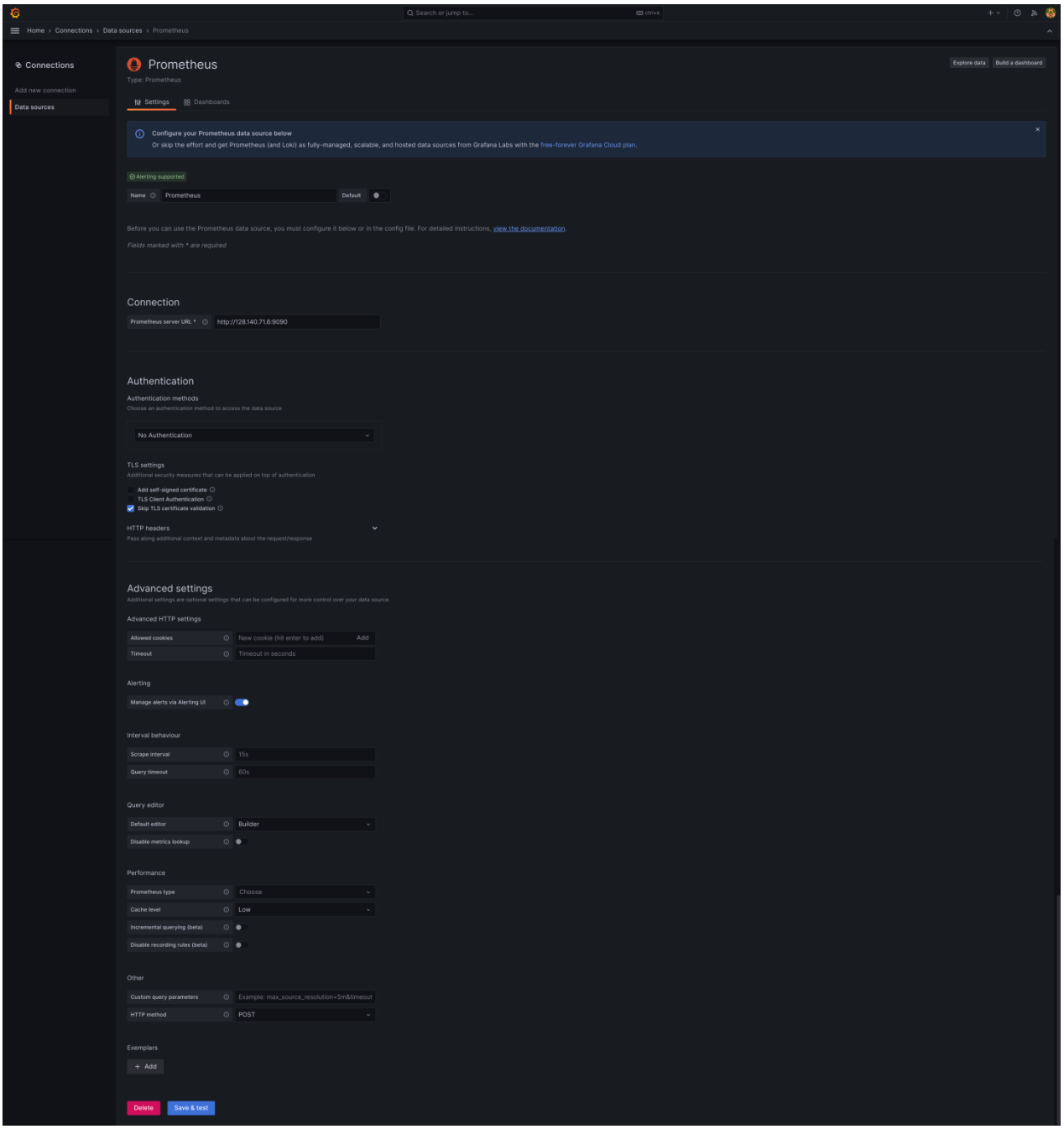


Рис.3.54 Датасоурс прометеус

У наших датасоурсах мають бути вказані айпі адреси серверів на яких запущені контейнери з базами данних на порт, який прокинутий до контейнера. Також потрібно вказати креди для підключення до баз даних. Всі інші параметри ми залишаємо по замовчуванню.

Після того як ми налаштували датасоурси нам потрібно нашатувати наші дашбоарди для візуалізації наших метрик. Це виконується модулем community.grafana.grafana_dashboard наступним чином:

```
name: Import Grafana dashboard node_exporter
 community.grafana.grafana_dashboard:
 grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
 grafana_user: "admin"
 grafana_password: "admin"
 state: present
 commit_message: Updated by ansible
 overwrite: yes
 path: "{{playbook_dir}}/dashboards/node_exporter.json"
 ignore_errors: true


- name: Import Grafana dashboard
 community.grafana.grafana_dashboard:
 grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
 grafana_user: "admin"
 grafana_password: "admin"
 state: present
 commit_message: Updated by ansible
 overwrite: yes
 path: "{{playbook_dir}}/dashboards/gatling-dashboard.json"
 ignore_errors: true

- name: Import Grafana dashboard trend2
 community.grafana.grafana_dashboard:
 grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
```

```
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/gatling-report-trend2.json"
    ignore_errors: true

  - name: Import Grafana dashboard report_rev3
    community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/grafana-report_rev3.json"
    ignore_errors: true
```

Для створення дашбоардів ми використаємо готові шаблони які можна завантажити на сайті https://grafana.com/grafana/dashboards/. З цього сайту обираємо дашбоард для ноде експортер і для гатлінга. Дані дашбоарди можна імпортувати вручну по айді дашбоарда, або скопіювати всю конфігурацію з дашбоарду в джейсон фонматі і так само заімпортити, але ми пішли шляхом автоматичного налаштування дашбордів. Для цього нам потрібно скопіювати джейсон конфігурацію дашбоаду, зберегти її у файлах, і передати у наш модуль в параметрі path. Після того як відпрацює дана роль ми маємо побачити 4 дашбоарди з метриками - два для гатлінга, і один для застосунка. Показано на рис.3.55.

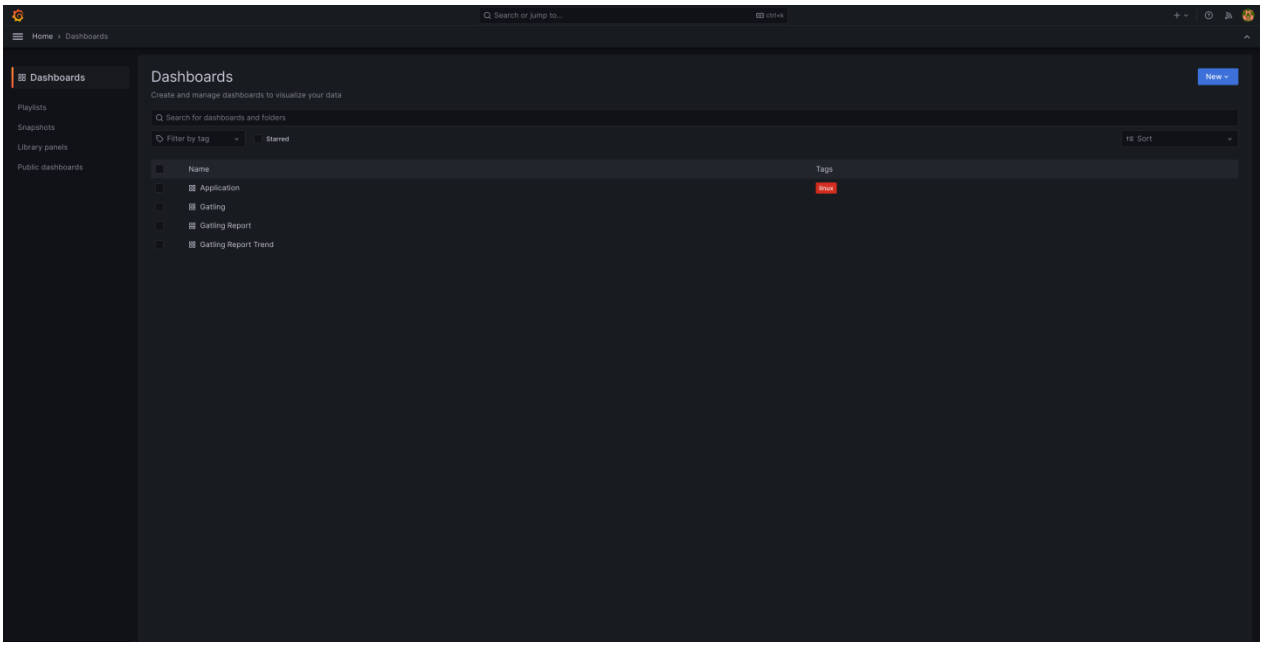


Рис.3.55 Дашбоарди графани

Розглянемо кожен дашбоард окремо. Дашбоард Application використовує датасоурс для прометеуса, в якому у нас зберігаються метрики з сервера на якому розгорнутий наш застосунок. В даному дашбоарді виводяться основні метрики для аналізу застосунку, а саме скільки він споживає потужностей процесора, як навантажеє мережу, скільки йому потрібно оперативної памяті, і як навантажує диски. Також виводиться основна інформація по потужностях сервера. Даний дашбоард показано на рис.3.56.

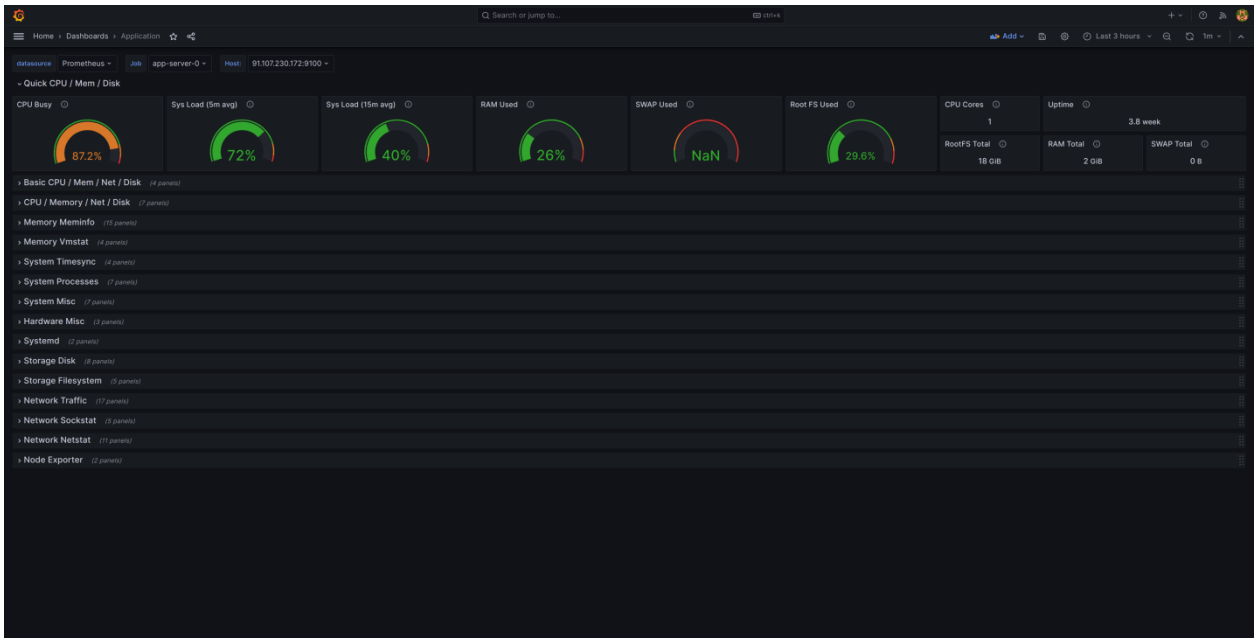


Рис.3.56 Дашбоард Application

Далі перейдемо до дашбоарду Gatling, показано на рис.3.57. Даний дашбоард використовує датасоурс інфлюксдб і виводить частину метрики які відправляє туди гатлінг. Тут ми бачимо метркики по таймінгах відповідей реквестів по 95 персентилю. В разі необхідності можна додати ще по 50, 90 і 99. А також кількість еорів і вдало відпрацьованих реквестів, майсимаьльні таймінги на відповіді, і загальну середню кількість реквестів, яку робить гатлінг.

Рис.3.57 Дашбоард Gatling

Наступний репорт - Gatling Report. Даний репорт також використовує дата провайтед для інфлюксДб і виводить метрики неузагальнені, а по кожному реквесту окремо. Також показує загальну кількість реквестів, таймінги, скільки реквестів у нас пройшло вдало, а скільки з ерорами, кількість реквестів по персентилях. Даний дашбоард показано на рис.3.58.

Рис.3.58 Дашбоард Gatling Report

Останній дашбоард Gatling Report Trend використовує датасоурс для інфлюксдб і виводить загальні метрики по нашій симуляції, а саме скільки часу триває тестуваннння, скільки реквестів було з ерорами протягом всієї симуляції, скільки

реквестів в секунду робить гатлінг під час симуляції. Тобто ми маємо узагальнену інформацію за якийсь період тестування. Даний дашбоард показано на рис.3.59.

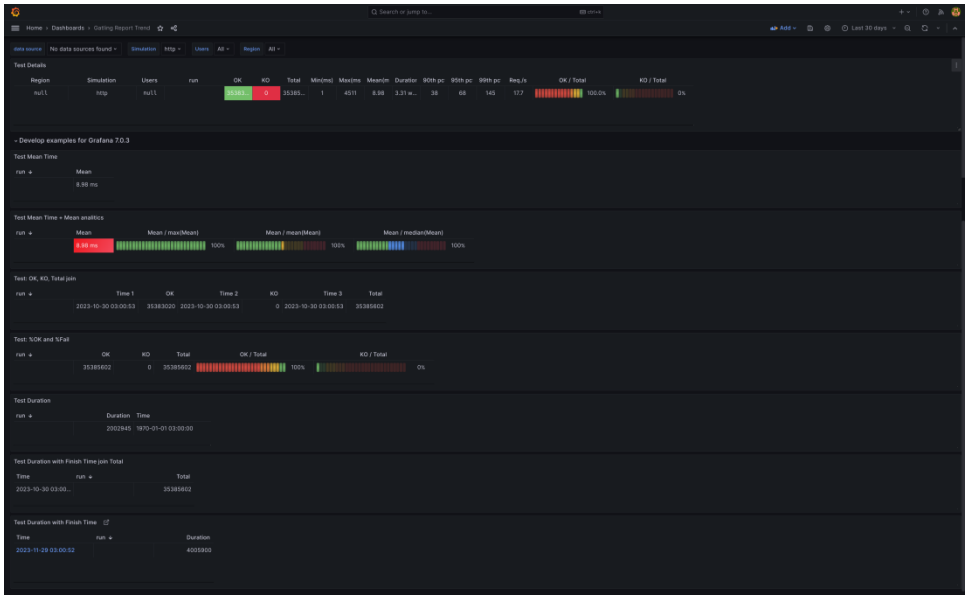


Рис.3.59 Дашбоард Gatling Report Trend

3.3 Розробка тестів навантаження

3.3.1 Загальна схема роботи тестів навантаження

Запуск тестів навантаження виконується плейбуком bot_start, в якому виконується наступна команда в контейнері з тестами навантаження:

command: mvn -X gatling:test -Dgatling.simulationClass=simulation.BasicSimulation -Dgatling.conf.file=gatling-configs/gatling.conf

Для тестів навантаження у нас створений мавен проект, і як ми бачимо з команди тести саме мавеном і запускаються. Також ми повинні передати симуляцію яку нам потрібно запустити і конфігураційний файл для самого гатлінгу.

Для мавен проекта основний конфігураційний файл це pom.xml, в якому ми вказуємо всі необхідні залежності, а саме бібліотеки джава і скала, які будуть завантажені з мавен репозиторія, і мавен плагіни які потрібні для запуску тестів

навантаження, написаних з використанням гатлінгу. Файл pom.xml показано на рис.3.60.

Рис. 3.60 Pom.xml тестів навантаження

В Pom.xml вказано два плагіна: scala-maven-plugin і gatling-maven-plugin. Перший потрібен для компіляції самого проекта, оскільки у нас використовується в проекті скала і сам гатлінг написаний на скала, другий потрібен для запуску тестів навантаження, в якому ми вказуємо пекедж з нашими симуляціями.

Також при запуску ми вказуємо конфігураційний файл для самого гатлінгу. Показано на рис.3.61.

```
gatling {
  core {
    #outputDirectoryBaseName = "" # The prefix for each simulation result folder (then suffixed by the report generation timestamp)
    #runDescription = ""          # The description for this simulation run, displayed in each report
    #encoding = "utf-8"           # Encoding to use throughout Gatling for file and string manipulation
    #simulationClass = ""         # The FQCN of the simulation to run (when used in conjunction with noReports, the simulation for which assertions will be validated)
    #elFileBodiesCacheMaxCapacity = 200      # Cache size for request body EL templates, set to 0 to disable
    #rawFileBodiesCacheMaxCapacity = 200     # Cache size for request body Raw templates, set to 0 to disable
    #rawFileBodiesInMemoryMaxSize = 1000     # Below this limit, raw file bodies will be cached in memory
    #pebbleFileBodiesCacheMaxCapacity = 200  # Cache size for request body Peeble templates, set to 0 to disable
    #feederAdaptiveLoadModeThreshold = 100   # File size threshold (in MB). Below load eagerly in memory, above use batch mode with default buffer size
    #shutdownTimeout = 10000                 # Milliseconds to wait for the actor system to shutdown
    extract {...}
    directory {...}
  }
  socket {
    connectTimeout = 300000                # Timeout in millis for establishing a TCP socket
    tcpNoDelay = true
    #soKeepAlive = false                   # if TCP keepalive configured at OS level should be used
    #soReuseAddress = false
  }
  netty {...}
  ssl {
    #useOpenSsl = true                    # if OpenSSL should be used instead of JSSE (only the latter can be debugged with -Djava.net.debug=ssl)
    #useOpenSslFinalizers = false         # if OpenSSL contexts should be freed with Finalizer or if using RefCounted is fine
    #handshakeTimeout = 10000             # TLS handshake timeout in millis
    #useInsecureTrustManager = true       # Use an insecure TrustManager that trusts all server certificates
    #enabledProtocols = []             # Array of enabled protocols for HTTPS, if empty use Netty's defaults
    #enabledCipherSuites = []          # Array of enabled cipher suites for HTTPS, if empty enable all available ciphers
    #sessionCacheSize = 0              # SSLSession cache size, set to 0 to use JDK's default
    #sessionTimeout = 0                # SSLSession timeout in seconds, set to 0 to use JDK's default (24h)
    #enableSni = true                     # When set to true, enable Server Name indication (SNI)
    keyStore {...}
    trustStore {...}
  }
  charting {
    noReports = true       # When set to true, don't generate HTML reports
    #maxPlotPerSeries = 1000 # Number of points per graph in Gatling reports
    #useGroupDurationMetric = false  # Switch group timings from cumulated response time to group duration.
    indicators {
      #lowerBound = 800      # Lower bound for the requests' response time to track in the reports and the console summary
      #higherBound = 1200    # Higher bound for the requests' response time to track in the reports and the console summary
      percentile1 = 50      # Value for the 1st percentile to track in the reports, the console summary and Graphite
      percentile2 = 90      # Value for the 2nd percentile to track in the reports, the console summary and Graphite
      percentile3 = 95      # Value for the 3rd percentile to track in the reports, the console summary and Graphite
      percentile4 = 99      # Value for the 4th percentile to track in the reports, the console summary and Graphite
    }
  }
  http {
    #fetchedCssCacheMaxCapacity = 200          # Cache size for CSS parsed content, set to 0 to disable
    #fetchedHtmlCacheMaxCapacity = 200         # Cache size for HTML parsed content, set to 0 to disable
    #perUserCacheMaxCapacity = 200             # Per virtual user cache size, set to 00 to disable
    #warmUpUrl = "https://gatling.io"          # The URL to use to warm-up the HTTP stack (blank means disabled)
    #enableGA = true                           # Very light Google Analytics (Gatling and Java version), please support
    pooledConnectionIdleTimeout = 300000       # Timeout in millis for a connection to stay idle in the pool
    requestTimeout = 300000                    # Timeout in millis for performing an HTTP request
    #enableHostnameVerification = false        # When set to true, enable hostname verification: SSLEngine.setHttpsEndpointIdentificationAlgorithm("HTTPS")
    dns {
      #queryTimeout = 5000                     # Timeout in millis of each DNS query in millis
      #maxQueriesPerResolve = 6                # Maximum allowed number of DNS queries for a given name resolution
    }
  }
  jms {
    #replyTimeoutScanPeriod = 1000  # scan period for timedout reply messages
  }
  data {
    writers = [console, graphite]      # The list of DataWriters to which Gatling write simulation data (currently supported : console, file, graphite)
    console {
      #light = false                # When set to true, displays a light version without detailed request stats
      #writePeriod = 5              # Write interval, in seconds
    }
    leak {
      #noActivityTimeout = 30  # Period, in seconds, for which Gatling may have no activity before considering a leak may be happening
    }
    graphite {
#       light = true              # only send the all* stats
      host = "128.140.71.6"         # The host where the Carbon server is located
      port = 2003                # The port to which the Carbon server listens to (2003 is default for plaintext, 2004 is default for pickle)
      protocol = "tcp"           # The protocol used to send data to Carbon (currently supported : "tcp", "udp")
      rootPathPrefix = "gatling.http" # The common prefix of all metrics sent to Graphite
#       bufferSize = 8192          # Internal data buffer size, in bytes
#       writePeriod = 10           # Write period, in seconds
    }
  }
}
```

Рис.3.61 Конфігураційний файл гатлінгу

Даний конфігкраційний файл має бути створений в ресурсах проекта. Тут можна налаштовувати кор гатлінга, сокети, ссл, репортінг і персентилі, хтта протокол.

Всю цю конфігурацію ми залишимо по замовчуванню. Нам потрібно тут тільки змінити налаштування графіту, де нам треба вказати айпі адресу сервера на якому у нас запущений інфлюксДБ, а також порт, на якому запущений ІнфлюксДб. Окрім цього протокол, по якому гатлінг буде відправляти метрики в інфлюксДБ, і префікс для наших метрик. Даний префікс ми побачимо на дашбоарді метрик з гатлінгу.

Після цього ми маємо налаштувати логування в нашому проекті. Для цього використовуємо бібліотеку Logback. Для цього в ресурсах нам потрібно створити наступний файл - logback.xml, показано на рис.3.62.

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

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy/MM/dd HH:mm:ss:SSS}|%p||autotests||${instance_id}|%logger{0}|%m%n</pattern>
        </encoder>
        <immediateFlush>true</immediateFlush>
    </appender>
    <appender name="ASYNC_CONSOLE" class="ch.qos.logback.classic.AsyncAppender">
        <!--<queueSize>1024</queueSize>-->
        <discardingThreshold>0</discardingThreshold>
        <appender-ref ref="CONSOLE"/>
    </appender>

    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${Log4jFileName}</file>
        <!--        <file>./log/loadTest.log</file>-->
        <encoder>
            <pattern>%d{yyyy/MM/dd HH:mm:ss:SSS}|%p||autotests||${instance_id}|%logger{0}|%m%n</pattern>
        </encoder>
        <immediateFlush>true</immediateFlush>
        <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
            <fileNamePattern>${Log4jFileName}_%d{yyyy-MM-dd}_%i.log</fileNamePattern>
            <maxFileSize>9MB</maxFileSize>
        </rollingPolicy>
    </appender>
    <appender name="ASYNC_FILE" class="ch.qos.logback.classic.AsyncAppender">
        <discardingThreshold>0</discardingThreshold>
        <appender-ref ref="FILE"/>
    </appender>

    <logger name="io.gatling" level="INFO"/>
    <logger name="simulation" level="INFO"/>
    <logger name="scenarios" level="INFO"/>
    <logger name="core" level="INFO"/>

    <root level="WARN">
        <appender-ref ref="ASYNC_CONSOLE"/>
        <!--        <appender-ref ref="ASYNC_FILE"/>-->
    </root>

</configuration>
```

Рис.3.62 Конфігураційний файл для Logback

В даному файлі ми налаштовуємо типи логування і формат наших логів. Це все нам потрібно буде для перевірки коректності роботи наших сценаріїв, і пошуку та аналізу проблем, якщо такі будуть.

Після того як ми виконали базові налаштування можемо перейти до написання самих тестів навантаження. Вся діаграма класів тестів навантаження показана на рис. 3.63.

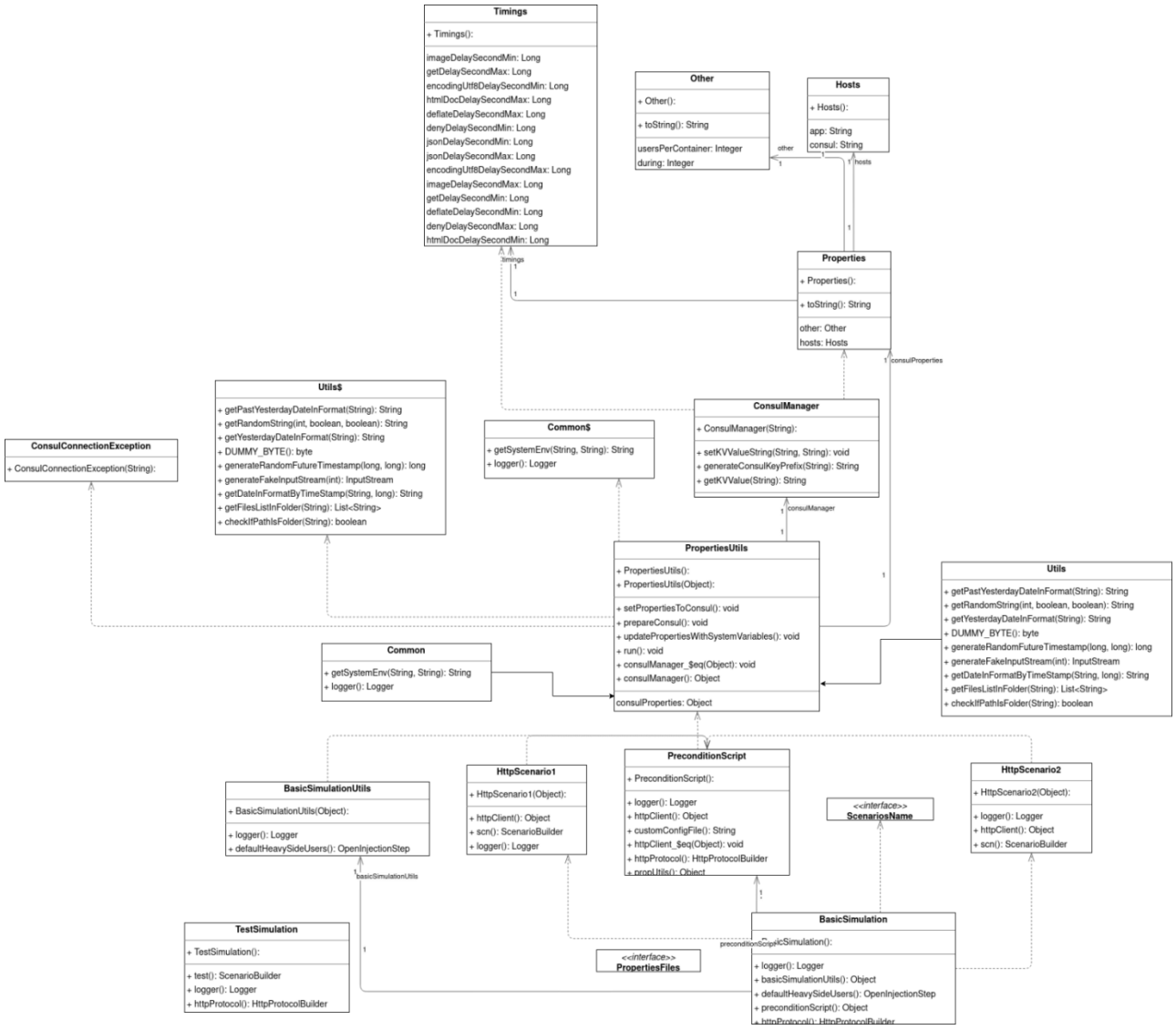


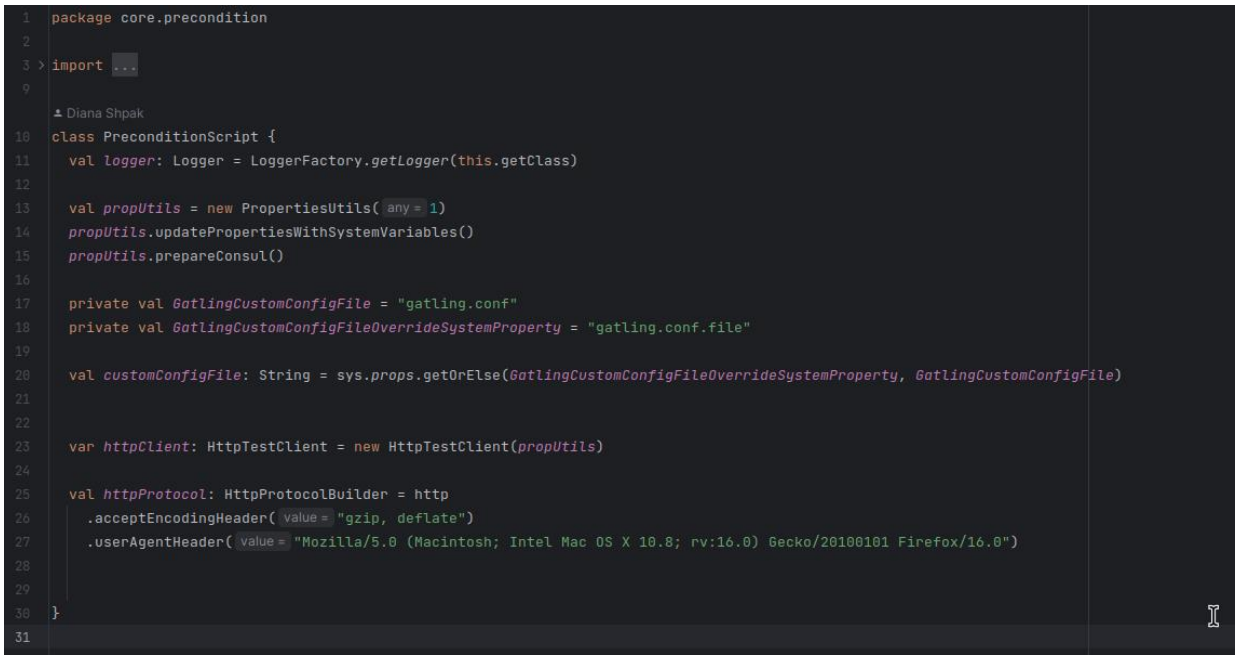
Рис.3.63 Діаграма класів тестів навантаження

Запуск самих тестів починається з запуску симуляції яку ми передаємо параметром -Dgatling.simulationClass=simulation.BasicSimulation. Це клас який є наслідником класу Simulation, показано на рис. 3.64.

```scala
package simulation

import core.constans.ScenariosName
import core.precondition.PreconditionScript
import io.gatling.core.Predef._
import io.gatling.core.config.GatlingConfiguration
import io.gatling.core.controller.inject.open.OpenInjectionStep
import io.gatling.http.protocol.HttpProtocolBuilder
import org.slf4j.{Logger, LoggerFactory}
import scenarios.{HttpScenario1, HttpScenario2}

import scala.concurrent.duration.DurationInt

Diana Shpak
class BasicSimulation extends Simulation {
  GatlingConfiguration.loadForTest()
  GatlingConfiguration.loadActorSystemConfiguration()

  val logger: Logger = LoggerFactory.getLogger(this.getClass)
  val preconditionScript: PreconditionScript = new PreconditionScript
  val basicSimulationUtils: BasicSimulationUtils = new BasicSimulationUtils(preconditionScript)
  val httpProtocol: HttpProtocolBuilder = preconditionScript.httpProtocol
  val defaultHeavySideUsers: OpenInjectionStep = basicSimulationUtils.defaultHeavySideUsers()

  preconditionScript.propUtils.getConsulProperties.getOther.getScenario match {
    case ScenariosName.HTTP1 =>
      setUp(
        new HttpScenario1(preconditionScript).scn.inject(defaultHeavySideUsers)).protocols(httpProtocol)

    case ScenariosName.HTTP2 =>
      setUp(
        new HttpScenario2(preconditionScript).scn.inject(defaultHeavySideUsers)).protocols(httpProtocol)

    case _ =>
      throw new Exception("WRONG SCENARIO NAME")
  }
}
```

Рис. 3.64 Базова симуляція

Для наших тестів навантаження нам потрібне централізоване сховище для параметрів тестів. Ми обрали консул, відповідно перед запуском наших тестів нам потрібно перш за все завантажити наші дефолтні параметри у консул(всі вони лежать в ресурсах, і це все виконується в обєкті PreconditionScript). Показано на рис.3.65.



```
package core.precondition

import ...

Diana Shpak
class PreconditionScript {
  val logger: Logger = LoggerFactory.getLogger(this.getClass)

  val propUtils = new PropertiesUtils( any = 1)
  propUtils.updatePropertiesWithSystemVariables()
  propUtils.prepareConsul()

  private val GatlingCustomConfigFile = "gatling.conf"
  private val GatlingCustomConfigFileOverrideSystemProperty = "gatling.conf.file"

  val customConfigFile: String = sys.props.getOrElse(GatlingCustomConfigFileOverrideSystemProperty, GatlingCustomConfigFile)


  var httpClient: HttpTestClient = new HttpTestClient(propUtils)

  val httpProtocol: HttpProtocolBuilder = http
    .acceptEncodingHeader( value = "gzip, deflate")
    .userAgentHeader( value = "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0) Gecko/20100101 Firefox/16.0")


}

```

Рис. 3.65 Обєкт PreconditionScript

Метод updatePropertiesWithSystemVariables оновлює проперті в файлах за наступними пріоритетами: найвищий пріоритет в параметрах які зберігаються в змінних середовища в якому запускаються тести навантаження(це зроблено для того щоб у нас була можливість конфігурувати наші проперті передаючи їх як змінну середовища в контейнер з тестами навантаження), далі параметри з проекта який ми можемо змінювати і створювати декілька проектів з своїми змінними(це необхідно для того щоб ми могли зберегти набір тестів навантаження під певний продакшин, і найнижчий пріоритет мають параметри в проперті файлах які лежать в commonDefaultProperties, показано на рис.3.66. Дефолтні параметри нам потрібні для встановлення значень по замовчуванню для параметрів які не потрібно змінювати, або для тих які є загальними для всіх наших продакшинів. Метод prepareConsul зберігає всі наші параметри в консул після їх оновлення. Після цього як у нас налаштований консул, далі ми зможемо ці параметри використовувати в наших тестах навантаження. Також нам потрібно тут створити клієнти до наших сервісів (у нас він один - httpClient,) і базову конфігурацію нашого клієнта. Оскільки в нас може бути декілька симуляцій, то винесемо базову конфігурацію симуляції в обєкт BasicSimulationUtils. Показано на рис.3.66. В даному обєкті втановлюється кількість

юзерів. В гатлінгу один юзер це один потік, а дюрінг це час протягом якого запустяться всі юзери-потоки. Цей параметр дуже важливий, оскільки у нас може бути дуже багато потоків, і щоб вони всі одночасно не запустились, то можна розділити їх в часі, і таким чином не створимо критичну ситуацію на застосунку який тестуємо. Варто зазначити, що дана ситуація є нереальною для будь якого продакшина аби всі користувачі зайшли одночасно на аплікейшин.

Рис. 3.66. Обєкт BasicSimulationUtils

У нас 2 сценарія використання даного продукту, тому у нас має бути можливість запускати той чи інший сценарій. Це все реалізовано в базовій симуляції свіч кейсом. Сценарій який нам потрібно запустити будемо передавати параметром з плейбука.

3.3.2 Розробка сценаріїв навантаження

Переходимо до розробки нашого сценарію. Зазвичай для цього, якщо у нас є можливість, потрібно проаналізувати наш продакшин, а саме його логи і метрики, дізнатися яку кількість і які саме реквести користувач робить на сервіси і з якими параметрами, зібрати статистику за певний час, додати це апі в сценарій, і в результаті налаштувати сценарій таким чином, аби кількість реквестів відповідала продакшину.

Потрібний нам сценарій у нас запускається коли у файлі в змінну other.properties в змінну scenario передаємо http1 чи http2. Відповідно цього параметру попадаємо у відповідний кейс і запускаємо сценарій. Також в даному файлі ми можемо змінити

кількість потоків в контейнері, змінивши значення змінної usersPerContainer, і час протягом якого всі потокі запустяться за допомогою змінної during. Показано на рис.3.67.

```
overwriteConsul=1
scenario=http1
usersPerContainer=1
during=1
```

Рис.3.67 Конфігураційний файл other.properties

Після цього даний парамер попадає в свіч кейс в базовій симуляції і запускається перший сценарій для навантаження. Показано на рис.3.68.

```
preconditionScript.propUtils.getConsulProperties.getOther.getScenario match {
  case ScenariosName.HTTP1 =>
    setUp(
      new HttpScenario1(preconditionScript).scn.inject(defaultHeavySideUsers)).protocols(httpProtocol)

  case ScenariosName.HTTP2 =>
    setUp(
      new HttpScenario2(preconditionScript).scn.inject(defaultHeavySideUsers)).protocols(httpProtocol)

  case _ =>
    throw new Exception("WRONG SCENARIO NAME")
}
```

Рис. 3.68 Свіч кейс для запуску першого сценарія

Розглянемо наші сценарії: сценарій http1 показано на рис.3.69, сценарій http2 показано на рис.3.70.

Перш за все в нашому сценарії ми створюємо логер для виведення логів. Далі ми створюємо клієнт з запитами до нашого сервісу. Оскільки у нас тести навантаження написані з використанням гатлінга, то наш сценарій пишеться в блоці ScenarioBuilder. На початку сценарія ми оголошужємо глобальні змінні які нам потрібні для апі, після цього у нас йде неприривний цикл в якому викликається апі з нашого клієнта. Показано на рис.3.71.

```
package scenarios

import ...


Diana Shpak
class HttpScenario1(preconditionScript: PreconditionScript) {
  val logger: Logger = LoggerFactory.getLogger(this.getClass)

  val httpClient: HttpTestClient = preconditionScript.httpClient

  val scn: ScenarioBuilder = scenario( scenarioName = "#HTTP SCENARIO 1#").
    exec(
      exec {
        session =>
          val ipApp = preconditionScript.propUtils.getConsulProperties.getHosts.getApp
          logger.info("ipApp" , ipApp)
          session
            .set("doGetTime", 0L).set("getTimeCalculate", false)
            .set("doDeflateTime", 0L).set("deflateTimeCalculate", false)
            .set("doDenyTime", 0L).set("denyTimeCalculate", false)
            .set("doEncodingUtf8Time", 0L).set("encodingUtf8TimeCalculate", false)
      }.
      exec(
        forever() {
          exec(
            httpClient.get,
            httpClient.deflate,
            httpClient.deny,
            httpClient.encodingUtf8,
            pause(Duration.create(1000, TimeUnit.MILLISECONDS))
          )
        }
      )
    )
}
```

Рис.3.69 Сценарій HTTP1

```
1  package scenarios
2
3  import ...
11
12
   Diana Shpak
13 class HttpScenario2(preconditionScript: PreconditionScript) {
14   val logger: Logger = LoggerFactory.getLogger(this.getClass)
15
16   val httpClient: HttpTestClient = preconditionScript.httpClient
17
18   val scn: ScenarioBuilder = scenario( scenarioName = "#HTTP SCENARIO 1#").
19     exec(
20       exec {
21         session =>
22           val ipApp = preconditionScript.propUtils.getConsulProperties.getHosts.getApp
23           logger.info("ipApp" , ipApp)
24           session
25             .set("doHtmlDocTime", 0L).set("htmlDocTimeCalculate", false)
26             .set("doJsonTime", 0L).set("jsonTimeCalculate", false)
27             .set("doImageTime", 0L).set("imageTimeCalculate", false)
28       }.
29         exec(
30           forever() {
31             exec(
32               httpClient.htmlDoc,
33               httpClient.json,
34               httpClient.image,
35               pause(Duration.create(1000, TimeUnit.MILLISECONDS))
36             )
37           }
38         )
39     )
40 }
41
```

Рис.3.70 Сценарій HTTP2

```
package core.servicesClients.http

import core.utils.{PropertiesUtils, Utils}
import io.gatling.core.Predef._
import io.gatling.core.structure.ChainBuilder
import io.gatling.http.Predef._
import org.slf4j.{Logger, LoggerFactory}

import java.time.Instant

▲ Diana Shpak
class HttpTestClient(val propertiesUtils: PropertiesUtils) {
  val logger: Logger = LoggerFactory.getLogger(this.getClass)

  val GET: String = "/get"
  val DEFLATE: String = "/deflate"
  val DENY: String = "/deny"
  val ENCODING_UTF8: String = "/encoding/utf8"
  val HTML_DOC: String = "/html"
  val JSON: String = "/json"
  val IMAGE: String = "/image"


  val get: ChainBuilder = exec(
    doIf(session => session("getTimeCalculate").as[Boolean] == true) {
      exec {
        session =>
          val stateTime = session("doGetTime").as[Long]
          val delaySecondMin = propertiesUtils.getConsulProperties.getTimings.getGetDelaySecondMin.longValue()
          val delaySecondMax = propertiesUtils.getConsulProperties.getTimings.getGetDelaySecondMax.longValue()
          val randomTime = delaySecondMin + (Math.random * ((delaySecondMax - delaySecondMin) + 1)).asInstanceOf[Long]
          val time = stateTime + randomTime
          session.set("doGetTime", time).set("getTimeCalculate", false)
      }
    },
    doIf(session => session("doGetTime").as[Long] <= (Instant.now().toEpochMilli / 1000)) {
      exec(
        http( requestName = "GET")
          .get(session => "http://"  + propertiesUtils.getConsulProperties.getHosts.getApp + GET)
          .check(status.is( expected = 200))
          .check(bodyString.saveAs( key = "getResponse"))
      ).exec { session =>
        val response: String = session("getResponse").as[String]
        logger.info("getResponse - " + response)
        session
          .set("doGetTime", Instant.now().toEpochMilli / 1000)
          .set("getTimeCalculate", true)
      }
    }
  )

  val deflate: ChainBuilder = exec(
    doIf(session => session("deflateTimeCalculate").as[Boolean] == true) {
      exec {
        session =>
          val stateTime = session("doDeflateTime").as[Long]
          val delaySecondMin = propertiesUtils.getConsulProperties.getTimings.getDeflateDelaySecondMin.longValue()
          val delaySecondMax = propertiesUtils.getConsulProperties.getTimings.getDeflateDelaySecondMax.longValue()
          val randomTime = delaySecondMin + (Math.random * ((delaySecondMax - delaySecondMin) + 1)).asInstanceOf[Long]
          val time = stateTime + randomTime
          session.set("doDeflateTime", time).set("deflateTimeCalculate", false)
      }
    },
    doIf(session => session("doDeflateTime").as[Long] <= (Instant.now().toEpochMilli / 1000)) {
      exec(
        http( requestName = "DEFLATE")
          .get(session => "http://" + propertiesUtils.getConsulProperties.getHosts.getApp + DEFLATE)
          .check(status.is( expected = 200))
          .check(bodyString.saveAs( key = "deflateResponse"))
      ).exec { session =>
        val response: String = session("deflateResponse").as[String]
        logger.info("deflateResponse - " + response)
        session
          .set("doDeflateTime", Instant.now().toEpochMilli / 1000)
          .set("deflateTimeCalculate", true)
      }
    }
  )

  val deny: ChainBuilder = ...

  val encodingUtf8: ChainBuilder = ...

  val htmlDoc: ChainBuilder = ...

  val json: ChainBuilder = ...

  val image: ChainBuilder = ...
}
```

Рис. 3.71 Клієнт для нашого застосунку

Давайте розглянемо один в реквестів в нашому клієнті. Існує два основні блоки: в перноші вираховуються таймінги по яким викликається реквест, в другому викликається сам реквест. Всі діапазони таймінгів зберігаються в проперті файлах в ресурсах проекту. Показано на рис.3.72

Рис. 3.72 Конфігураційни файз з діапазонами таймінгів для реквестів

Ми задаємо максимальне і мінімальне значення в секундах. Після виклику апі рандомом обирається число з даного діапазону, і наступний виклик відбудеться через час який нам випав. Це реалізовано для того щоб апі не викликалось синхронно, а його виклики кожен раз змінювались в потрібному діапазоні часу, аби змоделювати роботу реального користувача.

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному розділі було виконано розробку фреймворку тестування навантаження веб додатків, який скрадатиметься з декількох складових:

- проекту навантаження, де міститимуться сценарії тестування;
- проекту розгортання, деплою інфраструктури для запуску тестів;
- проекту розгортання, деплою інфраструктури для системи моніторингу;
- проекту для розгортання додатку який тестуємо.

Виконано огляд засобів реалізації програмного додатку, обґрунтовано та наведено перелік необхідних сервісів для роботи додатку. Зазначено мінімальні системні вимоги для коректної роботи розробленої системи, а також визначено основні завдання, що мають бути вирішені програмою.

Розглянуто структуру програмного додатку та описано основні функціональні рішення, використані при його розробці. Надано короткий та змістовний опис роботи з системою. Визначено основні етапи роботи розробленої системи, сформовано та взаємопов'язано основні компоненти програмного рішення.

4 РОЗДІЛ. РЕЗУЛЬТАТИ ТЕСТУВАННЯ

4.1 Запуск тестів навантаження і аналіз результатів за першим сценарієм

Для запуску нашого сценарію нам потрібно підняти 2 сервери, що ми зробимо тераформом. Для цього встановимо в файлі variables.tf, в параметр instances = 2, і виконаємо команду terraform plan. Дана команда покаже нам ресурси які створяться в клауді. Це лише вивід інформації, поки без реалізації. Результати виконання команди показані на рис. 4.1.

```
# hcloud_server.web[0] will be created
+ resource "hcloud_server" "web" {
    + allow_deprecated_images    = false
    + backup_window              = (known after apply)
    + backups                    = false
    + datacenter                 = (known after apply)
    + delete_protection          = false
    + firewall_ids               = (known after apply)
    + id                         = (known after apply)
    + ignore_remote_firewall_ids = false
    + image                      = "ubuntu-22.04"
    + ipv4_address               = (known after apply)
    + ipv6_address               = (known after apply)
    + ipv6_network               = (known after apply)
    + keep_disk                  = false
    + labels                     = {
        + "type" = "bot"
      }
    + location                   = "nbg1"
    + name                       = "bot-server-0"
    + rebuild_protection         = false
    + server_type                = "cx11"
    + ssh_keys                   = [
        + "13992008",
      ]
    + status                     = (known after apply)
    + user_data                  = "QxEK9LpOj64qw46IgJOiZDWjnKw="
  }

# hcloud_server.web[1] will be created
+ resource "hcloud_server" "web" {
    + allow_deprecated_images    = false
    + backup_window              = (known after apply)
    + backups                    = false
    + datacenter                 = (known after apply)
    + delete_protection          = false
    + firewall_ids               = (known after apply)
    + id                         = (known after apply)
    + ignore_remote_firewall_ids = false
    + image                      = "ubuntu-22.04"
    + ipv4_address               = (known after apply)
    + ipv6_address               = (known after apply)
    + ipv6_network               = (known after apply)
    + keep_disk                  = false
    + labels                     = {
        + "type" = "bot"
      }
    + location                   = "nbg1"
    + name                       = "bot-server-1"
    + rebuild_protection         = false
    + server_type                = "cx11"
    + ssh_keys                   = [
        + "13992008",
      ]
    + status                     = (known after apply)
    + user_data                  = "QxEK9LpOj64qw46IgJOiZDWjnKw="
  }

# hcloud_server_network.web_network[0] will be created
+ resource "hcloud_server_network" "web_network" {
    + id          = (known after apply)
    + ip          = (known after apply)
    + mac_address = (known after apply)
    + server_id   = (known after apply)
    + subnet_id   = (known after apply)
  }

# hcloud_server_network.web_network[1] will be created
+ resource "hcloud_server_network" "web_network" {
    + id          = (known after apply)
    + ip          = (known after apply)
    + mac_address = (known after apply)
    + server_id   = (known after apply)
    + subnet_id   = (known after apply)
  }

Plan: 6 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + web_servers_ips    = {
      + bot-server-0 = (known after apply)
      + bot-server-1 = (known after apply)
    }
  + web_servers_status = {
      + bot-server-0 = (known after apply)
      + bot-server-1 = (known after apply)
    }

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.
```

Рис. 4.1 Виконання команди terraform plan

З результатів виконання команди ми бачимо що створиться 6 ресурсів і 2 сервера на нашому клауді. Тепер нам потрібно застосувати дані зміни командою terraform apply. Результати виконання команди показані на рис.4.2

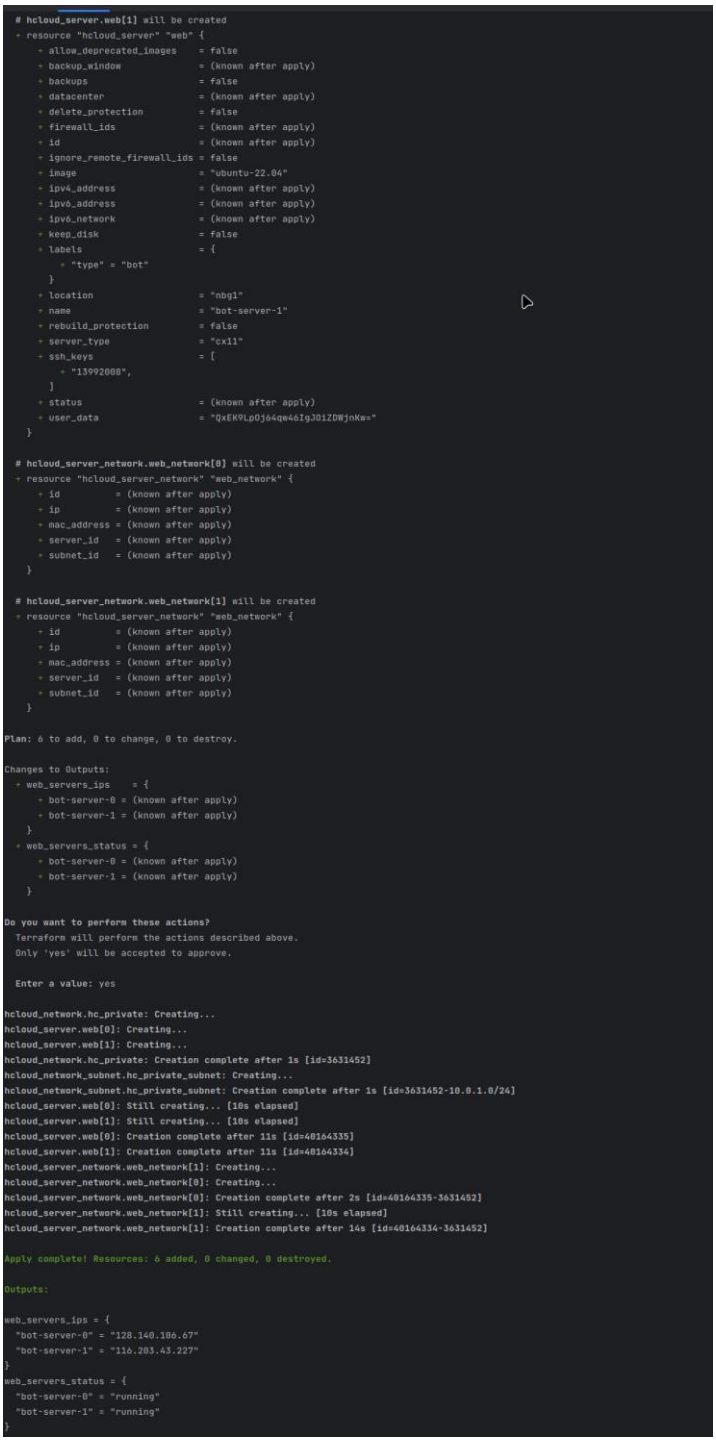


Рис.4.2 Результати виконання команди terraform apply

З результатів виконання terraform apply ми бачимо що всі сервери створені, і terraform вивів нам наші айпі адреси серверів. Також ми це можемо перевірити на

клауді, перейшовши на вкладку servers, де побачимо 2 додаткових сервера для запуску наших тестів навантаження. Показано на рис.4.3.

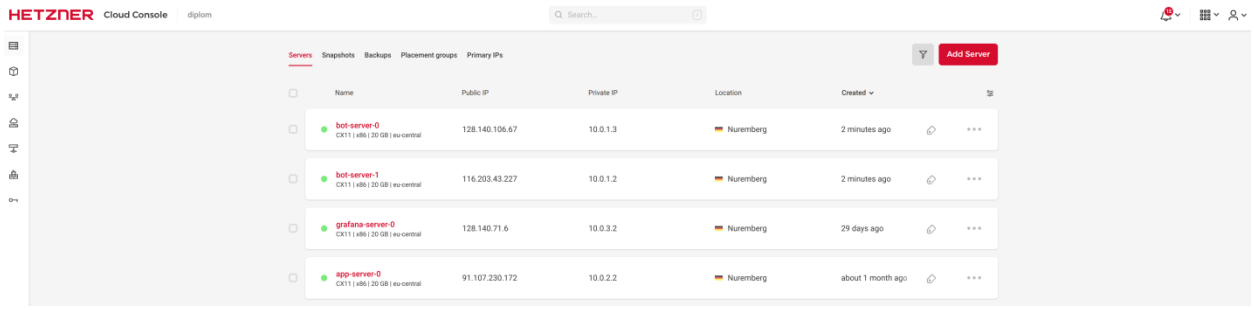


Рис.4.3 Список серверів на клауді

Після того як наша інфраструктура створена можемо перейти до налаштування самих машин. Наступним кроком буде запуск плейбука precondition командою ansible-playbook -i inventory precondition.yml. Результати виколнання якого показані на рис.4.4.

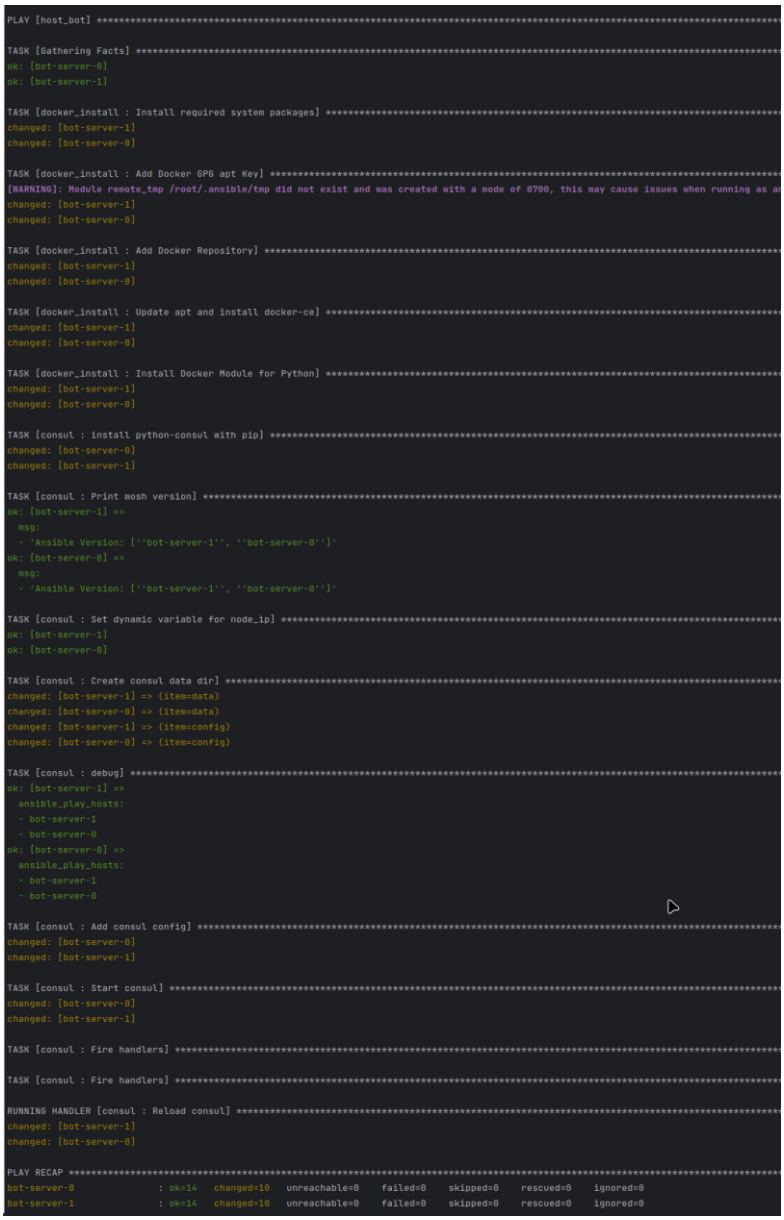


Рис.4.4 Результат виконання плейбука precondition

Після того як ми підготували сервери до запуску, можемо запускати тести навантаження. Розпочнемо з 50 потоків на кожний контейнер з рампом 300 секунд. Це означає що в кожному контейнері всі потоки рівномірно запустяться протягом 5 хвилин. Також це можна побачити і проконтролювати на метриках. Після того як запуститься все навантаження нам потрібно буде дати тестам попрацювати приблизно 20 хвилин і проаналізувати метрики, як з наших тестів навантаження, так і з застосунка в графані. Щоб запустити тести навантаження нам потрібно запустити плейбук bot_start командою ansible-playbook -i inventory bot_start.yml. Показано на рис.4.5.

```
PLAY [host_bot] ****************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************
ok: [bot-server-0]
ok: [bot-server-1]

TASK [bot_start : Print mosh version] ******************************************************************************************
ok: [bot-server-1] =>
  msg:
  - 'Ansible Version: [''bot-server-1'', ''bot-server-0'']'
ok: [bot-server-0] =>
  msg:
  - 'Ansible Version: [''bot-server-1'', ''bot-server-0'']'

TASK [bot_start : Start bot container] *****************************************************************************************
changed: [bot-server-0]
changed: [bot-server-1]

PLAY RECAP *********************************************************************************************************************
bot-server-0               : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
bot-server-1               : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Рис.4.5 Результати виконання плейбука bot_start

Після виконання плейбука bot_start перейдемо в графану на дашбоард по гатлінгу. Тут ми маємо побачити метрики з тестів навантаження. Показано на рис.4.6.

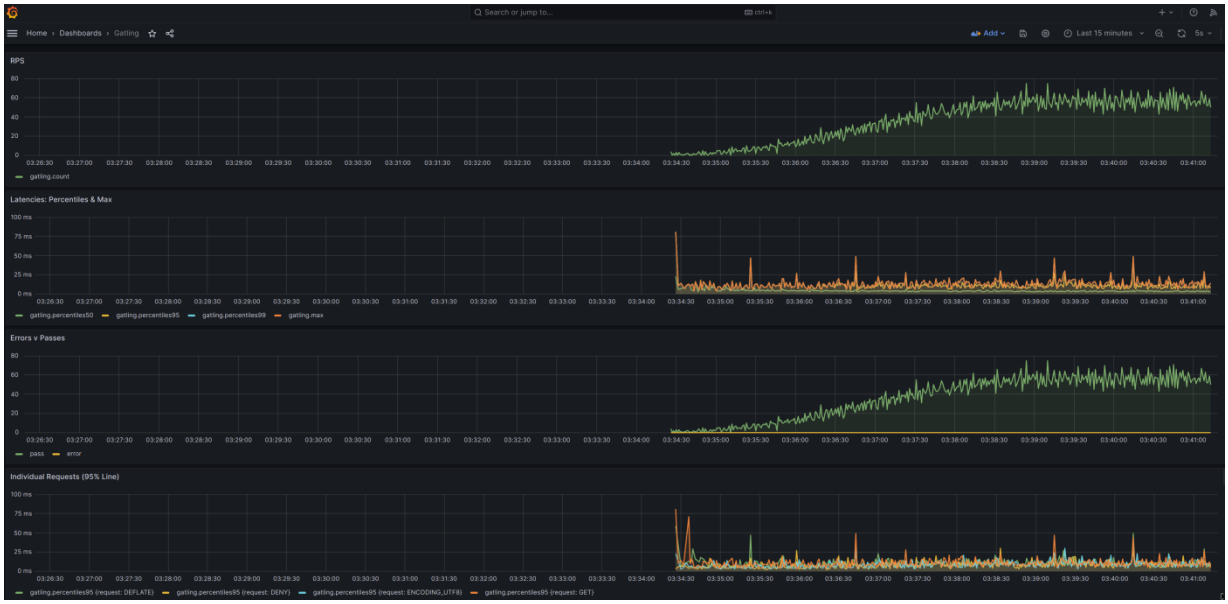


Рис.4.6 Метрики з тестів навантаження

Як ми бачимо у нас навантаження піднялося за 5 хвилин, запущений перший сценарій і присутні метрики по таймінгах кожного реквеста. Тепер нам потрібно потримати навантаження прибизно 20 хвилин, і проаналізувати більш детально метрики по кожному реквесту та метрики з сервера на якому запущений наш застосунок.

Проаналізуємо метрики з наших тестів навантаження за 20 хвилин їх роботи рис. 4.7.

Рис.4.7. Метрики з гатлінгу за 20 хвилин роботи тестів навантаження

Як ми бачимопо по графіку RPS тести навантаження роблять загалом 60 запитів в секунду. Також на графіку Latencies: Percentiles & Max ми можемо побачити максимальні затримки по кожному персентилю, і як показано вони у нас не більші 25 мс. Крім того у нас присутній графік кількості реквестів які пройшли з помилками і без. При даному навантаженні як ми бачимо у нас запити з помилками відсутні. Останній графік показує час респонсу на 95 персентилі по кожному з реквестів окремо. Проаналізуємо даний графік більш детально. На рис.4.7 показано таймінги по реквесту DEFLATE, на рис.4.8 по ревесту DENY, на рис.4.9 по ревесту ENCODING_UTF8, на рис.4.10 по ревесту GET.

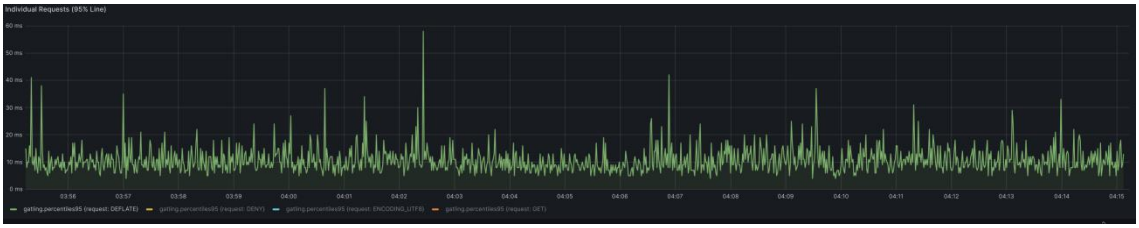


Рис.4.8 Таймінги по реквесту DEFLATE

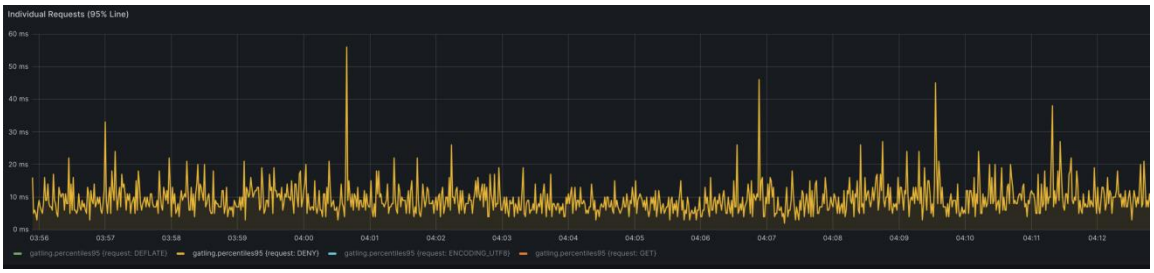


Рис.4.9 Таймінги по ревесту DENY

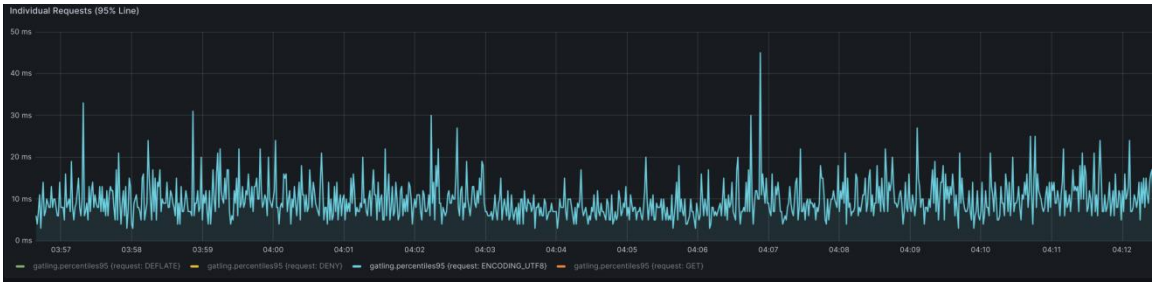


Рис.4.10 Таймінги по ревесту ENCODING_UTF8

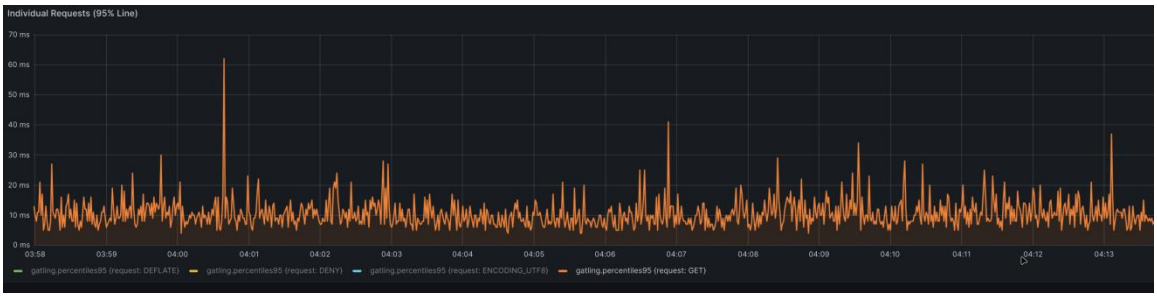


Рис.4.11 Таймінги по ревесту GET.

Як ми бачимо з даних метрик час на обробку нашиї реквестів при 60 запитах в секунду складає в середньому 10-15 мс. Запити які виконуються з помилками відсутні, також у нас є дашбоарти, де ми можемо розглянути більш детальні метрики по кожному запиту і в загальному по всьому сценарію тестування.

Розглянемо дашбоард Gatling Report. Це більш розширений дашбоард і він дам дозволяє вивести загальну інформацію про симуляцію, реквести які ми викликаємо в даній симуляції, загальну кількість по кожному запиту, кількість запитів з помилками і без по кожному запиту. Також ці всі параметри ми можемо подивитися по різних персентилях. Глобальна інформація по симуляції показана на рис.4.12.

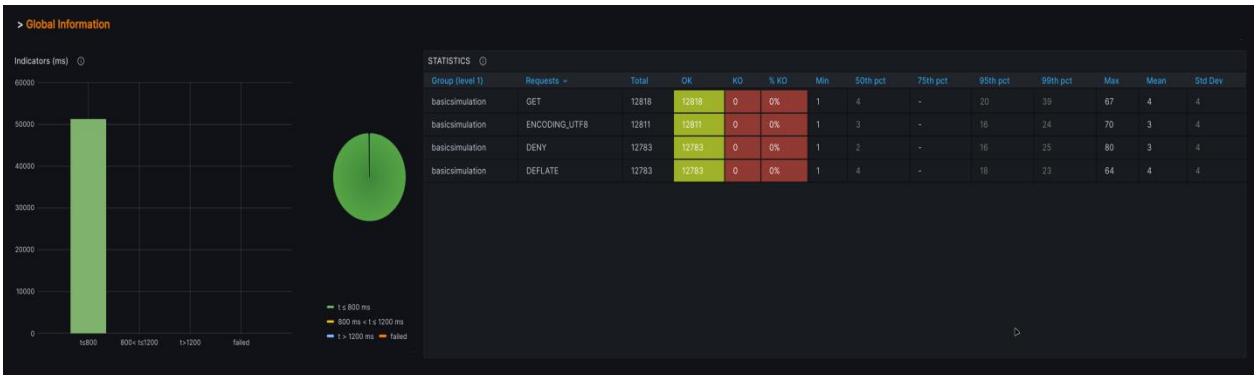


Рис.4.12 Глобальна інформація по симуляції

На даному дашбоарді ми можемо перевірити таймінги по кожному реквесту окремо на різних персентилях. Показано на рис.4.13.

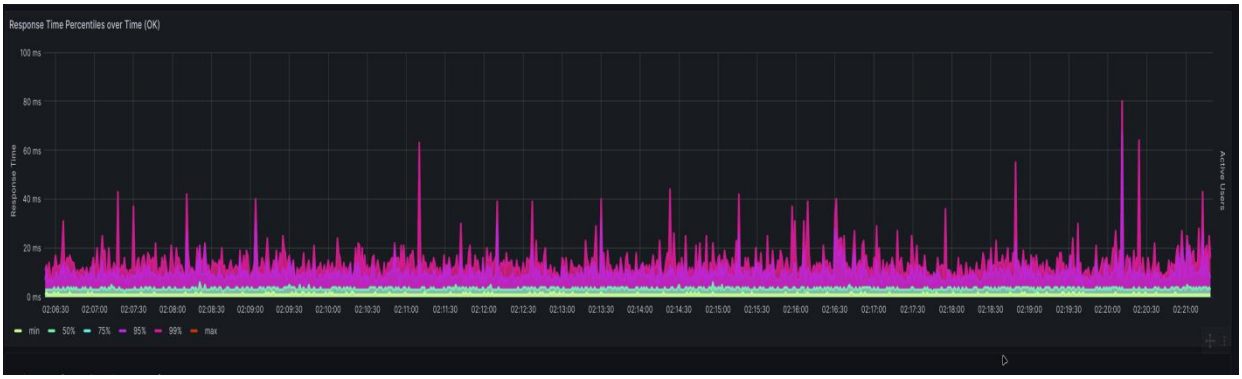


Рис.4.13 Таймінги по різних персентилях

Окрім цього у нас є метрика на якій виводиться кількість реквестів відправлених в секунду. Показано на рис.4.14.

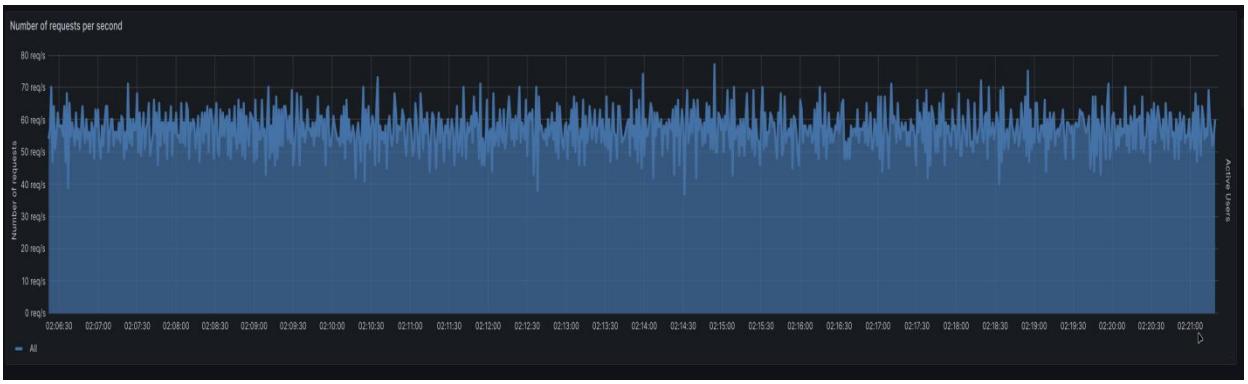


Рис.4.15 Кількість реквестів відправлених в секунду

І остання метрика яка є на цьому дашбоарді - це кількість респонсів які прийшли клієнту в секунду. Показано на рис.4.15.

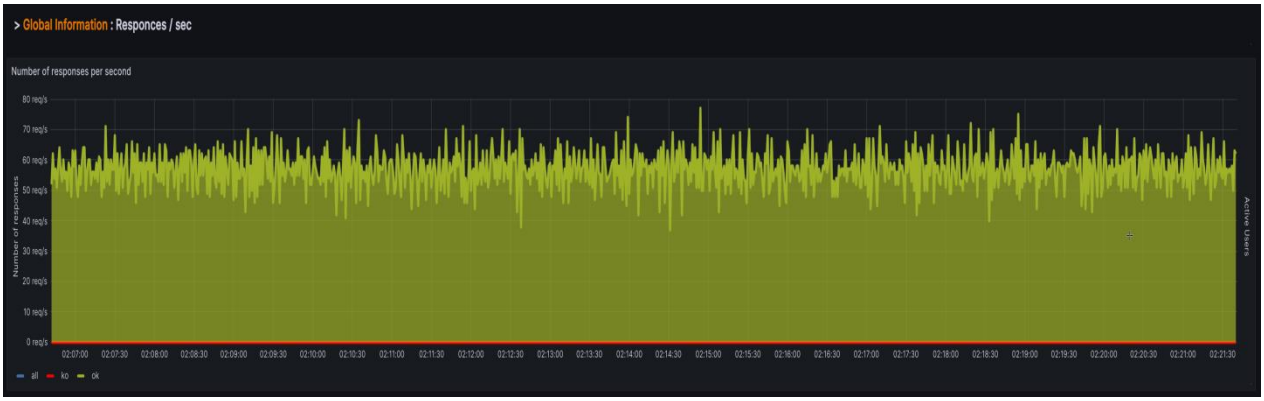


Рис.4.15 Кількість респонсів в секунду

Всі ці метрики у нас виводяться по кожному запиту окремо, відповідно ми можемо проаналізувати все більш детально.

Ще ми маємо додатковий дашбоард Gatling Report Trend, який показує загальну інформацію по нашій симуляції, а саме час тестування, кількість реквестів з помилками і без, час запуску симуляціх, максимальні і мінімальні таймінги. Дана інформація виводиться по різним персентилям і дає нам дуже узагальнену інформацію. Частіше всього використовується для формування репортів по тестуванню. Показано на рис. 4.16

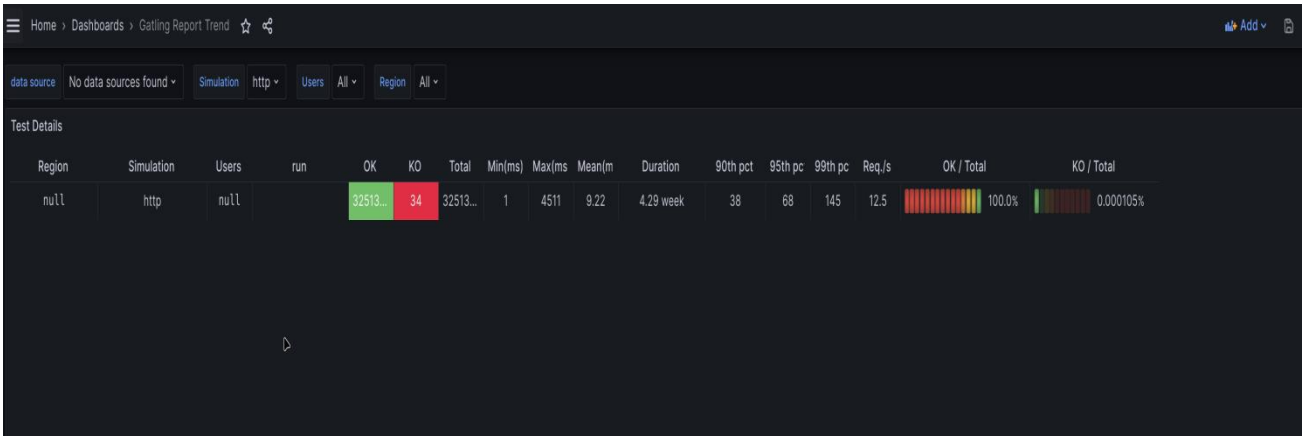


Рис.4.16 Дашбоард Gatling Report Trend

Представлені дашбоарди і метрики виводяться з клієнта і допомагають проаналізувати нам роботу нашого застосунку і як його робота впливатиме на

користувачів. Також у нас є ще один дашбоард - Application, який виводить метрики з сервера на якому розгорнутий застосунок. Показано на рис.4.17

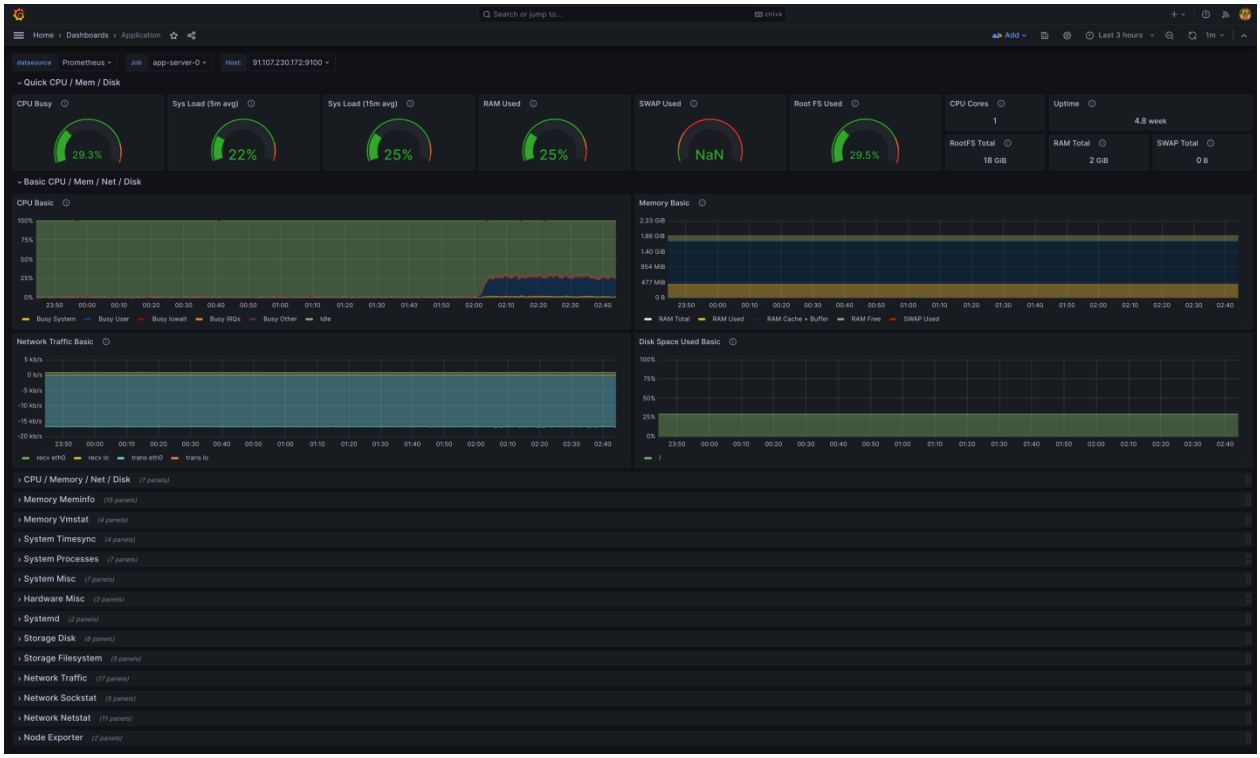


Рис.4.17 Дашбоард Application

Як бачимо на даному дашбоарді у нас присутні всі потрібні нам метрики щоб проаналізувати роботу нашого аплікейшина при потрібному нам навантаженні. Тут відображаються основні характеристики сервера, метрики по споживанню цпу, памяті, навантаження на мережу, диски, що є цілком достватньо аби зробити висновки як працює аплікейшин з кожною зміною версії коду, і як це веливає на споживання ресурсів, адже ми завжди можемо порівняти метрики до і після деплоя кожної нової версії.

Наступним кроком проаналізуємо роботу аплікейшину. Нас тут цікавлять 2 основні метрики - це споживання цпу і памяті. На даний момент ми відправляємо в середньому 60 реквестів в секунду. Цпу у нас навантажено майже на 30 відсотків, що є показано на рис.4.17, значить ми ще маємо великий запас, і можемо піднімати навантаження на наш застосунок. По споживанню памяті трохи гірша ситуація. При такому навантаженні у нас вільно 166 мегабайт оперативної памяті з 1.8 гігабайта

доступної, показано на рис. 4.18. Оскільки маємо великий запас по цпу, то можна підняти навантаження і перевірити як наш аплікейшин працює при вищому лоаді.

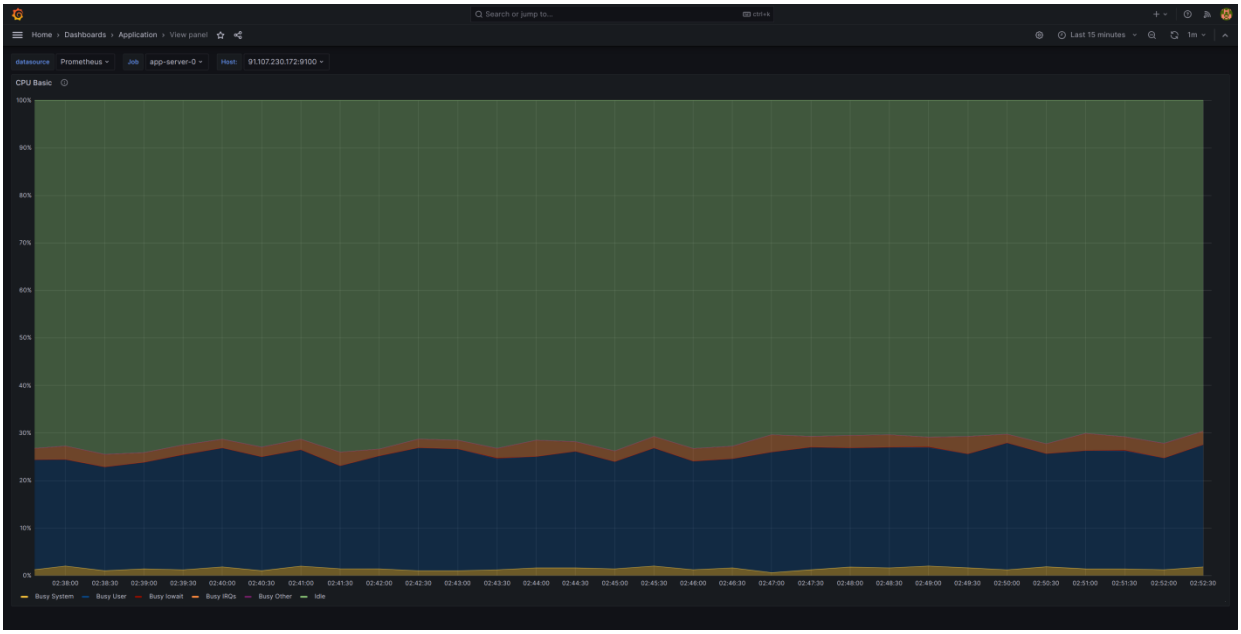


Рис.4.18 Споживання цпу

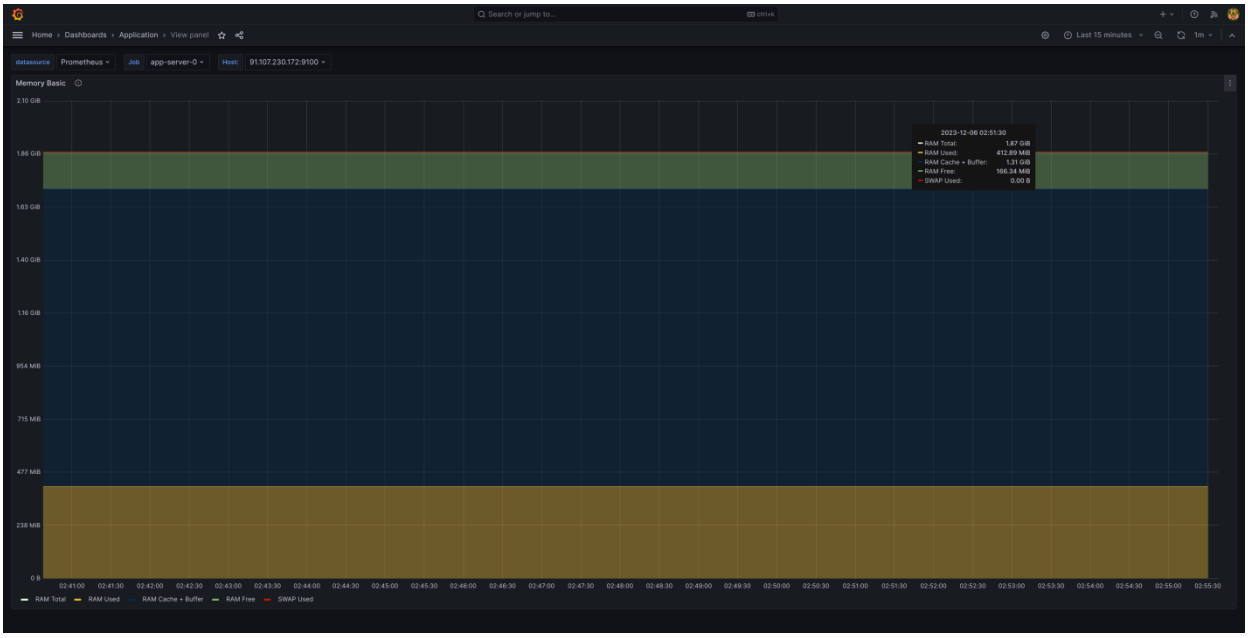


Рис.4.19 Споживання памяті

Давайте збільшимо кількість серверів на яких у нас запускаються тести навантаження. Також збільшимо кількість потоків в кожному сервері ще по 150. Загалом у нас буде запущено 600 потоків з 4 серверів і подивимось як це вплине на

робооту нашого застосунку. Аби здійснити цю перевірку наше вантаження не потрібно зупиняти, ми просто в файлі variables.tf встановлюємо кількість інстансів серверів рівною 4, і знову виконуємо команду terraform apply. Таким чином тереформ перевірить які нам потрібні ще ресурси і створить їх. Показано на рис.4.20.

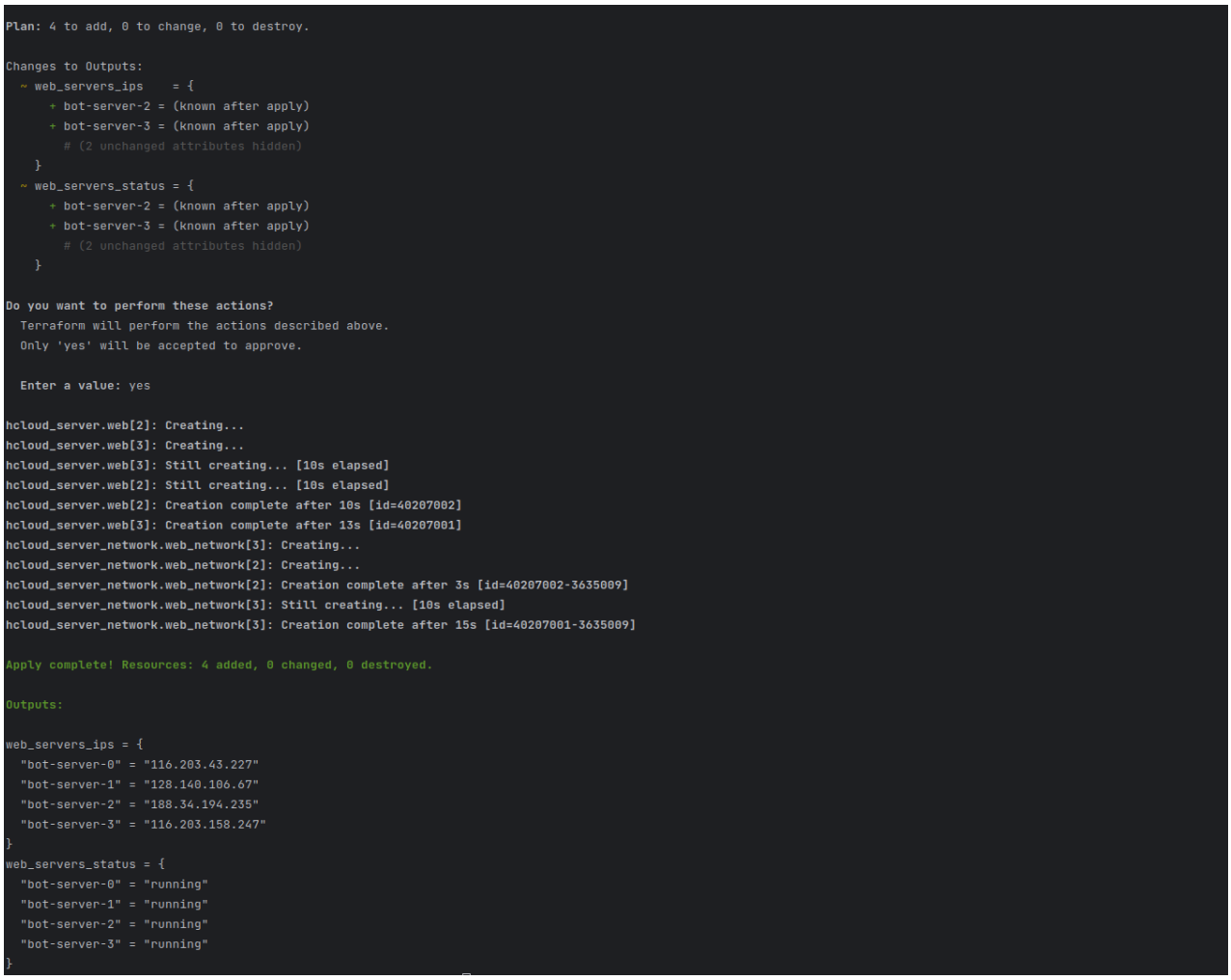


Рис. 4.20 Результат виконання команди terraform apply

Як ми бачимо тераформ не видалив попередньо створені сервери і навантаження не зупинилось, та тести навантаження продовжують працювати далі. Після цього нам потрібно запустити плейбук для налаштування серверів командою ansible-playbook -i inventory precondition.yml, який налаштує 2 нових сервера. Результати виконання плейбука показано на рис. 4.21.

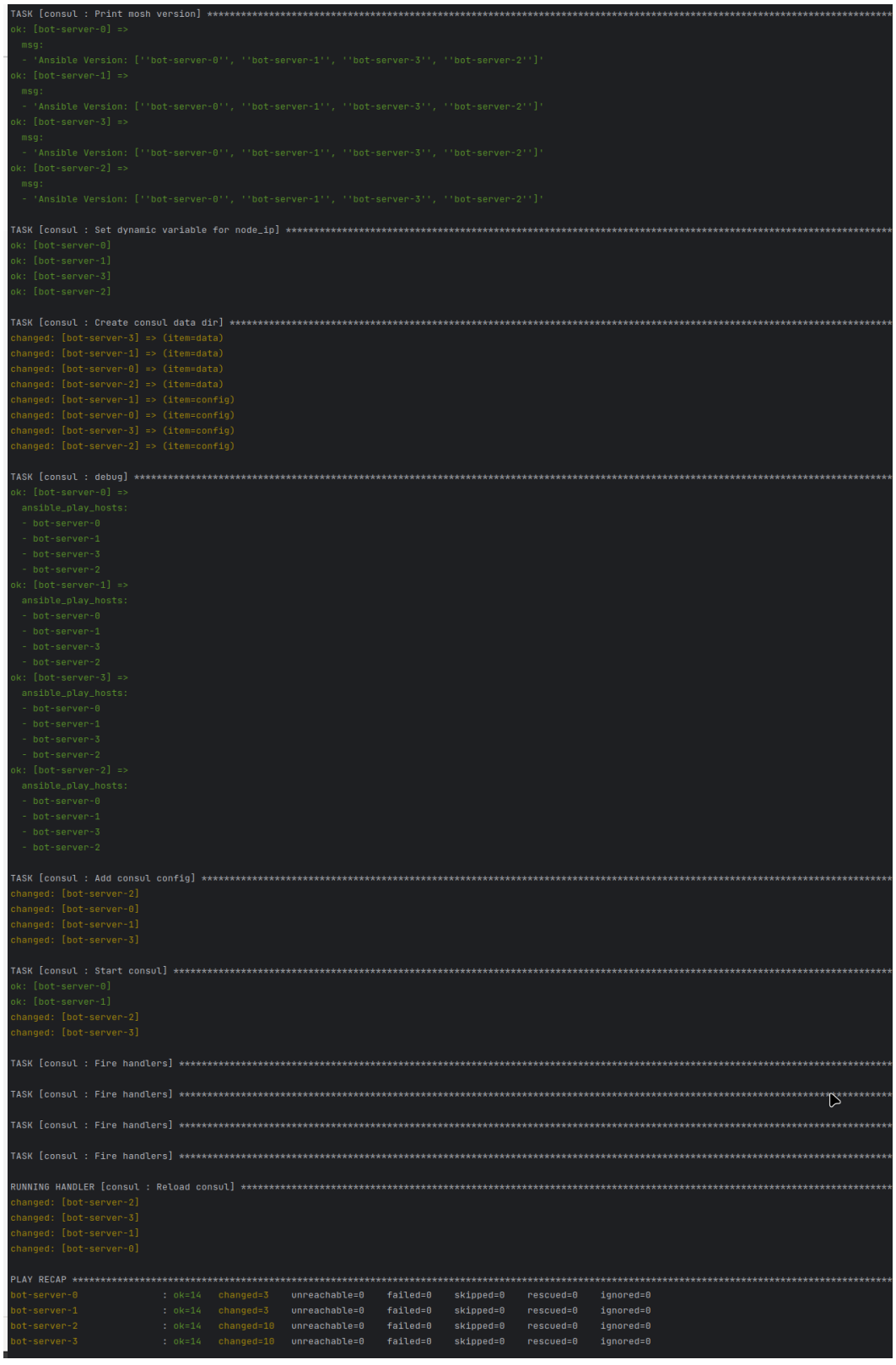


Рис.4.21 Результати виконання плейбука precondition

Наступним кроком нам потрібно збільшити кількість потоків тестів навантаження на кожному з серверів. Для цього маємо змінити параметр

usersPerContainer: "150". Це означає що у нас за 5 хвилин в кожному сервері запуститься 150 потоків. Оскільки параметр during: "300" у нас встановлений 300 секунд, показано на рис.4.22, після цього потрібно запустити плейбук bot_start командою ansible-playbook -i inventory bot_start.yml, результат виконання якої показаний на рис.4.23.

```yaml
---
- name: Print mosh version
  debug:
    msg:
      - 'Ansible Version: {{ ansible_play_hosts }}'

- name: Start bot container
  docker_container:
    name: "gatling-bot{{ emul_prefix | default('') }}"
    image: "aurora0/bot:latest"
    env:
      GrafanaPostfix: "gatling"
      propertiesFile: "diplom"
      scenario: "http1"
      during: "300"
      usersPerContainer: "150"
      app: "{{hostvars['app-server-0'].ansible_host}}:8000"
      consul: "172.17.0.1"
    volumes:
    #      - /tmp/gatling/gatling.conf:/opt/gatling/conf/gatling.conf
    #      - /var/log/gatling:/opt/gatling/results
    state: started
    pull: true
    command: mvn -X gatling:test -Dgatling.simulationClass=simulation.BasicSimulation -Dgatling.conf.file=gatling-configs/gatling.conf
    log_driver: json-file
    log_options:
      max-size: "500m"
      max-file: "1"

```

Рис.4.22 Параметри плейбука bot_start

```
PLAY [host_bot] ****************************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************
ok: [bot-server-2]
ok: [bot-server-1]
ok: [bot-server-3]
ok: [bot-server-0]

TASK [bot_start : Print mosh version] ******************************************************************************************************
ok: [bot-server-0] =>
  msg:
  - 'Ansible Version: [''bot-server-0'', ''bot-server-1'', ''bot-server-3'', ''bot-server-2'']'
ok: [bot-server-1] =>
  msg:
  - 'Ansible Version: [''bot-server-0'', ''bot-server-1'', ''bot-server-3'', ''bot-server-2'']'
ok: [bot-server-3] =>
  msg:
  - 'Ansible Version: [''bot-server-0'', ''bot-server-1'', ''bot-server-3'', ''bot-server-2'']'
ok: [bot-server-2] =>
  msg:
  - 'Ansible Version: [''bot-server-0'', ''bot-server-1'', ''bot-server-3'', ''bot-server-2'']'

TASK [bot_start : Start bot container] *****************************************************************************************************
changed: [bot-server-1]
changed: [bot-server-0]
changed: [bot-server-2]
changed: [bot-server-3]

PLAY RECAP *********************************************************************************************************************************
bot-server-0               : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
bot-server-1               : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
bot-server-2               : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
bot-server-3               : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Рис.4.23 Результат виконання команди ansible-playbook -i inventory bot_start.yml

Дана команда зупинить контейнер з тестами навантаження на серверах і запустить його з новими параметрами. Відповідно це ми і побачимо на наших метриках. Показано на рис.4.24

Рис.4.24 Перезапуск тестів навантаження

Як ми бачимо з наших метрик таймінги у виросли в два рази, кількість реквестів в секунду збільшилась в 4 рази, споживання цпу збільшилось до 87 відсотків, показано на рис.4.25, і це означає що максимально наш застосунок при даних характетистиках серера, показано на рис. 4.26, витримує майже 200 реквестів в секунду. Споживання памяті без змін.

Рис.4.25 Споживання цпу і памяті

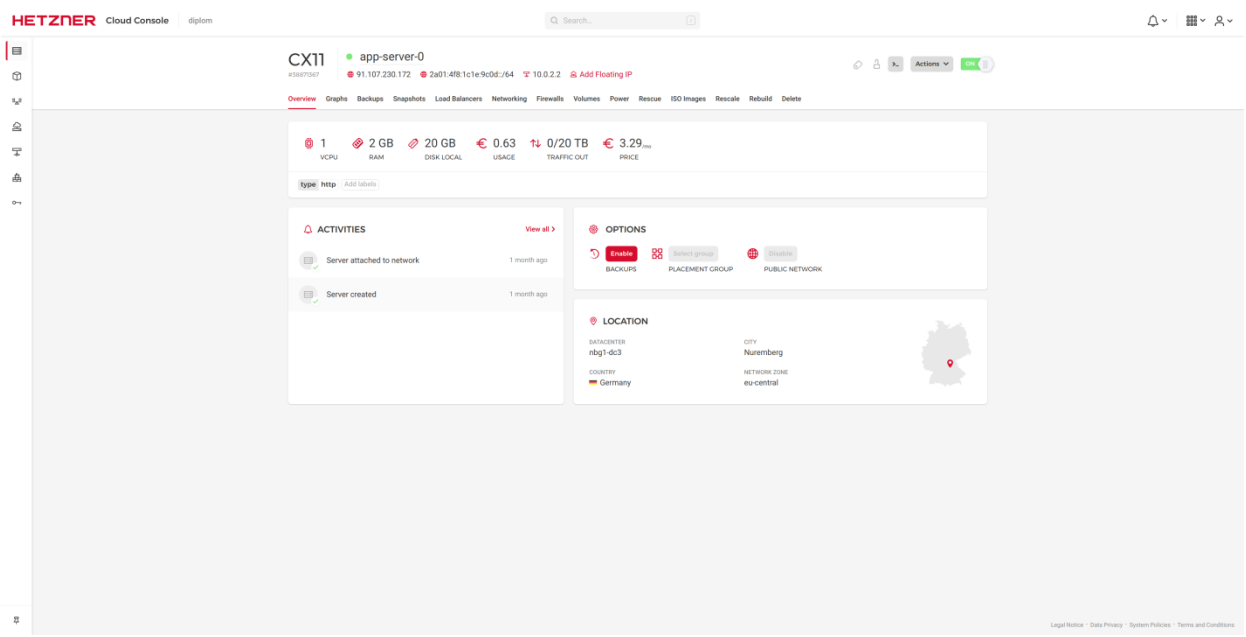


Рис.4.26 Характеристики сервера для застосунку

Якщо нам потрібно щоб застосунок витримував більше навантаження то можливо змінити тип інстансу на більш потужний(рис. 4.27), або додати кілька інстансів з такими ж параметрами, а перед ними встановити балансувальник трафіку який буде розділяти навантаження між серверами.

Рис.4.27 Параметри інстансів на хецнер клауді

На даному етапі тестування першого сценарію можна вважати завершеним. Ми визначили максимальну кількість потоків і реквестів яку витрримує наш застосунок.

розгорнутий на даному типі заліза і знапропонували можливі варіанти покращення для збільшення продуктивності застосунку.

4.2 Запуск тестів навантаження і аналіз результатів за другим сценарієм

Переходимо до сценарію HTTP2. Судячи з тестування попереднього сценарію тестів навантаження для повного намантаження нам потрібно буде 4 сервери на яких ми будем запускати тести. Для цього в файлі variables.tf в параметр instances встановлюємо 4, після чого запускаємо команду terraform apply, яка запустить наш тераформ скрипт і створить відповідні нам ресурси. Результати виконання команди показані на рис.4.28.

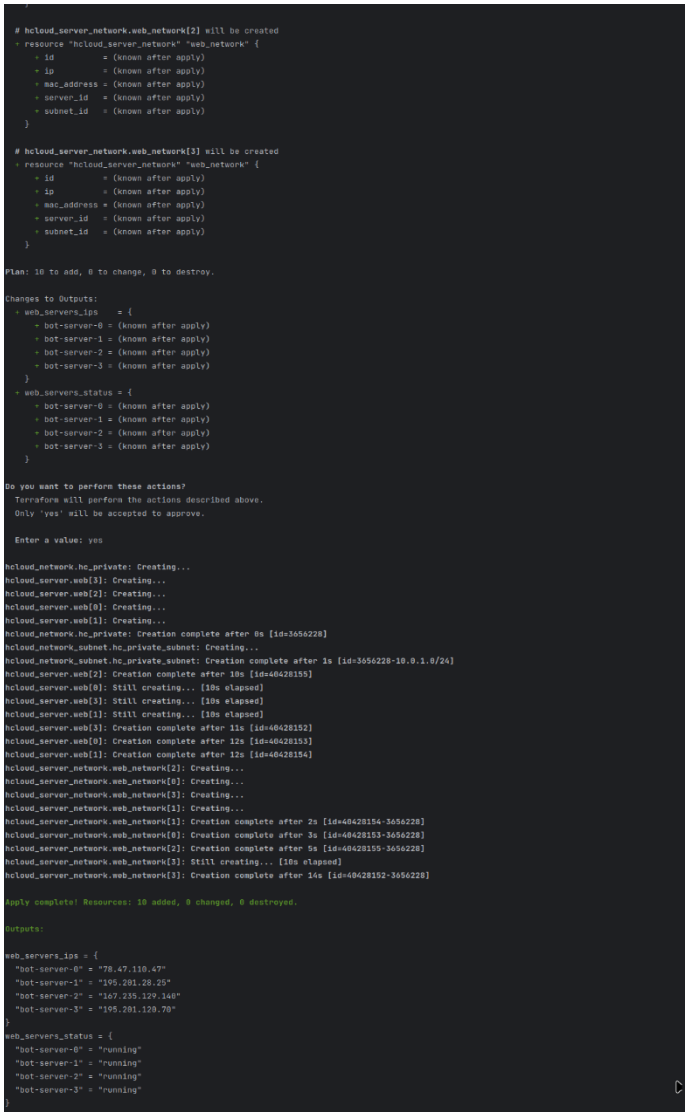


Рис.4.28 Результат виконання команди terraform apply

Перейдемо на клауд і перевіримо чи наші сервери створилися. Показано на рис. 4.29

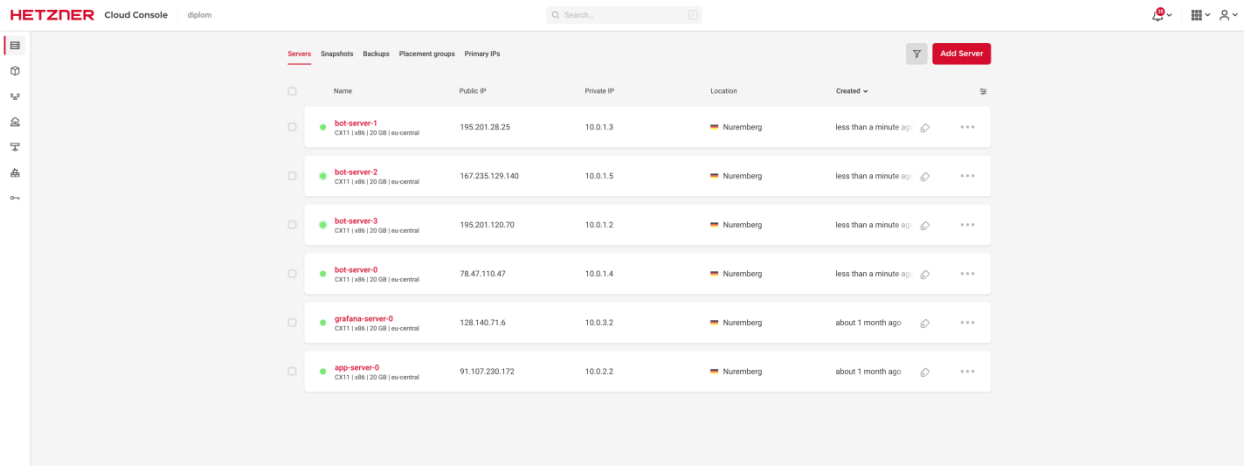


Рис. 4.29 Список серверів на клауді

Як бачимо наші сервери створені. Далі нам потрібно їх підготувати до запуску тестів навантаження, запустивши наступну команду - ansible-playbook -i inventory precondition.yml. Плейбук precondition проінсталює докер, консул, і налаштує його з усіма потрібними нам залежностями. Показано на рис. 4.30.

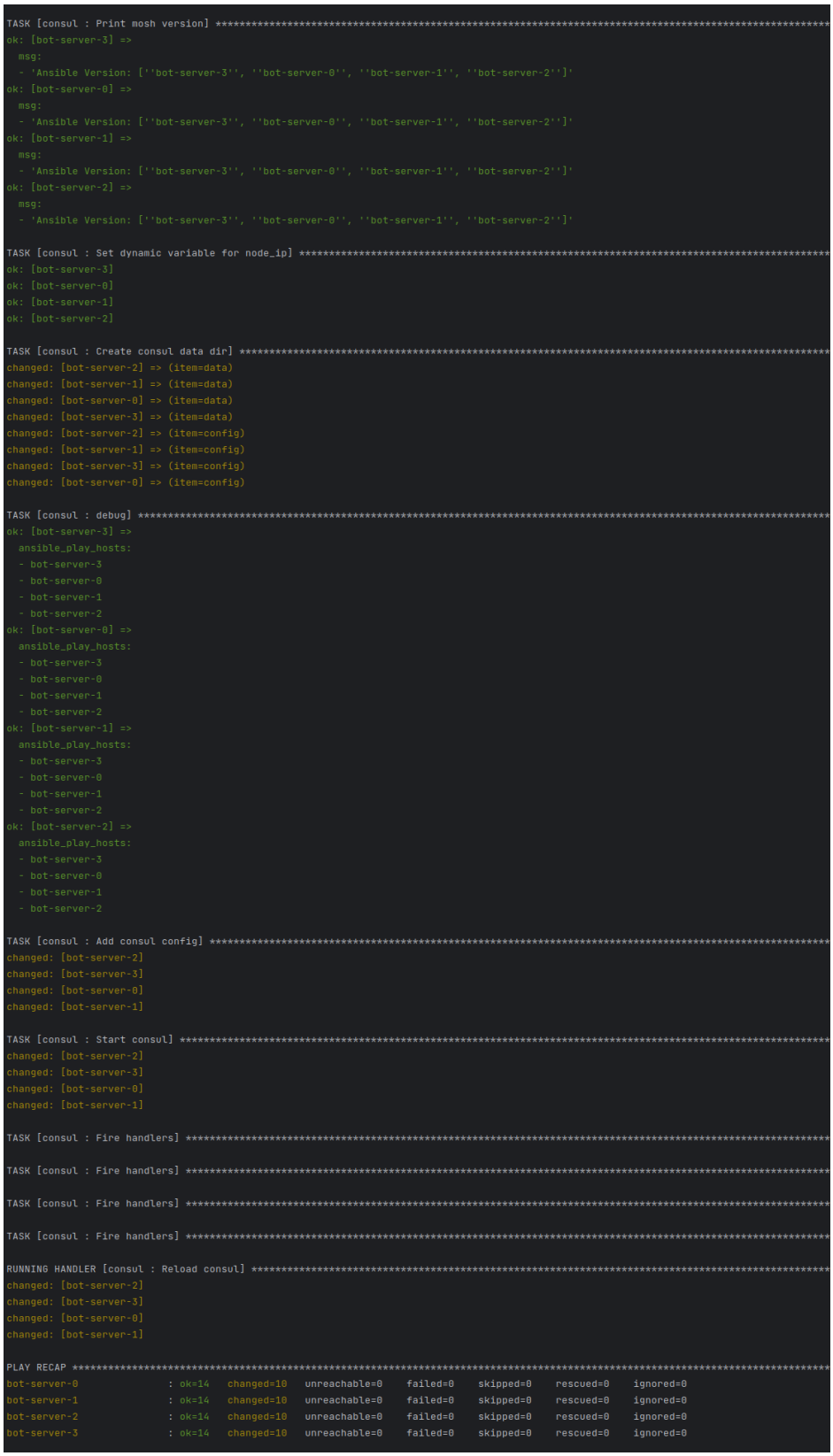


Рис.4.30 Виконання плейбуку precondition

Як бачимо плейбук запустився на всіх 4 серверах. Тепер перевіримо чи працює консул в якому має бути зареєстровано 4 ноди, що означатиме що кластер консула працює вірно, показано на рис. 4.31.

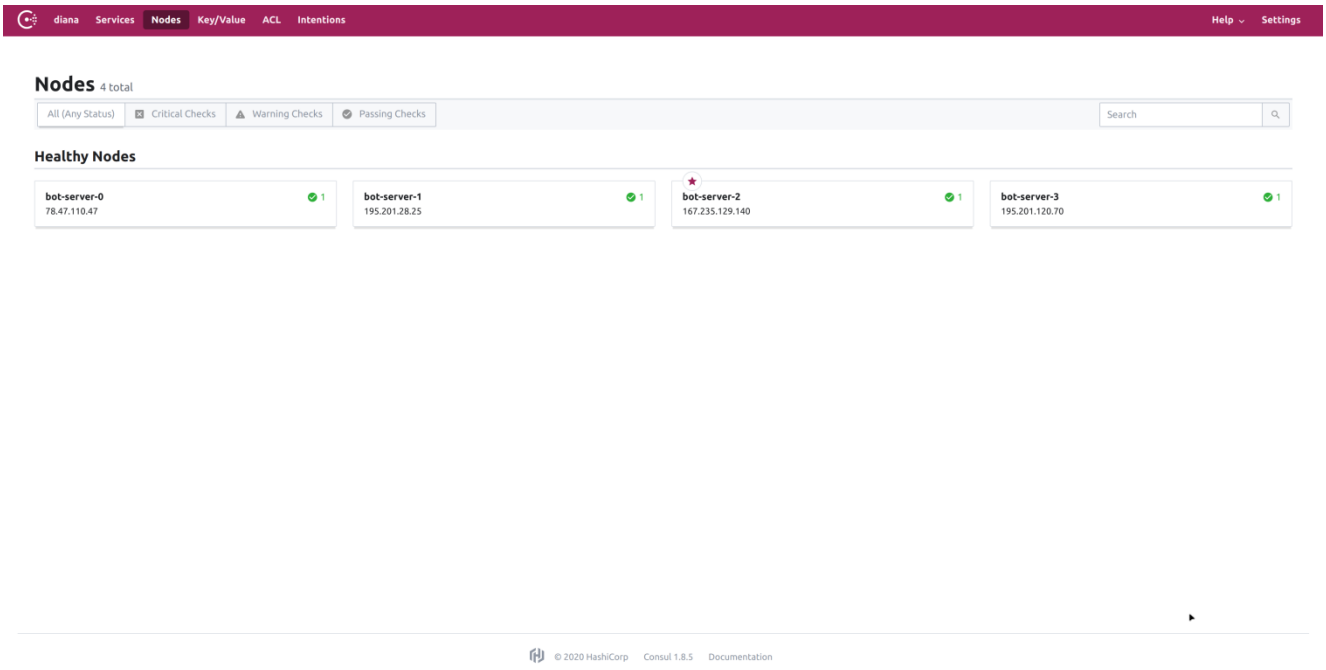


Рис. 4.31 Список нод в консулі

Як бачимо всі ноди зареєстровані та на них пройшли всі хелс чеки, а значить наші сервери готові дозапуску тестів навантаження. Змінимо наш сценарій запуску, оскільки ми будемо запускати HTTP2 вперше. В попередньому сценарії ми запускали навантаження на всіх доступних серверах відразу, а зараз будемо запускати навантаження по черзі на кожному сервері окремо, при цьому прослідкуємо зміни в навантаженні на цпу і споживанні озу на сервері з аплікейшином. Для цього нам потрібно для початку перевірити інвенторі і переконатися що всі сервери доступні. Для цього виведемо граф нашого інвенторі, показано на рис. 4.32.

```
@all:
  |--@ungrouped:
  |--@hcloud:
  |  |--app-server-0
  |  |--grafana-server-0
  |  |--bot-server-3
  |  |--bot-server-0
  |  |--bot-server-1
  |  |--bot-server-2
  |--@host_http:
  |  |--app-server-0
  |--@host_grafana:
  |  |--grafana-server-0
  |--@host_bot:
  |  |--bot-server-3
  |  |--bot-server-0
  |  |--bot-server-1
  |  |--bot-server-2
```

Рис. 4.32 Інвенторі

Як бачимо всі сервери доступні і можемо запустити навантаження на першому сервері. Для цього нам потрібно налаштувати плейбук bot_start.yml, змінивши scenario на http2, during залишимо 300, і кількість юзерів usersPerContainer 150, показано на рис. 4.33.

```
---
- name: Print mosh version
  debug:
    msg:
      - 'Ansible Version: {{ ansible_play_hosts }}'

- name: Start bot container
  docker_container:
    name: "gatling-bot{{ emul_prefix | default('') }}"
    image: "aurora0/bot:latest"
    env:
      GrafanaPostfix: "gatling"
      propertiesFile: "diplom"
      scenario: "http2"
      during: "300"
      usersPerContainer: "150"
      app: "{{hostvars['app-server-0'].ansible_host}}:8000"
      consul: "172.17.0.1"
    volumes:
    #      - /tmp/gatling/gatling.conf:/opt/gatling/conf/gatling.conf
    #      - /var/log/gatling:/opt/gatling/results
    state: started
    pull: true
    command: mvn -X gatling:test -Dgatling.simulationClass=simulation.BasicSimulation -Dgatling.conf.file=gatling-configs/gatling.conf
    log_driver: json-file
    log_options:
      max-size: "500m"
      max-file: "1"
```

Рис. 4.33 Параметри плейбука bot_start.yml

Плейбук запустимо наступною командою - ansible-playbook -i inventory bot_start.yml -l bot-server-0. Результат показано на рис. 4.34. Як бачимо флажком -l ми можемо вказувати кількість серверів на яких будемо запускати тести навантеження. Якщо потрібно запустити відразу на двої і більше то сервери вказуємо через кому.

```
PLAY [host_bot] ****************************************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************************
ok: [bot-server-0]

TASK [bot_start : Print mosh version] ******************************************************************************************************************
ok: [bot-server-0] =>
  msg:
  - 'Ansible Version: [''bot-server-0'']'

TASK [bot_start : Start bot container] *****************************************************************************************************************
changed: [bot-server-0]

PLAY RECAP *********************************************************************************************************************************************
bot-server-0               : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Рис. 4.34 Результат виконання команди ansible-playbook -i inventory bot_start.yml -l bot-server-0

Як ми бачимо тести навантаження запустились на одному сервері. Тепер перейдемо в графану і перевіримо наші метрики з гатлінгу, показано на рис. 4.35

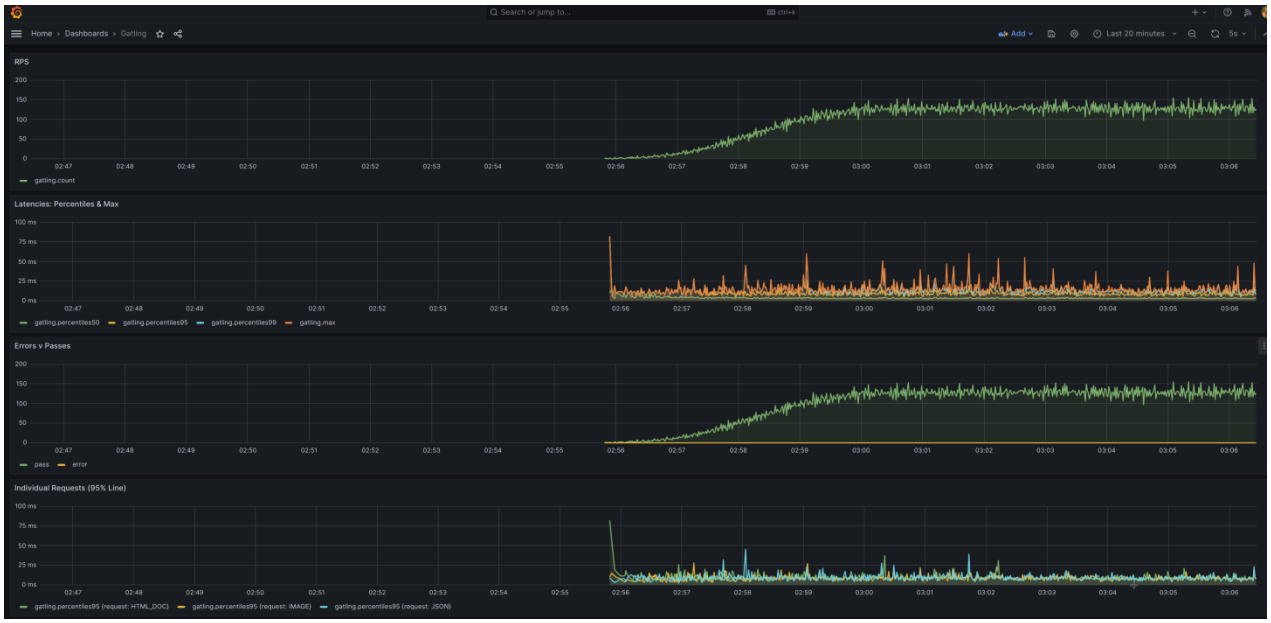


Рис. 4.35 Метрики з гатлінгу

Як ми бачимо навантаження запустилось і вийшло на пікове за 5 хвилин. Зачекаємо десь 15 хвилин і перевіримо на скільки сильно навантажує наш сервер застосунок при 150 користувачах. Таймінги на всі наші запити складають в середньому 10 мс, показано на рис. 4.36.

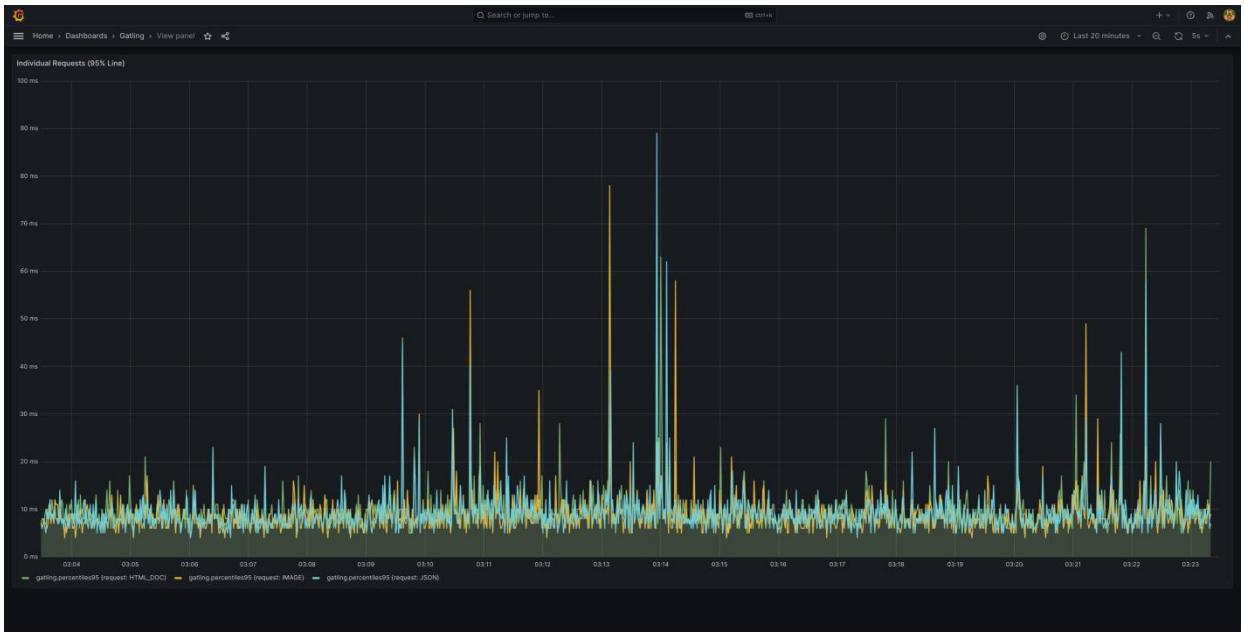


Рис.4.36 Таймінги на запити

Тепер перевіримо навантаження на цпу і споживання памяті нашим застосунком на сервері. Показано на рис. 4.37

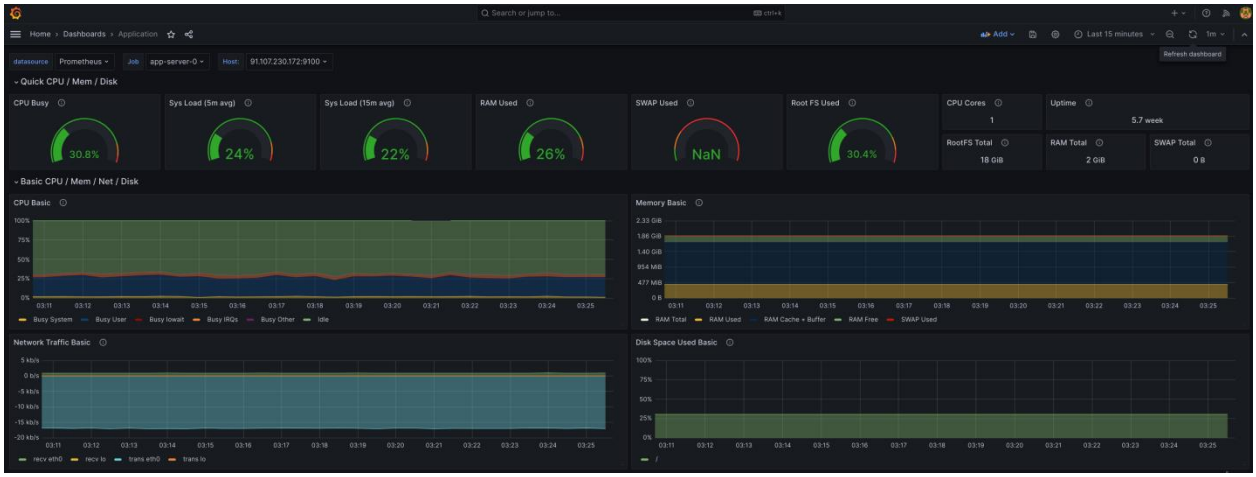


Рис.4.37 Метрики з серверу на якому запущений застосунок

Як ми бачимо цпу завантажене на 32 відсотки, память зайнята на 1.6 гігабайти(показано на рис.4.38). Сценарій HTTP2 значно більше завантажує ЦПУ. По пам’яті зміни відсутні, оскільки ми з одного сервера навантажили застосунок на 32 відсотки, а це 150 потоків. Далі запустимо ще 1 сервер на 150 потоків і порівняємо навантаження.

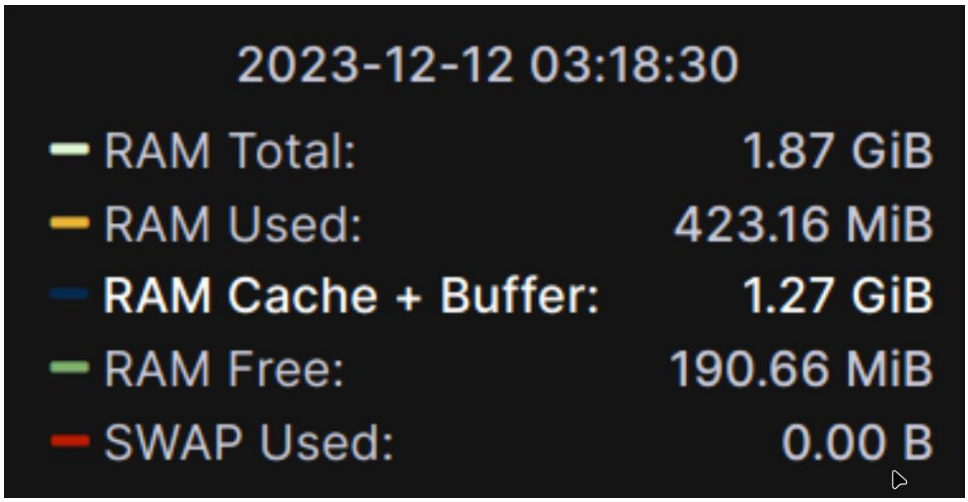


Рис. 4.38 Завантаження пам'яті

Після того як запустили додаткове навантаження, споживання цпу виросло до 52 відсотків, а споживання памяті залишилось на попередньому рівні. Показано на рис.4.39

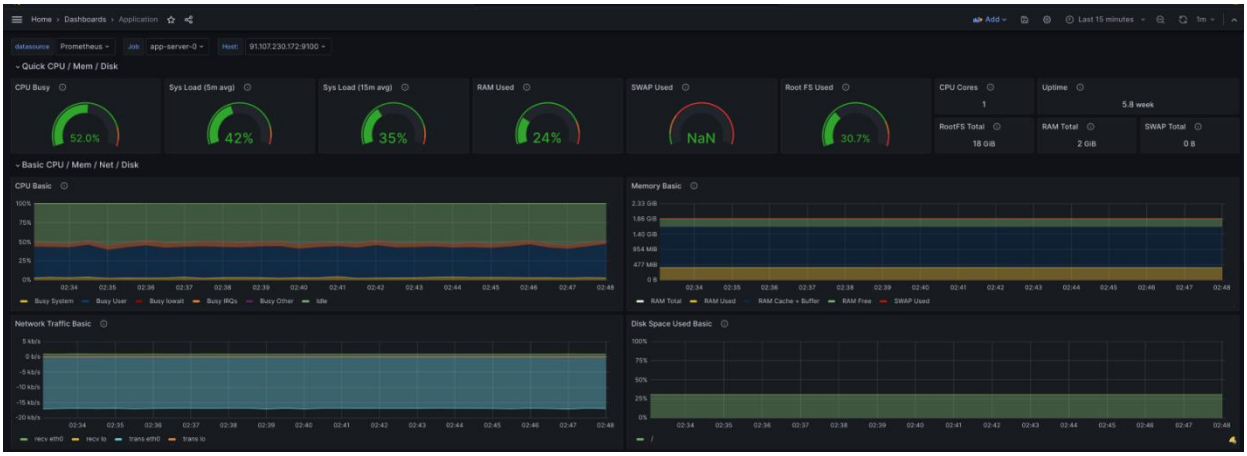


Рис. 4.39 Споживання цпу

Таймінги на запити виросли приблизно до 12 мс (показано на рис.4.37). Окрім цього у нас ще є доволі великий запас по ресурсах цпу і памяті. Запустимо ще 1 сервер навантаження, що збільшить кількість потоків на 150.

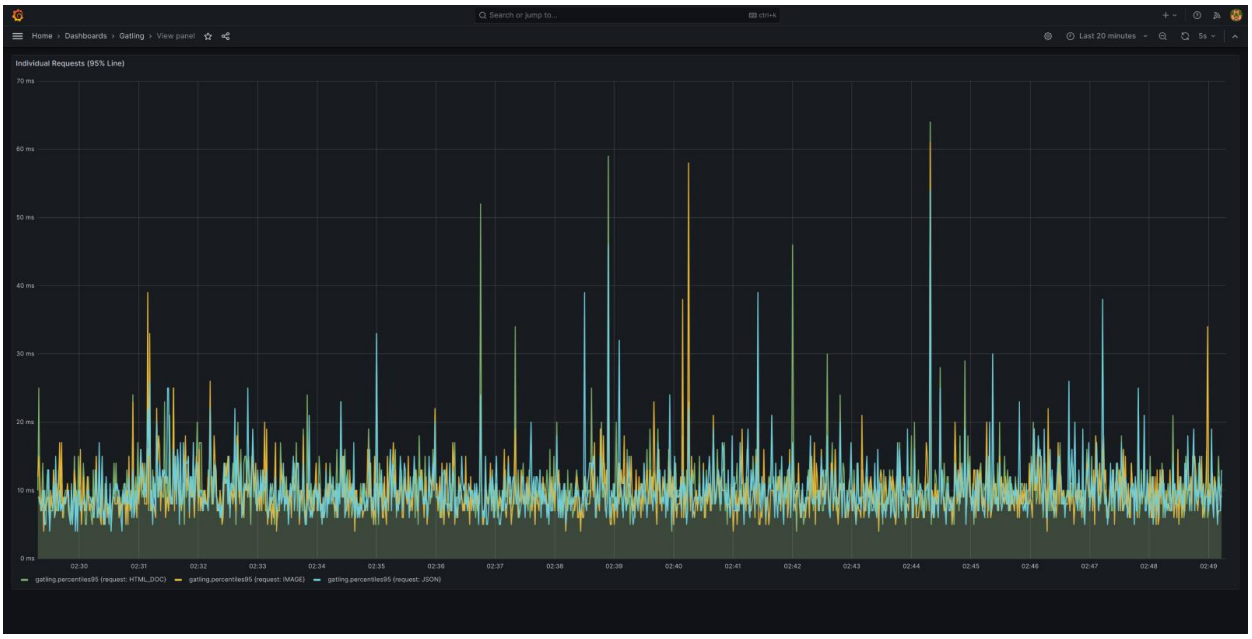


Рис. 4.40 Таймінги на запити

Після підняття навантаження до 450 потоків навантаження по цпу виросло до 59-62 відсотків (показано на рис 4.41), зміни по памяті відсутні. Спробуємо підняти ще на 150 потоків.

Рис. 4.41 Споживання цпу

Проаналізуємо метрики після підняття навантаження і розпочнемо з таймінгів (показано на рис.4.42). Таймінги у нас виросли приблизно на 3 мс, і становлять десь 15 мс в середньому.

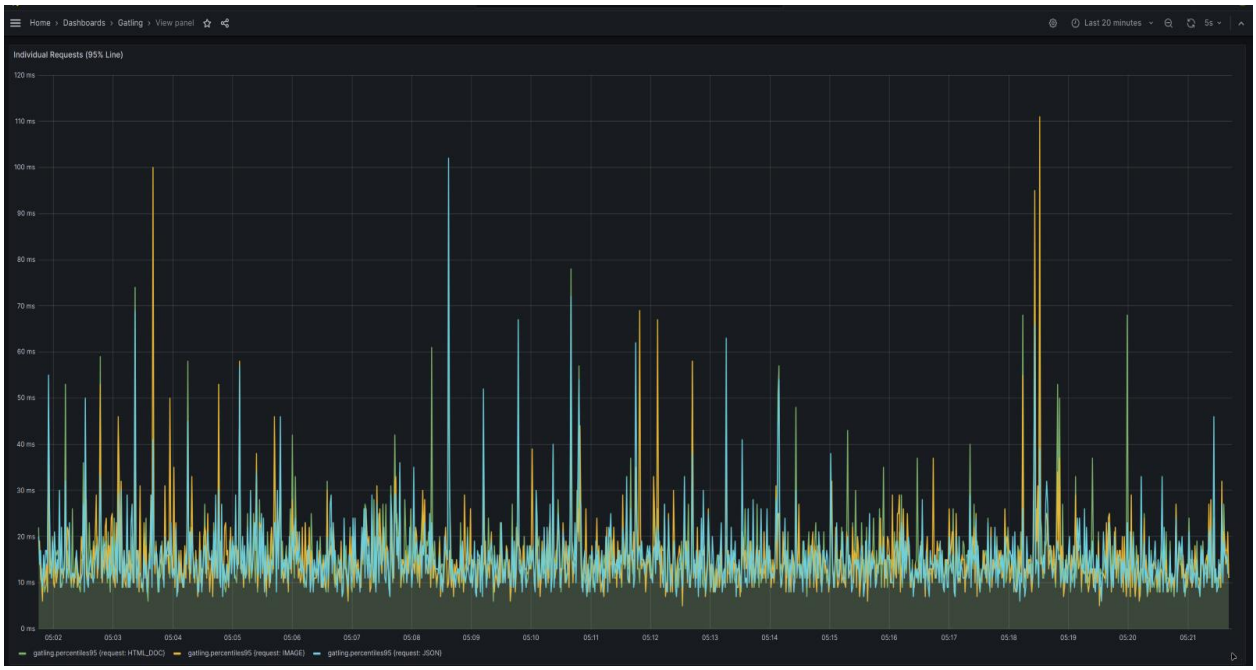


Рис.4.42 Таймінги на запити

Тепер проаналізуємо параметри самого сервера на якому запущений застосунок. Показано на рис. 4.43

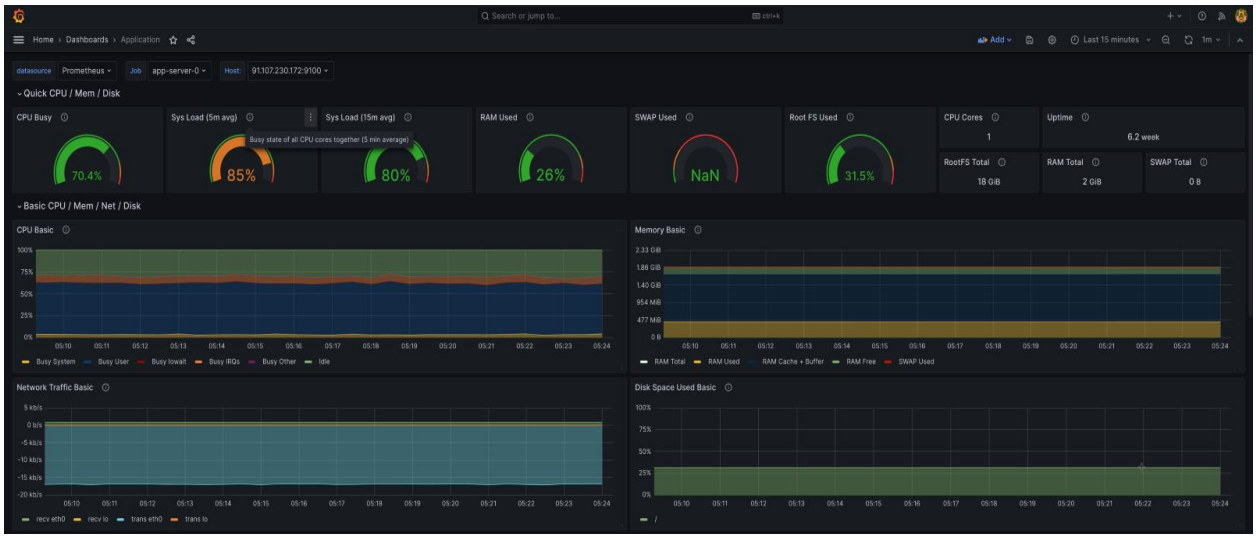


Рис. 4.43 Споживання цпу

Після підняття навантаження споживання цпу виросло до 70 відсотків. На відміну від першого сценарія при тестуванні якого при такій же кількості потоків навантаження становило на 12-15 відсотків більше, зміни по памяті фактично відсутні, а значить у нас є ще можливість збільшити навантаження. Для цього нам потрібно створити ще 2 сервери для запуску тестів навантаження наступним чином:

у файлі variables.tf кількість інстансів встановимо 6, після чого запустимо команду terraform apply, результат виконання якої показано на рис.4.44

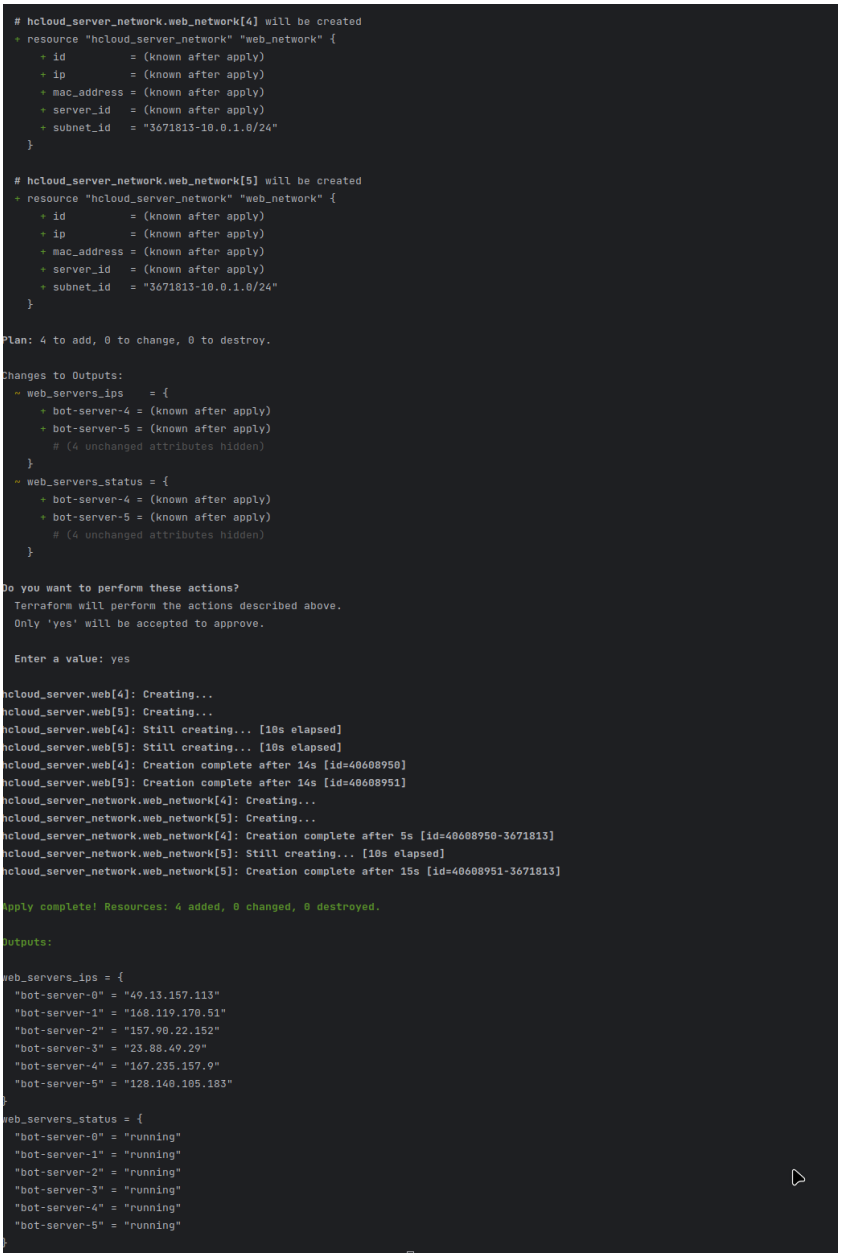


Рис.4.44 Результат виконання команди terraform apply

Як бачимо з логіки виконання команди terraform apply всі сервери створені. Наступним кроком нам потрібно сконфігурувати сервери для запуску, виконавши команду плейбук ansible-playbook -i inventory precondition.yml, результат виконання якої показано на рис.4.45. Ці маніпуляції ми робимо не зупиняючи тести навантаження, які ми запускали на попередніх серверах.

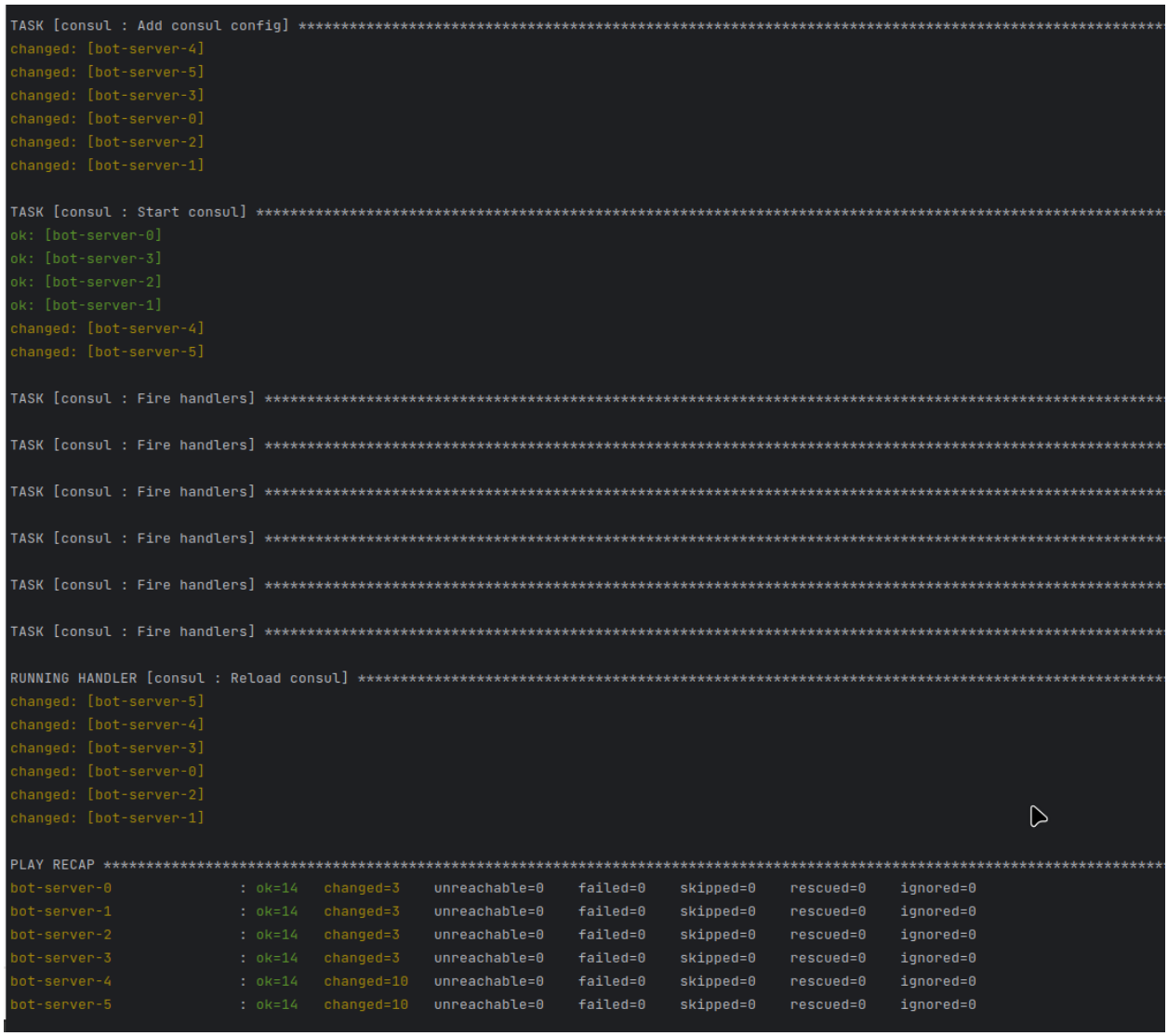


Рис.4.45 Результат виконання плейбуку precondition

Тепер ми маємо 2 додаткових сервера для запуску наших тестів навантаження. Запустимо тести навантаження на 5 сервері. Після того як ми запустили навантаження, таймінги піднялися в середньому до 20мс (показано на рис.4.46), і навантаження на цпу виросло до 76 відсотків (показано на рис. 4.47). Споживання памяті залишилось без змін.

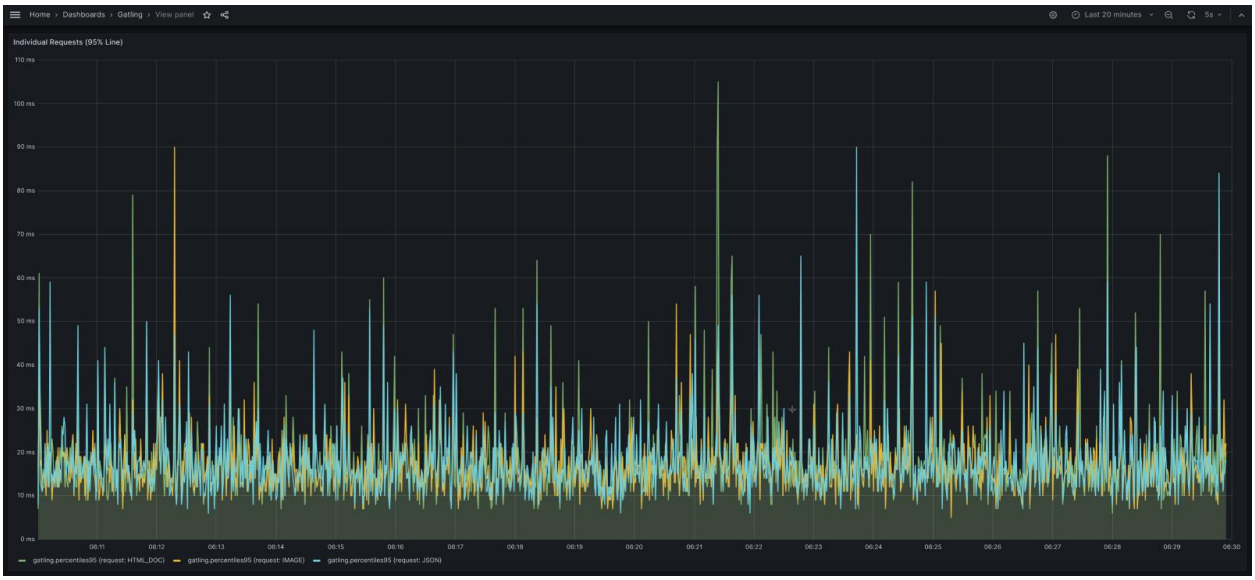


Рис.4.66 Таймінги на запити

Рис. 4.47 Навантаження на цпу

Як ми бачипо по ресурсах сервера на якому запущений застосунок у нас ще є невеликий запас, тож спробуємо підняти ще навантаження і запустити тести на 6 сервері. Після запуску тестів навантаження таймінги виросли в середньому до 25 мс (показано на рис. 4.48). Навантаження на цпу виросло до 86 відсотків (показано на рис. 4.49).

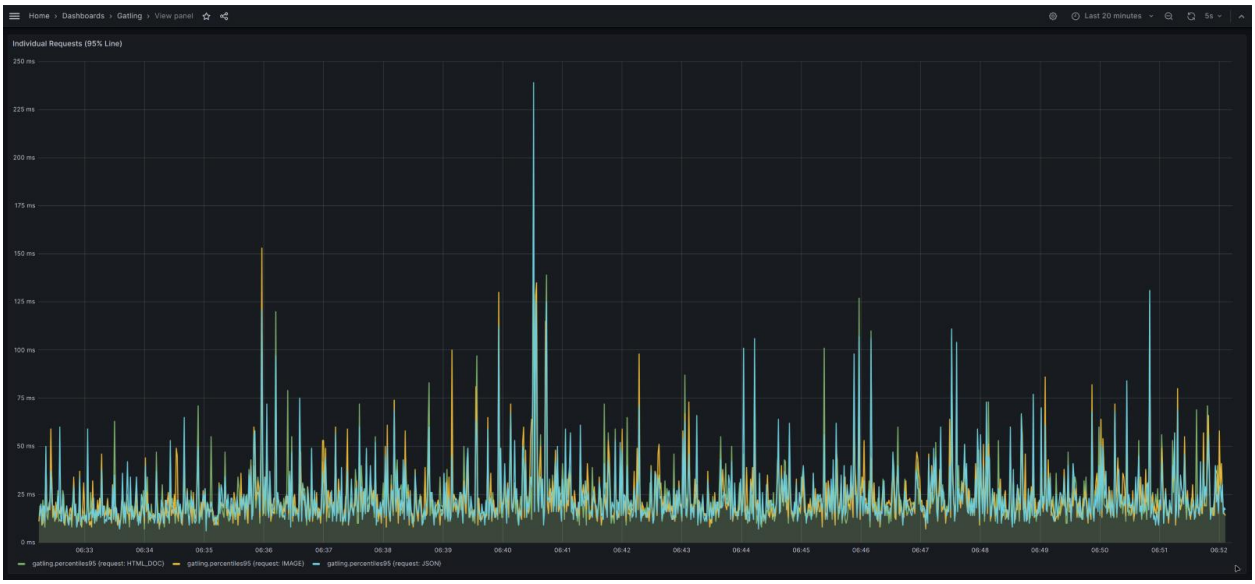


Рис. 4.48 Таймінги на запити

Рис.4.49 Навантаження на цпу

Судячи з результатів ми досягли максимального навантаження яке може витримати сервер з застосунком при даній конфігурації. Для нормальної роботи сервера потрібен запас по цпу від 10 до 20 відсотків. Даний сценарій тестів навантаження майже не споживає оперативної пам’яті і збільшення кількості потоків ніяк не вплинуло на це. Сервер витримав 900 потоків при середніх таймінгах на запити в 25мс.

ВИСНОВКИ ДО РОЗДІЛУ 4

У даному розділі було продемонстроване покрокове виконання тестів навантаження, та проведений аналіз і репортинг результатів. Розглянуто показники основних метрик інструменту моніторингу та їх вплив на коректну працездатність застосунку.

Спираючись на результати тестування, було підібрано оптимальне та максимальне навантаження на веб застосунок.

5 РОЗДІЛ. РОЗРОБКА СТАРТАП-ПРОЕКТУ

В сучасному світі розробка програмного забезпечення не є можливою без проведення етапу тестування навантаження, адже саме цей процес є одним із показників якості продукту. Головна відмінність та ідея розробленого програмного продукту полягає в застосування новітніх технологій та підходів для вирішення задачі тестування навантаження, тому на даному етапі доцільно спланувати основні положення подальшого розвитку та спланувати стратегію дій для створення високоефективної конкурентоспроможної системи. Далі в цьому розділі будуть наведені основні маркетингові, організаційні, фінансово-економічні та комерційні аспекти запропонованої системи.

5.1 Маркетинговий аналіз стартап-проекту

У рамках цього проекту основна ідея проекту: фреймворк тестування навантаження веб додатків для підвищення якості веб систем за заощядження коштів бізнесу у майбутньому. Фреймворк може бути сконфігурований для різних обсягів даних, апаратних можливостей веб додатків та потреб кінцевого користувача.

Таблиця 5.1. Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Фреймворк для тестування веб додатків, можливість конфігурації під конкретний веб сайт	Моделювання стабільних ситуацій навантаження (нормована кількість користувачів-відвідувачів сайту)	Можливість якісно оцінити стабільність стану веб додатку(його апі) і можливо знайти баги функціоналу в програмному коді веб додатку, що потребують фікса

Зміст ідеї	Напрямки застосування	Вигоди для користувача
	Моделювання критичних ситуацій навантаження (ненормована кількість користувачів-відвідувачів сайту)	Виявлення можливих збоїв під великим навантаженням користувачів та виконання ненормованої кількості запитів, дослідження роботи веб додатку у випадку збоїв функціоналу веб сайту та проблем з відведеними ресурсами
	Прогнозування максимального дозволеного трафіку користувачів а кількості запитів які витримає система веб додатку	Вираховування прийнятого навантаження для веб додатку та лімітація кількості користувачів, або ж регуляція ресурсів сайту для можливості підтримки більшого лоаду

Далі проаналізуємо потенційні технічні та економічні переваги ідеї та визначаємо, чим вона відрізняється від існуючих аналогічних або альтернативних продуктів. Зокрема, ми:

- Визначаємо перелік техніко-економічних характеристик та особливостей ідеї;
- Визнаємо попереднє коло проектів-конкурентів, що вже представлені на ринку, та збираємо інформацію про значення техніко-економічних показників власної проектної ідеї та проектів-конкурентів;
- Проводимо порівняльний аналіз показників. Визначаємо показники з а) поганими (W - слабкими), б) схожими (N - нейтральними) та в) добрими (S - сильними) значеннями для своєї ідеї.

Таблиця 5.2. Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№	Техніко-економічні характеристики ідеї	Проекти				W	N	S
		Мій	1	2	3			
1.	Швидкість генерування даних(користувачів, тощо)	+	+	+	-			+
2.	Наявність графічного інтерфейсу	-	-	+	-	-		
3.	Застосування на різних платформах(ОС)	+	+	-	+		+	
4.	Масштабованість	+	+	-	+			+
5.	Зрозумілість степів тестування	-	+	+	-		+	
6.	Ергономічне споживанням ресурсів	-	-	+	-		+	
7.	Точність та ясність репортів і показників результатів тестування	+	+	-	-			+
8.	Забезпечення цілісності даних та надійності системи	+	-	-	+			+
9.	Високий рівень безпеки	+	-	+	+			+

Визначений перелік слабких, сильних та нейтральних характеристик та властивостей ідеї потенційного товару є підґрунтям для формування його конкурентоспроможності. Згідно з таблицею 5.2, серед основних переваг розробленого продукту є підтримка роботи з високонаваантаженими мережами, швидкість генерування даних, масштабованість, Точність та ясність репортів і показників результатів тестування, Забезпечення цілісності даних та надійності системи, високий рівень безпеки на що і варто зробити акцент. Порівняно з існуючими

рішеннями, головними недоліками системи є наявність графічного інтерфейсу та зрозумілість степів тестування, що варто врахувати в майбутньому.

5.2 Технологічний аудит ідеї проекту

Таблиця 5.3. Технологічна здійсненність ідеї проекту

№	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1.	Проект для деплою та запуску тестів	Ansible, Terraform, Consul, Hetzner , Docker	Є в наявності	Доступні безкоштовно, окрім Hetzner
2.	Проект для написання сценаріїв тестування	Gatling, Scala,	Є в наявності	Доступні Безкоштовно
3.				
4.				
5.				
6.	Графічний -інтерфейс	Не є релевантим для фреймворку	Немає в наявності	Не доступноДоступна безкоштовно
7.	Система репортингу	Grafana, node_exporter, prometheus	Є в наявності	Доступні безкоштовно

Згідно з описаними технологіями, необхідними для реалізації проекту створення і розробка фрейміорку для тестування навантаження веб додатків є можливим. Усі необхідні технології, окрім системи прогнозування трафіка є в наявності та доступні безкоштовно. Об’єм генерованих користувачів та звпитів розробленим фреймворком залежатиме від потужності орендованих серверів на Hetzner, що э єдиною технологыю яка потребуэ додатково грошового вкладення

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Наступним здійснемо визначення ринкових можливостей, які відкриваються під час реалізації проекту на ринку, та ринкових загроз, які можуть стати на заваді реалізації проекту. На цьому етапі можна спланувати напрямок розвитку проекту з урахуванням стану ринкового середовища, потреб потенційних клієнтів і

конкуруючих проектних пропозицій. У таблиці 5.4 наведено попередній опис потенційних ринків для стартап-проектів, що розробляються.

Таблиця 5.4. Попередня характеристика потенційного ринку стартап-проекту

№	Показники стану ринку (найменування)	Характеристика
1.	Кількість головних гравців	>3
2.	Загальний обсяг продаж, грн/ум.од.	900 тис.
3.	Динаміка ринку (якісна оцінка)	Зростає 41% (до 2026 р.)
4.	Наявність обмежень для входу (характер обмежень)	Фінансові затрати, наявність кваліфікованих спеціалістів
5.	Специфічні вимоги до стандартизації та сертифікації	Наявність патенту, стандартизація від організацій IETF, ONF та IEEE
6.	Середня норма рентабельності в галузі або по ринку, %	38

Згідно з таблицею, цей ринок є привабливим для виходу на нього, оскільки він є достатньо прибутковим, має низьку кількість кон'юнктурних коливань і прогнозовані темпи зростання ринку до 2026 року. Подальший аналіз ринкових можливостей вимагає опису потенційних груп споживачів. У таблиці 5.5 наведено перелік характеристик та вимог до продуктів стартап-проекту.

Таблиця 5.5. Характеристика потенційних клієнтів стартап-проекту

№	Потреба, що Формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги до споживачів продукту
1.	Стрімке впровадження тестування навантаження для	Айті компанії яким необхідна перевірка навантаження веб додатку	Обсяг навантаження, спесифіка унікаьності функціоналу додатків	До продукції: Надійність; Безпека; Досуп до результатів роботи системи

	підвищення якості	задля уникнення подальших втрат та збереження коштів завдяки уникненню багів		До компанії-постачальника: доступ до службових даних фреймфорку допомога у підтримці продукту надання зворотнього зв’язку
2.	продуктів			
3.	Можливість керування трафіком на продакшинах			
4.	Здатність до роботи з високо-навантаженими мережами			
5.	Гнучкість у впровадженні			
6.	Масштабо-ваність			
7.	Репортинг та аналіз результатів тестування для демонмтрації замовникам	Айті компанії, індивідуальні клієнти	Варіація вимог та даних	

Проведемо аналіз ринкового середовища, а саме: складемо таблиці факторів, що сприяють ринковому впровадженню проекту (таблиця 5.7.), та факторів, що йому перешкоджають (таблиця 5.6.).

Таблиця 5.6. Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1.	Недостатній	Усі потенційні конкуренти та	Проведення маркетингової компанії закордоном

	рівень попиту на внутрішньому ринку	більшість потенційних клієнтів розташовані закордоном	з урахуванням особливостей ринку
2.	Незацікавленість потенційних клієнтів у продукті	Недовіра до технологічної продукту	Проведення демо, надання тестового доступу до продукту клієнтам на перших кроках
3.	Недостатній рівень конкуренто-спроможності	Рішення конкурентів задовольняють базові потреби клієнтів	Вдосконалення цінової політики за покращення якості надання послуг
4.	Неоднозначна економічна ситуація	Відсутність фінансування, недостатня кількість інвестицій	Пошук додаткових шляхів надходження коштів, заощадження витрат
5.	Малий спектр функціоналу	Недостатній функціонал для забезпечення потреб клієнтів	Побудова ітеративних процесів розробки, введення чітких систем планування із врахуванням ринкових потреб
6.	Проблеми з отриманням патентів та стандартизації	Масштабний бюрократичний прошарок для отримання юридичного захисту продукту	Отримання кваліфікованої юридичної допомоги від спеціалістів

Таблиця 5.7. Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1.	Залучення інвестицій	Наявність чіткого бізнес-плану, результатів аналізу ринку та конкуруючих рішень приваблює інвесторів та дозволяє пришвидшити вихід на ринок	Збільшення штату співробітників, пришвидшення темпів розробки та прискорення виходу на самоокупність

2.	Розширення клієнтської бази	Детальне дослідження ринку та проведення маркетингової кампанії потенційним клієнтам	Використання коштів клієнтів на покращення продукту та використання позитивного клієнтського досвіду в подальших маркетингових рішеннях
3.	Розширення функціоналу	Розширення функціоналу продукту для охоплення більшого ринкового сегменту	Ітеративний аналіз ринку для пошуку нових можливостей та приваблення клієнтів
4.	Вихід на міжнародний ринок	Надання доступнішого продукту для представників міжнародного ринку	Забезпечення лідерських позицій у ринковому сегменті, використання позитивного клієнтського досвіду, розширення функціоналу згідно з потребами міжнародного ринку

Далі проведемо аналіз пропозиції: визначемо риси конкуренції на ринку (таблиця 5.8.).

Таблиця 5.8. Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Тип конкуренції: *олігополія*	На ринку присутні декілька компаній, що надають подібні рішення	Пропозиція вигідніших умов впровадження та користування, розробка ключового функціоналу, відсутнього у конкурентів, проведення маркетингових кампаній з акцентами на ключових відмінностях продукту, надання тестового періоду користування

За рівнем конкурентної боротьби: *глобальний*	Існуючі рішення можуть бути впроваджені по всьому світу	Одразу організувати вихід на міжнародну арену та запровадити стратегії відповідно до критеріїв світового ринку
За галузевою ознакою: *внутрішньо-галузева*	Конкуренція в галузі айті, а саме розробок фрейміорків для тестування навантаження	Розробка продукту яка вирішує обмежену кількість задач, але краще, ніж інші рішення
Товарно-родова конкуренція	Продукти-замінники можуть виконувати подібні функції	Створення більш якісної та кращої продукції
Нецінова конкуренція	Конкуренція створюється за допомогою вдосконалення якості продукції	Продукт має постійно вдосконалюватись та покращувати свої характеристики
Не марочна інтенсивність	Споживачів цікавить більше характеристики продукту, ніж під яким брендом випущено продукт	Покращити якісні характеристики продукту порівняно з конкурентними

Проаналізуємо конкурентні умови в галузі більш детально відповідно до моделі п'яти сил М. Портера. З точки зору характеру конкуренції, Портер виділяє п'ять основних факторів, що впливають на привабливість вибору ринку. Вони полягають у наступному:

- Конкуренти, що вже існують у галузі
- Потенційні конкуренти
- Наявність замінників
- Постачальники, що конкурують за ринкову владу;
- Споживачі (а також).

Сильна позиція продукту в кожному з факторів означає, що він може забезпечити необхідний оборот капіталу і здатність впливати на інших учасників ринку, диктуючи свої умови співпраці. Характеристики факторів у цій моделі варіюються від галузі до галузі та змінюються з часом.

Таблиця 5.9. Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти в галузі	Постачальники	Клієнти	Продукти-замінники
	iBeta Quality Assurance, Infosys	Microexcel, A1QA	SPEC INDIA, Algoworks	Айті компанії де розробляються додатки з високим навантаженням	Cybage, JadeGlobal
Висновки	На міжнародних ринках існує конкуренція. Конкуренція невелика, оскільки кожна компанія концентрується на певному сегменті ринку.	Компанії займаються розробкою тестування м навантаження і за потреби можуть випустити схожий продукт	Постачальниками є міжнародними лідерами у сфері тестування навантаження, показник залучення є середній	Більшість клієнтів давно присутні на ринку і мають контакти з постачальниками та можуть бути зацікавлені у продукті	Є декілька подібних рішення для тестування навантаження, які є платними та покривають частину функціоналу

З результатів, наведених у табл. 5.9, можна зробити висновок, що в поточній ситуації ринок є життєздатним, конкуренція на даному етапі незначна, постачальники не зацікавлені пропонувати аналогічні рішення, а існуючі проекти охоплюють лише частину запропонованого функціоналу.

На основі аналізу конкуренції (табл. 5.9) визначено та обґрунтовано перелік конкурентних факторів (табл. 5.10) з урахуванням особливостей ідеї проекту (табл.

5.2), вимог споживачів до продукту (табл. 5.5) та факторів маркетингового середовища (табл. 5.6 — 5.7).

Таблиця 5.10.Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1.	Унікальність функціоналу	Надання вичерпного переліку унікальних функціональних можливостей для ви2рішення основних задач тестування навантаження
2.	Детальний репортинг та глибокий аналіз результатів	Надання графічного репорту результатів проведення тестування навантаження що дасть чітке розуміння стану веб додатку
3.	Надійність та масштабованість	Продукт є надійним та дозволяє масштабуватись в залежності від функціоналу веб додатків

За визначеними факторами конкурентоспроможності проведемо аналіз сильних та слабких сторін стартап-проекту.

Таблиця 5.11.Порівняльний аналіз сильних та слабких сторін стартап-проекту

№	Фактор конкуретноспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з продуктом						
			-3	-2	-1	0	+1	+2	+3
1.	Унікальність функціоналу	17		✓					
2.	Детальний репортинг та глибокий аналіз результатів	20				✓			
3.	Надійність та масштабованість	15			✓				

Фінальним етапом ринкового аналізу можливостей впровадження проекту є складання SWOT-аналізу (матриці аналізу сильних (Strength) та слабких (Weak) сторін, загроз (Troubles) та можливостей (Opportunities) (таблиця 5.12.) на основі виділених ринкових загроз та можливостей, а також сильних та слабких сторін.

Таблиця 5.12. SWOT- аналіз стартап-проекту

Сильні сторони: Унікальність функціоналу Надійність та масштабованість Сучасні технологічні підходи Орієнтація на міжнародний ринок Вирішення чітко сформованої клієнтської потреби	Слабкі сторони: Наявність вже впроваджених конкуруючих рішень Є потреба у часі для виходу на ринок Потреба у постійному надходженні коштів для підтримки та вдосконалення продукту
Можливості: Партнерство зі світовими мережевими лідерами Приваблення інвестицій Вдосконалення продукту на основі клієнтського досвіду Розширення клієнтської бази	Загрози: Швидший або успішніший вихід на ринок нового конкуруючого рішення Відсутність попиту на продукт

На основі SWOT-аналізу розробляється перелік альтернативних ринкових дій (перелік заходів) для виведення стартап-проекту на ринок та орієнтовний оптимальний час його виведення на ринок з урахуванням конкуруючих проектів, які можуть бути виведені на ринок (табл. 5.13).

Таблиця 5.13. Альтернативи ринкового впровадження стартап-проекту

№	Альтернатива ринкової поведінки	Ймовірність отримання ресурсів	Терміни реалізації
1.	Відкриття проекту та розповсюдження в рамках opensource ліцензії	Низька. Є можливість продажу компанії у випадку зацікавленості з боку великих компаній	Одразу після розробки
2.	Розповсюдження проекту за допомогою науково-публічної роботи: конференції, виступи і т.п.	Висока, оскільки публічними заходами можна поширити ідею проекту та завести знайомства з потенційними інвесторами та клієнтами	1 рік
3.	Участь у тендерах і укладення договорів з	Середня, так як на це впливають економічні, політичні, особисті	– 2 роки

	державними установами	мотиви і не можна гарантувати абсолютну прозорість та добросовісну конкуренцію за таких умов	

5.4 Розробка ринкової стратегії стартап-проекту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів (табл. 5.14).

Таблиця 5.14. Вибір цільових груп потенційних споживачів

№	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
1.	Аутсорс айті компанії	Не готові сприйняти продукт	Висока	Висока	Проста
2.	Аутстаф айті компанії	Готові сприйняти продукт	Середній	Висока	Проста
3.	Продуктові айті компанії	Не готові сприйняти продукт	Висока	Висока	Проста
4.	Державні веб продукти	Готові сприйняти продукт	Низька	Середня	Проста
5.	Приватні клієнти	Готові сприйняти продукт	Низька	Низька	Проста
Обрані цільові групи: дата-центри, приватні компанії. Має бути використана стратегія диференційованого маркетингу.					

Згідно з аналізом потенційних комерційних груп споживачів на запропонований продукт, було обрано стратегію диференційованого маркетингу,

оскільки компанія працює водночас із кількома сегментами, з розробкою окремих програм для ринкового впливу.

Таблиця 5.15. Визначення базової стратегії розвитку

№	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможі позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1.	Стратегія зростання	Вибірковий розподіл	Нарощування функціоналу, внутрішнє розширення, виробництво нових продуктів	Стратегія диференційованого маркетингу

Таблиця 5.16.Визначення базової стратегії конкурентної поведінки

№	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
1.	Ні	Комбінований підхід: існує ніша, де не використовуються аналогічні продукти, а наявні переваги дозволяють завоювати існуючих клієнтів.	Ні, компанія може використовувати наявний функціонал і вдосконалювати його згідно з клієнтським досвідом	Стратегія виклику лідера

На основі вимог споживачів з обраних сегментів до постачальника (стартап-компанії) та вимог до продукту (табл. 5.5.), а також в залежності від обраної базової

стратегії розвитку (табл. 5.15) та стратегії конкурентної поведінки (табл. 5.16) розробляється стратегія позиціонування (табл. 5.17). що полягає у формуванні ринкової позиції (комплексу асоціацій), за яким споживачі мають ідентифікувати торгівельну марку/проект.

Таблиця 5.17. Визначення стратегії позиціонування

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкуренто-спроможні позиції стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту
1.	Вирішення проблеми наявності дефектів у веб додатках та виявлення максимального навантаження на систему, надійність, стабільність, підтримка та зворотній зв’язок	Стратегія диференційова-ного маркетингу	Унікальність функціоналу, надійність, масштабованість	Унікальність Надійність Клієнто-орієнтованість

У результаті отримали узгоджену систему рішень щодо ринкової поведінки стартап-компанії, що визначатиме напрями роботи стартап-компанії на ринку.

5.5 Розробка маркетингової програми стартап-проекту

Першим кроком є формування маркетингової концепції продукту, який отримає споживач (таблиця 5.18.).

Таблиця 5.18. Визначення ключових переваг концепції потенційного товару

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1.	Вирішення проблеми тестування навантаження веб додатків	Архітектурно-незалежне рішення з підтримкою адаптивних технологій	Використання сучасних технологій та підходів, робота з високонавантаженими додатками, надійність, безпека.
2.	Зручний репортинг та аналіз результатів	Детальний аналіз допомагає чітко визначити максимальне навантаження на додаток що дозволить уникнути дефектів та втрати коштів бізнесу	Інтуітивно зрозумілий репортинг
3.	Підтримка та зворотній зв’язок	Простота у використанні та налаштуванні, менші витрати часу на підтримку	Допомога у налаштуванні та підтримці фреймворку

Далі розробляється трирівнева маркетингова модель товару: уточнюється ідея продукту та/або послуги, його фізичні складові, особливості процесу його надання.

Таблиця 5.19. Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові	
I. Товар за задумом	Фреймворк для тестування навантаження веб додатку	
	Властивості/характеристики	Розмір

II. Товар у реальному виконанні	Хмарне сховище для віртуалізації серверів	6 MB
	Локальна обчислювальна машина	16 GB
	Якість: внутрішнє тестування, логування виконання, система підтримки	
	Пакування: SaaS-концепти, створення API та дашбордів для клієнтів.	
	Марка: назва організації-розробника + назва товару	
III. Товар із підкріпленням	До продажу: ліцензія на використання	
	Після продажу: надання доступу до API	
За рахунок чого потенційний товар буде захищено від копіювання: Комплексне поєднання властивостей і характеристик, закладене на другому та третьому рівнях товару.		

Таблиця 5.20. Визначення меж встановлення ціни

№	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі становлення ціни на товар/послугу
1.	2000$ - 10000$/міс.	2000$ - 1000$/міс.	>50000$/міс.	4000-6000/міс.

Таблиця 5.21. Формування системи збуту

№	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1.	Покупка ліцензії на продукт або договір про надання послуг без передачі ПО до замовника	Надання доступу до API, розгортання системи на архітектурі клієнта	Канал нульового рівня	Інтернет, тендерні торги

Таблиця 5.22. Концепція маркетингових комунікацій

№	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1.	Купують товар на вимогу	Наукові кола, Інтернет, соціальні мережі, форуми	Інтернет-маркетинг	Презентація товару, унікальності і якості.	Пояснення функцій та можливо-стей продукту.

ВИСНОВКИ ДО РОЗДІЛУ 5

П`ятий розділ присвячено розробці стартап-проекту на основі розробленого фреймворку, що дозволяє проводити тестування навантаження веб додатків.

Для дослідження в рамках розробки стартап-проекту було виконано наступні завдання:

Наведено загальний опис ідеї стартап-проекту, описано функціонал, визначено основних конкурентів та порівняно функціонал існуючих продуктів для визначення переваг розроблюваного продукту над конкурентами.

Проаналізовано сучасні ринкові можливості для запуску проекту та успішного виходу на ринок. Виявлено можливості та загрози стартап проекту та сплановані дії щодо їх запобігання.

Порівняно перспективи запуску в порівнянні з конкурентами, встановлено план та ринкову стратегію розвитку продукту, визначено цільові групи та способи просування проекту.

Розроблено маркетингову програму для просування продукту на ринку, описано способи та основні канали збуту, визначено пріоритетні цільові групи та маркетингові повідомлення для розширення клієнтської бази.

В підсумку, отримано стартап-проект для запуску розробленого програмного продукту на базі дипломного проекту для виходу на ринок.

ВИСНОВКИ

Магістерська робота присвячена розробці фреймворку тестування навантаження веб додатків.

У роботі було виконано дослідження терміну тестування навантаження, обрані інструменти для реалізації фреймворку та визначено основні аспекти та вектори розвитку. Розглянуто головні види тестування продуктивності, виконано огляд існуючих рішень. Виявлено проблему, що особливо актуальна для веб додатків: нездатність підтримувати велику кількість користувачів та запитів на систему, наявність дефектів у функціоналі продукту, що призводить до зниження якості веб сайту на в результаті втрати коштів для бізнесу. Запропоновано спосіб вирішення даної проблеми із застосуванням сучасних підходів и, а також розглянуто альтернативні підходи.

Сформовано основні завдання, які має вирішувати готовий фреймворк. Окрім цього, описано основні етапи побудови системи. Виокремлено та сформовано процес написання тестів, створення системи моніторингу та проекту для розгортання інфраструктури.

За проведеною розробкою реалізовано фінальну версію фреймворку. Виконано технологічний огляд засобів реалізації, обґрунтовано вибір мови програмування, архітектури та сервісів. Виконано опис структури програмного додатку та реалізації його компонентів. Надано завершальні результати процесу тестування, їх обґрунтування та пояснення.

Завдяки технологічним та концептуальним особливостям даний фреймворк може бути застосований до веб додатків з очікуваним великими навантаженням, так і із збалансованим. Також запропоноване рішення є масштабованим, тому з практичної точки зору, спосіб є достатньо універсальним та стабільним для застосування у виробничому середовищі. Окрім цього, методи роботи з великими даними дозволяють отримувати результати тестування в режимі реального часу, що з

практичної точки зору дозволяє вирішувати задачу економії часу у випадку виникнення дефектів, або зупинки роботи системи відповідно вимогам.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1) Ian Molyneaux, “The Art of Application Performance Testing: From Strategy to Tools” , 2019.

2) Surya Kosuri, “Excellence in Performance Testing and Engineering Handy Book” , 2016.

3) Performance Testing Tutorial – Types (Example) [Електронний ресурс] - Режим доступу до ресурсу: https://www.guru99.com/performance-testing.html

4) I. Microsoft Corporation, “Performance Testing Guidance for Web Applications”, 2007.

5) Gatling. Test. Succeed. Iterate. [Електронний ресурс] – Режим доступу до ресурсу: https://gatling.io/

6) Terraform. Docs [Електронний ресурс] – Режим доступу до ресурсу: https://developer.hashicorp.com/terraform?product_intent=terraform

7) Product Documentation for Red Hat Ansible Automation Platform 2.4[Електронний ресурс] – Режим доступу до ресурсу: https://access.redhat.com/documentation/enus/red_hat_ansible_automation_platform/2.4

8) Docker docs [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/

9) Luke Kysow , " Consul: Up and Running ", 2022.

10) Emma Hogbin Westby, “Git for Teams: A User-Centered Approach to Creating Efficient Workflows in Git”, 2015.

11) Eric Salituro , “Learn Grafana 7.0: A beginner's guide to getting well versed in analytics, interactive dashboards, and monitoring”, 2020

12) Julien Pivotto, Brian Brazil , “Prometheus: Up & Running, 2nd Edition”, 2023

13) Grafana docs [Електронний ресурс] – Режим доступу до ресурсу: https://grafana.com/docs/

14) an Miell and Aidan Hobson Sayers, “Docker in Practice, Second Edition” Computer Networks,vol. 75, 2019.

15) Key Elements of a Performance Management Framework [Електронний ресурс] – Режим доступу до ресурсу: https://www.performyard.com/articles/performance-management-framework

16) Андрій Кушлакб “Як провести тестування навантаження: що потрібно знати та якими інструментами користуватися”[Електронний ресурс] – Режим доступу до ресурсу: https://highload.today/uk/blogs/yak-provesti-testuvannya-navantazhennya-shho-potribno-znati-ta-yakimi-instrumentami-koristuvatisya/

17) Dac-Nhuong Le, Raghvendra Kumar, Gia Nhu Nguyen, Jyotir Moy Chatterjee , “Cloud Computing and Virtualization”, 2018.

18) ІНСТРУМЕНТИ ДЛЯ НАВАНТАЖУВАЛЬНОГО ТЕСТУВАННЯ [Електронний ресурс] , 2019. - Режим доступу до ресурсу: https://training.qatestlab.com/blog/technical-articles/load-testing-tools/

19) Hetzner Docs [Електронний ресурс] - Режим доступу до ресурсу: https://docs.hetzner.com/

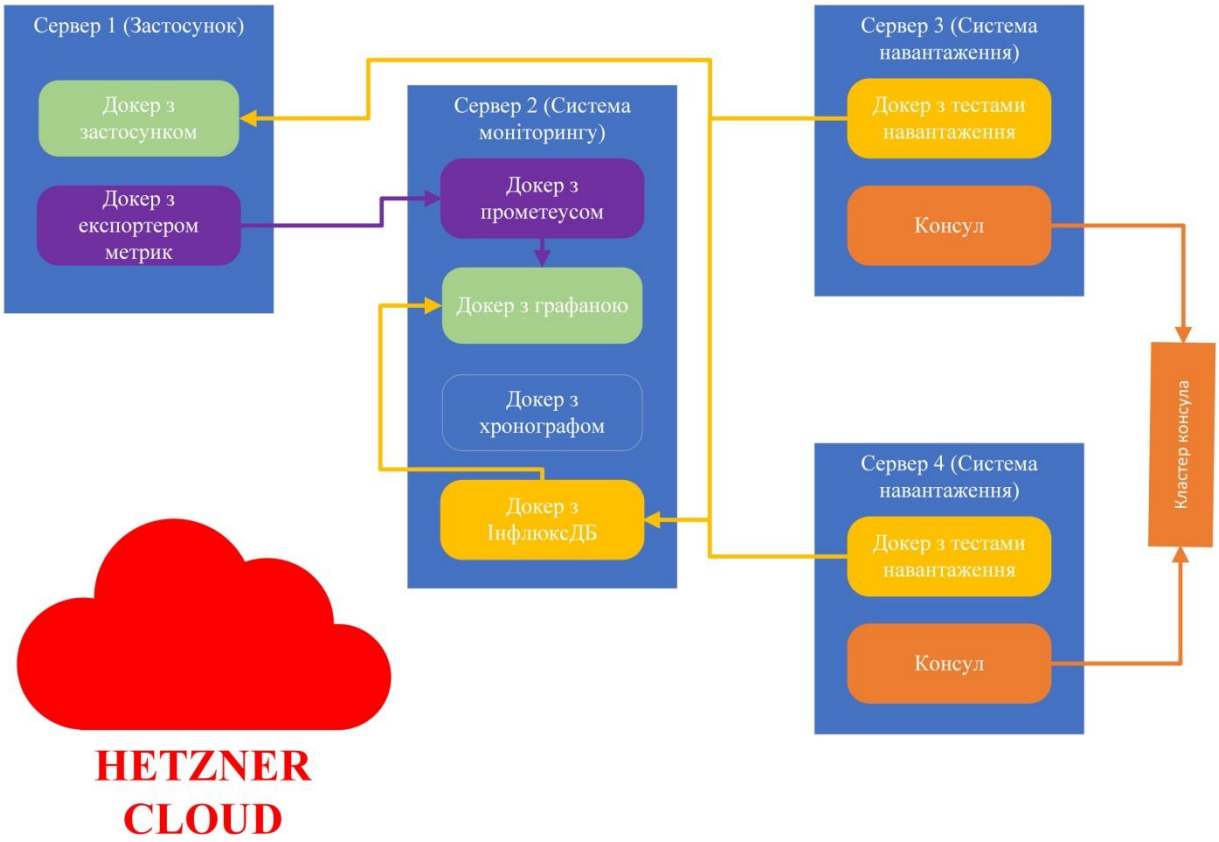
20) Michael Heap, “Ansible: From Beginner to Pro”, 2015.

21) Steve Souders, “High Performance Web Sites: Essential Knowledge for Front-End Engineers”, 2007.

22) Artur Ejsmont, “Web Scalability for Startup Engineers”, 2015

ДОДАТОК 1.

ЗАГАЛЬНА СТРУКТУРА ФРЕЙМВОРКУ НАВАНТАЖЕННЯ



ДОДАТОК 2.

ДІАГРАМА КЛАСІВ ПРОЕКТУ З ТЕСТАМИ НАВАНТАЖЕННЯ

ДОДАТОК 3.

ЛІСТИНГ КОДУ ПРОЕКТУ З ТЕСТАМИ НАВАНТАЖЕННЯ

```
package simulation

import core.constans.ScenariosName
import core.precondition.PreconditionScript
import io.gatling.core.Predef._
import io.gatling.core.config.GatlingConfiguration
import io.gatling.core.controller.inject.open.OpenInjectionStep
import io.gatling.http.protocol.HttpProtocolBuilder
import org.slf4j.{Logger, LoggerFactory}
import scenarios.{HttpScenario1, HttpScenario2}

import scala.concurrent.duration.DurationInt

class BasicSimulation extends Simulation {
  GatlingConfiguration.loadForTest()
  GatlingConfiguration.loadActorSystemConfiguration()

  val logger: Logger = LoggerFactory.getLogger(this.getClass)
  val preconditionScript: PreconditionScript = new PreconditionScript
  val basicSimulationUtils: BasicSimulationUtils = new
BasicSimulationUtils(preconditionScript)
  val httpProtocol: HttpProtocolBuilder = preconditionScript.httpProtocol
  val defaultHeavySideUsers: OpenInjectionStep =
basicSimulationUtils.defaultHeavySideUsers()

  preconditionScript.propUtils.getConsulProperties.getOther.getScenario match {
    case ScenariosName.HTTP1 =>
      setUp(
        new
HttpScenario1(preconditionScript).scn.inject(defaultHeavySideUsers)).protocols(httpProtocol)

    case ScenariosName.HTTP2 =>
      setUp(
        new
HttpScenario2(preconditionScript).scn.inject(defaultHeavySideUsers)).protocols(httpProtocol)

    case _ =>
      throw new Exception("WRONG SCENARIO NAME")
  }
}


package simulation

import core.precondition.PreconditionScript
import io.gatling.core.Predef.{DurationInteger, heavisideUsers}
import io.gatling.core.controller.inject.open.OpenInjectionStep
import org.slf4j.{Logger, LoggerFactory}

class BasicSimulationUtils(preconditionScript: PreconditionScript) {
  val logger: Logger = LoggerFactory.getLogger(this.getClass)

  def defaultHeavySideUsers(): OpenInjectionStep = {
    heavisideUsers(
```

```
      preconditionScript.propUtils.getConsulProperties.getOther.getUsersPerContainer)
      .during(preconditionScript.propUtils.getConsulProperties.getOther.getDuring.seconds)
  }

}


package simulation

import io.gatling.core.Predef._
import io.gatling.core.config.GatlingConfiguration
import io.gatling.core.structure.ScenarioBuilder
import io.gatling.http.Predef.http
import io.gatling.http.protocol.HttpProtocolBuilder
import org.slf4j.{Logger, LoggerFactory}
import scenarios._

import scala.concurrent.duration.DurationInt

class TestSimulation extends Simulation {
  GatlingConfiguration.loadForTest()
  GatlingConfiguration.loadActorSystemConfiguration()

  val logger: Logger = LoggerFactory.getLogger(classOf[TestSimulation])

  val httpProtocol: HttpProtocolBuilder = http
    .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0) Gecko/20100101
Firefox/16.0")
    .disableFollowRedirect

  val test: ScenarioBuilder = scenario("#TestSimulation#")
    .exec(
      exec {
        session =>
          logger.debug("Test")
          session
      }
    )

  setUp(
    test.inject(heavisideUsers(1).during(10.seconds)),
  ).protocols(httpProtocol)
}


package core.precondition

import core.servicesClients.http.HttpTestClient
import core.utils.PropertiesUtils
import io.gatling.core.Predef._
import io.gatling.http.Predef._
import io.gatling.http.protocol.HttpProtocolBuilder
import org.slf4j.{Logger, LoggerFactory}

class PreconditionScript {
  val logger: Logger = LoggerFactory.getLogger(this.getClass)

  val propUtils = new PropertiesUtils(1)
  propUtils.updatePropertiesWithSystemVariables()
  propUtils.prepareConsul()
```

```
  private val GatlingCustomConfigFile = "gatling.conf"
  private val GatlingCustomConfigFileOverrideSystemProperty = "gatling.conf.file"

  val customConfigFile: String =
sys.props.getOrElse(GatlingCustomConfigFileOverrideSystemProperty, GatlingCustomConfigFile)


  var httpClient: HttpTestClient = new HttpTestClient(propUtils)

  val httpProtocol: HttpProtocolBuilder = http
    .acceptEncodingHeader("gzip, deflate")
    .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0) Gecko/20100101
Firefox/16.0")


}


package scenarios

import core.precondition.PreconditionScript
import core.servicesClients.http.HttpTestClient
import io.gatling.core.Predef._
import io.gatling.core.structure.{ChainBuilder, ScenarioBuilder}
import org.slf4j.{Logger, LoggerFactory}

import java.util.concurrent.TimeUnit
import scala.concurrent.duration.Duration


class HttpScenario1(preconditionScript: PreconditionScript) {
  val logger: Logger = LoggerFactory.getLogger(this.getClass)

  val httpClient: HttpTestClient = preconditionScript.httpClient

  val scn: ScenarioBuilder = scenario("#HTTP SCENARIO 1#").
    exec(
      exec {
        session =>
          val ipApp = preconditionScript.propUtils.getConsulProperties.getHosts.getApp
          logger.info("ipApp" , ipApp)
          session
            .set("doGetTime", 0L).set("getTimeCalculate", false)
            .set("doDeflateTime", 0L).set("deflateTimeCalculate", false)
            .set("doDenyTime", 0L).set("denyTimeCalculate", false)
            .set("doEncodingUtf8Time", 0L).set("encodingUtf8TimeCalculate", false)
      }.
      exec(
        forever() {
          exec(
            httpClient.get,
            httpClient.deflate,
            httpClient.deny,
            httpClient.encodingUtf8,
            pause(Duration.create(1000, TimeUnit.MILLISECONDS))
          )
        }
      )
    )
```

```
}


package scenarios

import core.precondition.PreconditionScript
import core.servicesClients.http.HttpTestClient
import io.gatling.core.Predef._
import io.gatling.core.structure.{ChainBuilder, ScenarioBuilder}
import org.slf4j.{Logger, LoggerFactory}

import java.util.concurrent.TimeUnit
import scala.concurrent.duration.Duration


class HttpScenario1(preconditionScript: PreconditionScript) {
  val logger: Logger = LoggerFactory.getLogger(this.getClass)

  val httpClient: HttpTestClient = preconditionScript.httpClient

  val scn: ScenarioBuilder = scenario("#HTTP SCENARIO 1#").
    exec(
      exec {
        session =>
          val ipApp = preconditionScript.propUtils.getConsulProperties.getHosts.getApp
          logger.info("ipApp" , ipApp)
          session
            .set("doGetTime", 0L).set("getTimeCalculate", false)
            .set("doDeflateTime", 0L).set("deflateTimeCalculate", false)
            .set("doDenyTime", 0L).set("denyTimeCalculate", false)
            .set("doEncodingUtf8Time", 0L).set("encodingUtf8TimeCalculate", false)
      }.
      exec(
        forever() {
          exec(
            httpClient.get,
            httpClient.deflate,
            httpClient.deny,
            httpClient.encodingUtf8,
            pause(Duration.create(1000, TimeUnit.MILLISECONDS))
          )
        }
      )
    )
}


package scenarios

import core.precondition.PreconditionScript
import core.servicesClients.http.HttpTestClient
import io.gatling.core.Predef._
import io.gatling.core.structure.{ChainBuilder, ScenarioBuilder}
import org.slf4j.{Logger, LoggerFactory}

import java.util.concurrent.TimeUnit
import scala.concurrent.duration.Duration


class HttpScenario2(preconditionScript: PreconditionScript) {
```

```
  val logger: Logger = LoggerFactory.getLogger(this.getClass)

  val httpClient: HttpTestClient = preconditionScript.httpClient

  val scn: ScenarioBuilder = scenario("#HTTP SCENARIO 1#").
    exec(
      exec {
        session =>
          val ipApp = preconditionScript.propUtils.getConsulProperties.getHosts.getApp
          logger.info("ipApp" , ipApp)
          session
            .set("doHtmlDocTime", 0L).set("htmlDocTimeCalculate", false)
            .set("doJsonTime", 0L).set("jsonTimeCalculate", false)
            .set("doImageTime", 0L).set("imageTimeCalculate", false)
      }.
        exec(
          forever() {
            exec(
              httpClient.htmlDoc,
              httpClient.json,
              httpClient.image,
              pause(Duration.create(1000, TimeUnit.MILLISECONDS))
            )
          }
        )
    )
}
```

ДОДАТОК 4.

ЛІСТИНГ КОДУ ПРОЕКТУ ДЛЯ ПІДНЯТТЯ ІНФРАСТРУКТУРИ ТА ДЕПЛОЙМЕНТУ ТЕСТІВ НАВАНТАЖЕННЯ

```
################ Variables ################

variable "hcloud_token" {
  default = "0QtyGLqWyuWkrPn49Pb5aO7aM80bqqJLusgEAErFpvWDMXcRHQgJEkfgWn3Un77r"
}

# Define provider
terraform {
  required_providers {
    hcloud = {
      source = "hetznercloud/hcloud"
    }
  }
}



# Define Hetzner provider token
provider "hcloud" {
  token = var.hcloud_token
}

# Obtain ssh key data
data "hcloud_ssh_key" "ssh_key" {
  fingerprint = "bb:12:e1:98:0e:67:57:88:52:74:0e:3c:d9:a3:3f:8f"
}

# Create Debian 11 server
resource "hcloud_server" "web" {
  count       = var.instances
  name        = "bot-server-${count.index}"
  image       = var.os_type
  server_type = var.server_type
  location    = var.location
  ssh_keys  = ["${data.hcloud_ssh_key.ssh_key.id}"]
  labels = {
    type = "bot"
  }
  user_data = file("user_data.yml")
}


resource "hcloud_server_network" "web_network" {
  count     = var.instances
  server_id = hcloud_server.web[count.index].id
  subnet_id = hcloud_network_subnet.hc_private_subnet.id
}

resource "hcloud_network_subnet" "hc_private_subnet" {
  network_id   = hcloud_network.hc_private.id
  type         = "cloud"
  network_zone = "eu-central"
```

```
  ip_range     = var.ip_range
}

resource "hcloud_network" "hc_private" {
  name     = "hc_private"
  ip_range = var.ip_range
}

output "web_servers_status" {
  value = {
    for server in hcloud_server.web :
    server.name => server.status
  }
}

output "web_servers_ips" {
  value = {
    for server in hcloud_server.web :
    server.name => server.ipv4_address
  }
}
```

```
---
- name: Print mosh version
  debug:
    msg:
      - 'Ansible Version: {{ ansible_play_hosts }}'

- name: Start bot container
  docker_container:
    name: "gatling-bot{{ emul_prefix | default('') }}"
    image: "aurora0/bot:latest"
    env:
      GrafanaPostfix: "gatling"
      propertiesFile: "diplom"
      scenario: "http2"
      during: "300"
      usersPerContainer: "150"
      app: "{{hostvars['app-server-0'].ansible_host}}:8000"
      consul: "172.17.0.1"
    volumes:
    #      - /tmp/gatling/gatling.conf:/opt/gatling/conf/gatling.conf
    #      - /var/log/gatling:/opt/gatling/results
    state: started
    pull: true
    command: mvn -X gatling:test -Dgatling.simulationClass=simulation.BasicSimulation -
Dgatling.conf.file=gatling-configs/gatling.conf
    log_driver: json-file
    log_options:
      max-size: "500m"
      max-file: "1"
```

```
---
- name: Print mosh version
  debug:
    msg:
      - 'Ansible Version: {{ ansible_play_hosts }}'
```

```
- name: Start bot stop
  docker_container:
    name: "gatling-bot{{ emul_prefix | default('') }}"
    state: absent


---
- name: Install required system packages
  apt:
    pkg:
      - apt-transport-https
      - ca-certificates
      - curl
      - software-properties-common
      - python3-pip
      - virtualenv
      - python3-setuptools
    state: latest
    update_cache: true

- name: Add Docker GPG apt Key
  apt_key:
    url: https://download.docker.com/linux/ubuntu/gpg
    state: present

- name: Add Docker Repository
  apt_repository:
    repo: deb https://download.docker.com/linux/ubuntu focal stable
    state: present

- name: Update apt and install docker-ce
  apt:
    name: docker-ce
    state: latest
    update_cache: true

- name: Install Docker Module for Python
  pip:
    name: docker

---
- hosts:
    - host_bot
  user: root
  become: true
  become_method: sudo
#  gather_facts: false
  roles:
    - bot_start

---
- hosts:
    - host_bot
  user: root
  become: true
  become_method: sudo
#  gather_facts: false
  roles:
    - bot_stop
```

```
---
- hosts:
    - host_bot
  user: root
  become: true
  become_method: sudo
#  gather_facts: false
  roles:
    - docker_install
    - consul
```

ДОДАТОК 5.

ЛІСТИНГ КОДУ ПРОЕКТУ ДЛЯ РОЗГОРТАННЯ СИСТЕМИ МОНІТОРИНГУ

```
---
- hosts:
    - app_type_http
  user: root
  become: true
  become_method: sudo
  #  gather_facts: false
  roles:
    - deploy_exporter

- hosts:
    - app_type_grafana
  user: root
  become: true
  become_method: sudo
#  gather_facts: false
  roles:
    - docker_install
    - deploy_influxdb
    - deploy_grafana

- hosts:
    - localhost
  roles:
    - configure_grafana_dashboards


################ Variables ################

variable "hcloud_token" {
  default = "0QtyGLqWyuWkrPn49Pb5aO7aM80bqqJLusgEAErFpvWDMXcRHQgJEkfgWn3Un77r"
}

# Define provider
terraform {
  required_providers {
    hcloud = {
      source = "hetznercloud/hcloud"
    }
  }
}

# Define Hetzner provider token
provider "hcloud" {
  token = var.hcloud_token
}

# Obtain ssh key data
data "hcloud_ssh_key" "ssh_key" {
  fingerprint = "bb:12:e1:98:0e:67:57:88:52:74:0e:3c:d9:a3:3f:8f"
}
```

```
resource "hcloud_server" "grafana" {
  count       = var.instances_grafana
  name        = "grafana-server-${count.index}"
  image       = var.os_type
  server_type = var.server_type
  location    = var.location
  ssh_keys  = ["${data.hcloud_ssh_key.ssh_key.id}"]
  labels = {
    type = "grafana"
  }
  user_data = file("user_data.yml")
}

resource "hcloud_server_network" "grafana_network" {
  count     = var.instances_grafana
  server_id = hcloud_server.grafana[count.index].id
  subnet_id = hcloud_network_subnet.hc_private_subnet_grafana.id
}

resource "hcloud_network_subnet" "hc_private_subnet_grafana" {
  network_id   = hcloud_network.hc_private_grafana.id
  type         = "cloud"
  network_zone = "eu-central"
  ip_range     = var.ip_range_grafana
}

resource "hcloud_network" "hc_private_grafana" {
  name     = "hc_private_grafana"
  ip_range = var.ip_range_grafana
}



output "grafana_servers_status" {
  value = {
    for server in hcloud_server.grafana :
    server.name => server.status
  }
}

output "grafana_servers_ips" {
  value = {
    for server in hcloud_server.grafana :
    server.name => server.ipv4_address
  }
}


---
- name: Create influxdb datasource
  community.grafana.grafana_datasource:
    name: "InfluxDB"
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    org_id: "1"
    ds_type: "influxdb"
    ds_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:8086"
    database: "gatlingdb"
```

```
  ignore_errors: true

- name: create prometheus datasource
  community.grafana.grafana_datasource:
    name: Prometheus
    ds_type: prometheus
    url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    ds_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:9090"
    access: proxy
    tls_skip_verify: true
  ignore_errors: true

- name: Import Grafana dashboard node_exporter
  community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/node_exporter.json"
  ignore_errors: true


- name: Import Grafana dashboard
  community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/gatling-dashboard.json"
  ignore_errors: true

- name: Import Grafana dashboard trend2
  community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/gatling-report-trend2.json"
  ignore_errors: true

- name: Import Grafana dashboard report_rev3
  community.grafana.grafana_dashboard:
    grafana_url: "http://{{hostvars['grafana-server-0'].ansible_host}}:3000"
    grafana_user: "admin"
    grafana_password: "admin"
    state: present
    commit_message: Updated by ansible
    overwrite: yes
    path: "{{playbook_dir}}/dashboards/grafana-report_rev3.json"
  ignore_errors: true

 ---
- name: Restart a container
  docker_container:
```

```
    name: node-exporter
    image: prom/node-exporter
    state: started
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command: [
      "--path.procfs=/host/proc",
      "--path.sysfs=/host/sys",
      "--collector.filesystem.ignored-mount-
points=^/(sys|proc|dev|host|etc|rootfs/var/lib/docker/containers|rootfs/var/lib/docker/overl
ay2|rootfs/run/docker/netns|rootfs/var/lib/docker/aufs)($$|/)"
    ]
    ports:
      - "9100:9100"
    restart: yes


---
- debug: var=hostvars['app-server-0'].ansible_host
  tags:
    - grafana
- debug: var=hostvars['grafana-server-0'].ansible_host
  tags:
    - grafana
- name: Create consul conf dir
  file:
    path: /tmp/conf
    state: directory
  tags:
    - grafana
- name: Create consul data dir
  file:
    path: /tmp/data
    state: directory
  tags:
    - grafana

- name: Add consul config
  template:
    src: "prometheus.yml.j2"
    dest: "/tmp/conf/prometheus.yml"
  tags:
    - grafana

- name: Change permission on server.hcl.j2 file
  file:
    path: /tmp/conf/prometheus.yml
    state: file
    owner: root
    group: root
    mode: 0777

- name: Change permission on data
  file:
    path: /tmp/data
    state: directory
    owner: root
    group: root
```

```
    mode: 0777

- name: Change permission on conf
  file:
    path: /tmp/conf
    state: directory
    owner: root
    group: root
    mode: 0777

- name: Start a prometheus
  docker_container:
    name: prometheus
    image: prom/prometheus:latest
    state: started
    volumes:
      - /tmp/conf:/etc/prometheus
      - /tmp/data:/prometheus
    command: [
      "--config.file=/etc/prometheus/prometheus.yml",
      "--storage.tsdb.path=/prometheus",
      "--storage.tsdb.retention=200h"
    ]
    ports:
      - "9090:9090"
    restart: yes
  tags:
    - grafana

- name: Restart a grafana
  docker_container:
    name: grafana
    image: grafana/grafana:latest-ubuntu
    state: started
    ports:
      - "3000:3000"
    restart: yes
  tags:
    - grafana


---
- debug: var=hostvars['app-server-0'].ansible_host
  tags:
    - grafana
- debug: var=hostvars['grafana-server-0'].ansible_host
  tags:
    - influxdb
- name: Create conf dir
  file:
    path: /tmp/conf
    state: directory
  tags:
    - influxdb
- name: Create data_influxdb dir
  file:
    path: /tmp/data_influxdb
    state: directory
  tags:
    - influxdb
```

```
- name: Create data_chronograf dir
  file:
    path: /tmp/data_chronograf
    state: directory
  tags:
    - influxdb

- name: Add consul config
  template:
    src: "influxdb.conf.j2"
    dest: "/tmp/conf/influxdb.conf"
  tags:
    - influxdb

- name: Change permission on influxdb.conf file
  file:
    path: /tmp/conf/influxdb.conf
    state: file
    owner: root
    group: root
    mode: 0777

- name: Change permission on data_influxdb
  file:
    path: /tmp/data_influxdb
    state: directory
    owner: root
    group: root
    mode: 0777

- name: Change permission on data_chronograf
  file:
    path: /tmp/data_chronograf
    state: directory
    owner: root
    group: root
    mode: 0777

- name: Change permission on conf
  file:
    path: /tmp/conf
    state: directory
    owner: root
    group: root
    mode: 0777

- name: Start a influxdb
  docker_container:
    name: influxdb
    image: influxdb:1.8
    state: started
    volumes:
      - /tmp/data_influxdb:/var/lib/influxdb
      - /tmp/conf/influxdb.conf:/etc/influxdb/influxdb.conf
    env:
      INFLUXDB_GRAPHITE_ENABLED: "true"
      INFLUXDB_USERNAME: "admin"
      INFLUXDB_PASSWORD: "admin"
      INFLUXDB_DB: "gatlingdb"
    ports:
```

```
      - "8086:8086"
      - "2003:2003"
    restart: yes
  tags:
    - influxdb

- name: Start a chronograf
  docker_container:
    name: chronograf
    image: chronograf:latest
    state: started
    volumes:
      - /tmp/data_chronograf:/var/lib/chronograf
      - /tmp/conf/influxdb.conf:/etc/influxdb/influxdb.conf
    env:
      INFLUXDB_URL: "http://{{hostvars['grafana-server-0'].ansible_host}}:8086"
      INFLUXDB_USERNAME: "admin"
      INFLUXDB_PASSWORD: "admin"
    ports:
      - "{{hostvars['grafana-server-0'].ansible_host}}:8888:8888"
    restart: yes
  tags:
    - influxdb

---
- name: Install aptitude
  apt:
    name: aptitude
    state: latest
    update_cache: true

- name: Install required system packages
  apt:
    pkg:
      - apt-transport-https
      - ca-certificates
      - curl
      - software-properties-common
      - python3-pip
      - virtualenv
      - python3-setuptools
    state: latest
    update_cache: true

- name: Add Docker GPG apt Key
  apt_key:
    url: https://download.docker.com/linux/ubuntu/gpg
    state: present

- name: Add Docker Repository
  apt_repository:
    repo: deb https://download.docker.com/linux/ubuntu focal stable
    state: present

- name: Update apt and install docker-ce
  apt:
    name: docker-ce
    state: latest
    update_cache: true
```

```
- name: Install Docker Module for Python
  pip:
    name: docker
```

ДОДАТОК 6.

ЛІСТИНГ КОДУ ПРОЕКТУ ДЛЯ РОЗГОРТАННЯ ВЕБ ДОДАТКУ

```
############### Variables ###############

variable "hcloud_token" {
  default = "0QtyGLqWyuWkrPn49Pb5aO7aM80bqqJLusgEAErFpvWDMXcRHQgJEkfgWn3Un77r"
}

# Define provider
terraform {
  required_providers {
    hcloud = {
      source = "hetznercloud/hcloud"
    }
  }
}

# Define Hetzner provider token
provider "hcloud" {
  token = var.hcloud_token
}

# Obtain ssh key data
data "hcloud_ssh_key" "ssh_key" {
  fingerprint = "bb:12:e1:98:0e:67:57:88:52:74:0e:3c:d9:a3:3f:8f"
}

# Create Debian 11 server
resource "hcloud_server" "app" {
  count       = var.instances_app
  name        = "app-server-${count.index}"
  image       = var.os_type
  server_type = var.server_type
  location    = var.location
  ssh_keys  = ["${data.hcloud_ssh_key.ssh_key.id}"]
  labels = {
    type = "http"
  }
  user_data = file("user_data.yml")
}


resource "hcloud_server_network" "app_network" {
  count     = var.instances_app
  server_id = hcloud_server.app[count.index].id
  subnet_id = hcloud_network_subnet.hc_private_subnet_app.id
}

resource "hcloud_network_subnet" "hc_private_subnet_app" {
  network_id   = hcloud_network.hc_private_app.id
  type         = "cloud"
  network_zone = "eu-central"
  ip_range     = var.ip_range_app
}

resource "hcloud_network" "hc_private_app" {
```

```
  name     = "hc_private_app"
  ip_range = var.ip_range_app
}

output "app_servers_status" {
  value = {
    for server in hcloud_server.app :
    server.name => server.status
  }
}

output "app_servers_ips" {
  value = {
    for server in hcloud_server.app :
    server.name => server.ipv4_address
  }
}


---
- name: Install aptitude
  apt:
    name: aptitude
    state: latest
    update_cache: true

- name: Install required system packages
  apt:
    pkg:
      - apt-transport-https
      - ca-certificates
      - curl
      - software-properties-common
      - python3-pip
      - virtualenv
      - python3-setuptools
    state: latest
    update_cache: true

- name: Add Docker GPG apt Key
  apt_key:
    url: https://download.docker.com/linux/ubuntu/gpg
    state: present

- name: Add Docker Repository
  apt_repository:
    repo: deb https://download.docker.com/linux/ubuntu focal stable
    state: present

- name: Update apt and install docker-ce
  apt:
    name: docker-ce
    state: latest
    update_cache: true

- name: Install Docker Module for Python
  pip:
    name: docker
```

```
---
- name: Restart a container
  docker_container:
    name: httpbin
    image: kennethreitz/httpbin
    ports:
      - "8000:80"


---
- hosts:
    - app_http
  user: root
  become: true
  become_method: sudo
  roles:
    - docker_install

- hosts:
    - app_http
  user: root
  become: true
  become_method: sudo
  roles:
    - start_app
```