МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

Ігор ШЕЛЕХОВ

(підпис)

червня 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавр

зі спеціальності 122 - Комп'ютерних наук,

освітньо-професійної програми «Інформатика»

на тему: «Інформаційне та програмне забезпечення системи тестового контролю знань»

здобувача групи ІН – 94-1 Якименка Івана Олександровича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Іван ЯКИМЕНКО

(підпис)

Керівник,

кандидат наук, доцент Ігор ШЕЛЕХОВ

(підпис)

Суми – 2023

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри

_____ Ігор ШЕЛЕХОВ
(підпис)

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувача групи ІН-94-1 Якименка Івана Олександровича

1. Тема роботи: «Інформаційне та програмне забезпечення системи тестового контролю знань».
затверджую наказом по СумДУ від *«01» червня 2023 р. № 0475-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 09 червня 2023 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Інформаційний огляд предметної області, постановка й формування завдань дослідження. 2) Огляд засобів для розробки веб-додатків. 3)Огляд засобів автоматизації розгортання. 4) Програмна реалізація додатку 5) Реалізація системи автоматизації для додатку. 6) Розгляд і оцінка результатів.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
6. Консультанти до проєкту (роботи), із значенням розділів проєкту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.

Завдання прийняв до виконання _____ (підпис)

Керівник _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Інформаційний огляд предметної області, постановка й формування завдань дослідження*		
2	*Огляд засобів для розробки веб-додатків. Огляд засобів автоматизації розгортання*		
3	*Програмна реалізація додатку*		
4	*Реалізація системи автоматизації для додатку*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		
6	*Розгляд і оцінка результатів*		

Здобувач вищої освіти _____ (підпис)

Керівник _____ (підпис)

АНОТАЦІЯ

Записка: 117 стр., 56 рис., 23 додаток, 19 використаних джерел.

Обґрунтування актуальності теми роботи – тема кваліфікаційної роботи є актуальною, оскільки присвячена розв’язанню важливих практичних задач: поліпшення навчального процесу та автоматизації розгортання та підтримки програмного забезпечення.

Об’єкт дослідження — система тестового контролю знань та система її автоматичного розгортання і підтримки.

Мета роботи — розробка системи тестового контролю знань та системи її автоматичного розгортання і підтримки.

Методи дослідження — алгоритми аналізу та обробки даних.

Результати — розроблено повністю функціонуючу систему тестового контролю знань та система її автоматичного розгортання, інтеграції змін та підтримки.

ІНФОРМАЦІЙНЕ І ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ ТЕСТОВОГО КОНТРОЛЮ ЗНАНЬ.

ЗМІСТ

ВСТУП

У сучасному світі, де технології стають все більше інтегрованими у різні сфери нашого життя, навчання не залишається осторонь цього процесу. Здобуття знань стає все більш цифровим та вимагає відповідних інструментів для контролю та оптимізації навчального процесу. Останні тенденції світових подій змушують людей підлаштовуватися до нової реальності освітніх процесів, де основною ідеєю є гнучка та продуктивна дистанційна освіта. Серед переваг дистанційного навчання можна виділити:

— Доступність: мінімальні потреби одержувача знань для забезпечення безперервного доступу до ресурсів;

— Гнучкість: у процесі дистанційного навчання одержувач знань має можливість вибудовувати власний алгоритм вивчення матеріалу, освоюючи більшість навичок самостійно;

— Економія грошей та часу: дистанційна освіта не потребує присутності, а тому час, який зазвичай витрачається на переміщення до навчальних інститутів, може бути використаний з користю.

Через актуальність дистанційної освіти, зростає попит на відповідне програмне забезпечення(ПЗ). Найбільш популярними варіантами є інформаційні системи контролю знань у тестовій формі та інформаційні системи для навчальних курсів. Можливість імплементації подібних рішень оптимізують навчальний процес та спрощують оцінювання.

Однак, розробка ПЗ може бути складним процесом, що вимагає координації роботи команди розробників та операторів. Традиційний підхід до розробки та впровадження програмного забезпечення може призвести до проблем, таких як відсутність злагодженості між розробкою та експлуатацією, складність управління інфраструктурою та повільність впровадження змін.

Як і будь-яке ПЗ, системи контролю знань потребують систематизованого та сучасного підходу до розробки та підтримки. Таким

чином, стають затребуваними системи автоматизації розгортання та підтримки додатків.

З метою реалізації подібних систем, з'явилася концепція DevOps, яка є методологією, що поєднує у собі розробку програмного забезпечення та управління операціями з метою створення культури співпраці та автоматизації. DevOps надає можливість автоматизувати процеси збірки, тестування та розгортання програмного забезпечення, що дозволяє забезпечити швидке впровадження змін та зниження ризику помилок.

У даній роботі розглядається розробка додатку, який призначений для проведення тестувань знань та системи автоматизації для його збірки та розгортання інфраструктури. Для досягнення цієї мети будуть використані сучасні підходи до розробки ПЗ та засоби DevOps. Розгляд інструментів DevOps дозволить створити гнучку та автоматизовану інфраструктуру, яка може бути легко відтворена та є простою в обслуговуванні.

Результатом проведеної роботи є інформаційна система, в яку входять: веб-додаток для тестувань знань «Quizzy» і багатокомпонентна система його автоматизованого розгортання та інтеграції в хмарному середовищі.

Робота складається із вступу, аналітичного аналізу, постановки задачі, аналізу засобів розробки додатку та засобів автоматизації розгортання, опису програмної реалізації, висновків, списку використаних джерел та додатків.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Розробка веб-додатків

Сучасні підходи до розробки принесли чималі зміни до створення веб-додатків, пропонуючи ряд потужних інструментів і методів, які підвищують інтерактивність і можливості кінцевих продуктів.

Одним із фундаментальних аспектів розробки сучасних веб-додатків для тестів є використання HTML, CSS і JavaScript як основних технологій. HTML (Hypertext Markup Language) забезпечує структуру та макет вікторини, а CSS (Cascading Style Sheets) дозволяє розробникам визначати візуальне представлення та стиль.

У якості інструментів для імплементації логіки використовують один або декілька із багатьох фреймворків для веб-розробки, які пропонують мови програмування.

Серед фронт-енд фреймворків можна виділити:

— React;
— Vue;
— BootStrap;
— Ember;
— Angular.

Серед бек-енд фреймвоків в основному популярними є:

— Django;
— Ruby On Rails;
— Express;
— Spring;
— ASP.NET Core.

Відносно простим та швидким рішенням з широким функціоналом для розробки веб-дадатку є фреймворк мови Python, Django.

Веб-додатки у більшості випадків використовують клієнт-серверну

архітектуру, стандартними компонентами якої є база даних, сервер додатку або сервер логіки та веб-сервер для відображення веб-контенту, який є результатом роботи додатку.

Із плином часу запити щодо складності та функціональності веб-додатків зростають, а тому змінюються і основні критерії розробки:

- Форма доставки коду;
- Формат взаємодії компонентів додатку;
- Формат розгортання;
- Цільова платформа.

Із появою технології віртуалізації, галузь розробки позбавила себе проблеми залежностей на рівні операційної системи, що надало можливість розгортання того чи іншого ПЗ на декількох віртуальних машинах(ВМ), при цьому маючи лише одну фізичний юніт.

Із поширенням технології контейнеризації, яка базується на функції «cgroups» Linux-систем, погляди на сучасну розробку дещо змінилися. Із поширенням технологій контейнеризації, таких як Docker, Cri-o, Conteinerd та оркестраторами для контейнеризованих додатків, з'явилась можливість переходу від монолітної структури (для складних багатокомпонентних додатків) до мікросервісної структури. Таким чином робота декількох груп розробників над одним додатком являє собою роздільну розробку мікросервісів. Такі нововведення створюють додаткові шари ускладнень на рівні адміністрування та відділу операцій.

Контейнеризація стала невід'ємною частиною сучасного підходу до проєктування, а оркестратори, такі як Kubernetes(K8s), Docker Swarm, HashiCorp Nomad, перестали бути молодими технологіями, створивши новий стандарт для цільових платформ розгортання.

Із активним розвитком сервісів хмарних обчислень, хмарні провайдери надали можливість використовувати платформи оркестрації, як сервіси:

- Google Cloud Platform (GCP) – Google Kubernetes Engine (GKE);
- Amazon Web Services (AWS) – Elastic Kubernetes Service (EKS);

— Microsoft Azure (Azure) – Azure Kubernetes Service (AKS).

1.2 Методологія DevOps

DevOps (Development and Operations) – низка практик, націлених на оптимізацію взаємодії розробників ПЗ та фахівців інформаційного обслуговування, наближуючи їх робочі процеси і вводячи правила тісної співпраці.

Становлення DevOps методології стало поштовхом до появи позиції «DevOps-інженер», який є розробником і фахівцем з адміністрування систем одночасно, що із використанням практик автоматизації робить його єднальним елементом у складному процесі розробки.

DevOps найбільш корисний, якщо розробка ПЗ являє собою частий випуск. Денний цикл випусків, найчастіше, є проблемною темою для організацій, які займаються розробкою та випуском декількох додатків одночасно.

Для виділення основних інструментів методології DevOps, можна сформувати список за спрямуванням застосування:

— Системи контролю версій та менеджменту коду (Git, Mercurial, CVS);

— Інструменти створення «білдів» (Docker, Containerd, Cri-o);

— Інструменти збереження артефактів (Artifactory, Docker Hub, Nexus Repository, AWS S3 bucket, GCP GCS, Azure Blob Storage, JFrog Bintray);

— Системи безперервної доставки та інтеграції (GitLab CI/CD, GitHub Actions, ArgoCD для K8s, Jenkins);

— Інструменти типу «інфраструктура як код» (HashiCorp Terraform, AWS Cloud Formation, Azure Resource Manager, Google Cloud Deployment Manager, Ansible, Puppet, Pulumi, Chef, Vagrant);

— Інструменти безперервного моніторингу (Prometheus, Grafana, Zabbix, Nagios, Datadog, Elastic Stack, Sensu);

— Оркестратори контейнеризованих додатків (K8s, Docker Swarm, Nomad).

1.3 Постановка задачі

Розробити веб-додаток для проведення тестування знань та створити багатокомпонентну систему автоматизації його розробки, розгортання, оновлення та підтримки. Створити автоматизовану систему розгортання платформи для додатку, використовуючи хмарне оточення як цільове.

Для реалізації задачі необхідно виконати такі кроки:

— розробити веб-додаток для проведення тестувань знань;

— дослідити та виконати контейнеризацію додатку;

— створити хмарну інфраструктурну модель;

— розробити конфігурацію хмарної інфраструктури за допомогою інструменту опису інфраструктури;

— розробити систему автоматизації для підготовки середовища розгортання додатку;

— спроєктувати та створити атоматизовану систему взаємодії компонентів

— відтворити стандартну модель роботи системи та перевірити результати.

2 ЗАСОБИ РОЗРОБКИ ДОДАТКУ

2.1 Фреймворк Django

Django – високорівневий та відкритий фреймворк мови програмування Python для швидкого та ефективного створення проєктів в напрямку веб-програмування. Django вважається найкращим фреймворком, написаним на Python і входить до групи найбільш розповсюджених виборів для побудови веб-додатків.

Навколо Django створена широка спільнота, тому фреймворк став активно розвиватися силами ентузіастів.

До усім відомих ресурсів, реалізованих за допомогою Django входять:

— Pinterest,

— Dropbox,

— Spotify,

— The Washington Post.

Веб-сайти Django будуються за допомогою модульної структури. Це одна із вагомих відмінностей в архітектурному рішенні, на відміну від фреймворку Ruby On Rails.

Django спроєктовано із використанням принципу DRY (don’t repeat yourself), що є основою і важливою рисою програмування. Фреймворк дозволяє створювати веб-сайти за допомогою багатьох компонентів і шаблонів, що виключає повторювання коду та допомагає в оптимізації. Django є готовим до високонавантажених додатків.

Як і в багатьох інших фреймворках, Django реалізує Django паттерн MVC (model – view - controller), який часто називають MVT (model – view - template). Таким чином, розробник працює над бізнес-логікою та представленням додатка в окремих модулях. Компоненти MVT можуть бути використані окремо, незалежно один від одного.

Django використовує вбудований шаблонізатор Jinja2, який дозволяє

створювати шаблони із додатковою логікою для генерування сторінок HTML.

Django спочатку розроблявся як інструмент для новинних сайтів, і його архітектура надає багато інструментів для швидкої розробки типових ресурсів.

Розробникам не потрібно створювати контролери або сторінки для адміністративної частини сайту; Django має вбудований модуль управління контентом, який можна включити в будь-який сайт, створений за допомогою Django, що дозволяє керувати багатьма сайтами на одному сервері одночасно. Модуль управління дозволяє створювати, змінювати і видаляти будь-який об'єкт контенту сайту, протоколює всі операції, а також надає інтерфейс для управління користувачами і групами (з призначенням прав доступу).

Документація Django визначає модель як «інформація про дані із ключовими полями та моделями поведінки». Зазвичай, одна модель відповідає одній таблиці в базі даних.

Django офіційно підтримує такі бази даних:

— PostgreSQL;
— MariaDB;
— MySQL;
— Oracle;
— SQLite.

2.2 HTML та CSS

HTML (мова розмітки гіпертексту) і CSS (каскадні таблиці стилів) - це дві основні технології, що використовуються при створенні веб-сторінок для структурування та формування веб-контенту.

HTML є основою веб-сторінки і забезпечує структурну основу для веб-сторінки. Теги та елементи використовуються для опису різних елементів, з яких складається веб-сторінка, таких як заголовки, абзаци, зображення, посилання, форми тощо. Теги HTML беруться у квадратні дужки (< >) і складаються з назви тегу та його вмісту.

CSS використовується для форматування веб-сторінки, формування її зовнішнього вигляду та візуального оформлення. Правила стилів, що містяться у файлі CSS, можна використовувати для визначення кольорів, шрифтів, розмірів, полів, рамок та інших властивостей елементів веб-сторінки. CSS використовується для створення привабливої веб-сторінки та зручного інтерфейсу.

Однією з головних переваг HTML і CSS є те, що вони розділяють структуру і представлення веб-сторінки. Це дозволяє розробникам ефективно визначати структуру контенту за допомогою HTML і легко змінювати зовнішній вигляд сторінки за допомогою CSS без зміни структури сторінки.

HTML і CSS - це стандарти, прийняті Консорціумом Всесвітньої павутини (W3C). Вони підтримуються всіма сучасними браузерами і використовуються мільйонами веб-розробників по всьому світу для створення зручних і привабливих веб-сторінок.

HTML і CSS дуже важливі для веб-розробників, оскільки вони є базовими технологіями для веб-дизайну та розробки і використовуються для створення інтерактивних веб-сайтів, адаптивних мобільних додатків тощо.

2.3 MySQL

MySQL - одна з найпопулярніших і найпоширеніших систем управління базами даних (СКБД). Вона має відкритий вихідний код і є безкоштовною, що робить її привабливим вибором для розробників, яким потрібне надійне і потужне рішення для зберігання та управління даними.

Основними особливостями MySQL є:

— Гнучкість і масштабованість: MySQL можна використовувати для веб-додатків різного розміру та складності, від невеликих проєктів до великих корпоративних систем. Вона підтримує широкий спектр функціональних можливостей і може обробляти великі обсяги даних.

— Багатоплатформенність: MySQL підтримує широкий спектр операційних систем, включаючи Windows, macOS і різні дистрибутиви Linux, що дозволяє використовувати її в різних середовищах.

— Простота використання: MySQL має інтуїтивно зрозумілий інтерфейс та обширну документацію, що робить її зрозумілою для початківців у сфері баз даних. Розробники можуть швидко вивчити основи і почати використовувати MySQL у своїх проєктах.

— MySQL користувач взаємодіє з базою даних за допомогою стандартної мови запитів SQL (Structured Query Language). Це дозволяє розробникам легко створювати запити, оновлювати дані та отримувати результати з бази даних.

— Надійність і безпека MySQL забезпечує механізми резервного копіювання та відновлення даних, щоб запобігти їх втраті у випадку непередбачуваних подій. Вона також підтримує рівні доступу до даних за допомогою користувачів і дозволів, щоб забезпечити безпеку і конфіденційність інформації.

— Масштабованість: MySQL дозволяє розробникам розподіляти навантаження між різними серверами та використовувати кластеризацію для підвищення продуктивності та доступності системи.

MySQL використовується в найрізноманітніших проєктах, від веб-сайтів і блогів до корпоративних систем, електронної комерції та соціальних мереж.

Висока продуктивність і надійність MySQL зробили її однією з найпопулярніших технологій управління базами даних у світі веб-розробки.

2.4 Docker

Docker - це платформа з відкритим вихідним кодом, яка дозволяє розробникам автоматизувати розгортання, масштабування та управління додатками за допомогою контейнеризації. Контейнери - це легкі, автономні, виконувані одиниці, які інкапсулюють програмне забезпечення та всі його залежності, включаючи операційну систему, бібліотеки та середовище виконання. Контейнери забезпечують узгоджене та ізольоване середовище для запуску додатків, що дозволяє їм узгоджено працювати в різних обчислювальних середовищах.

Основні пеерваги використання Docker:

— контейнеризація: Docker використовує технологію контейнеризації для пакування програми та всіх її залежностей у стандартні одиниці, які називаються контейнерами. Контейнери забезпечують узгоджене та повторюване середовище і дозволяють програмам надійно працювати на різних системах.

— мобільність: Контейнери є дуже портативними і можуть працювати на будь-якому хості, де встановлено Docker. Він також інкапсулює весь стек додатків, що полегшує розгортання додатків у різних середовищах, таких як середовище розробки, середовище тестування та «продакшн» середовище.

— ефективність: Контейнери легкі та ефективні, оскільки використовують ядро хост-системи, ізолюючи при цьому процеси додатків. Як результат, вони ефективніше використовують ресурси та скорочують час запуску порівняно з традиційними віртуальними машинами.

— масштабованість: Docker забезпечує горизонтальне масштабування, дозволяючи розгортати додатки в декількох контейнерах. Така гнучкість дозволяє легше справлятися зі зростаючими робочими навантаженнями та забезпечує ефективне використання ресурсів.

— модульність та мікросервіси: Docker підтримує модульний підхід до розробки додатків. Додатки можна розбити на невеликі, незалежні

компоненти, які називаються мікросервісами, кожен з яких працює у власному контейнері. Такий підхід підвищує гнучкість, ремонтопридатність і масштабованість системи в цілому.

— інтеграція з DevOps: Docker відіграє важливу роль у DevOps екосистемі, забезпечуючи узгодженість між середовищами. Контейнери можна легко версіонувати, ділитися та розгортати, що полегшує співпрацю між командами розробників та адміністраторів.

— екосистема та інструменти: Docker має велику екосистему з різноманітними інструментами та сервісами, які покращують робочі процеси контейнеризації. Серед них Docker Compose плагін для управління декількома контейнерними додатками, Docker Swarm оркестратор для організації контейнерів на декількох Docker хостах.

Використовуючи Docker і контейнери, розробники можуть швидше розгортати додатки, спростити управління залежностями, розробляти додаткі легкомасштабованими і поліпшити весь процес розробки і доставки ПЗ.

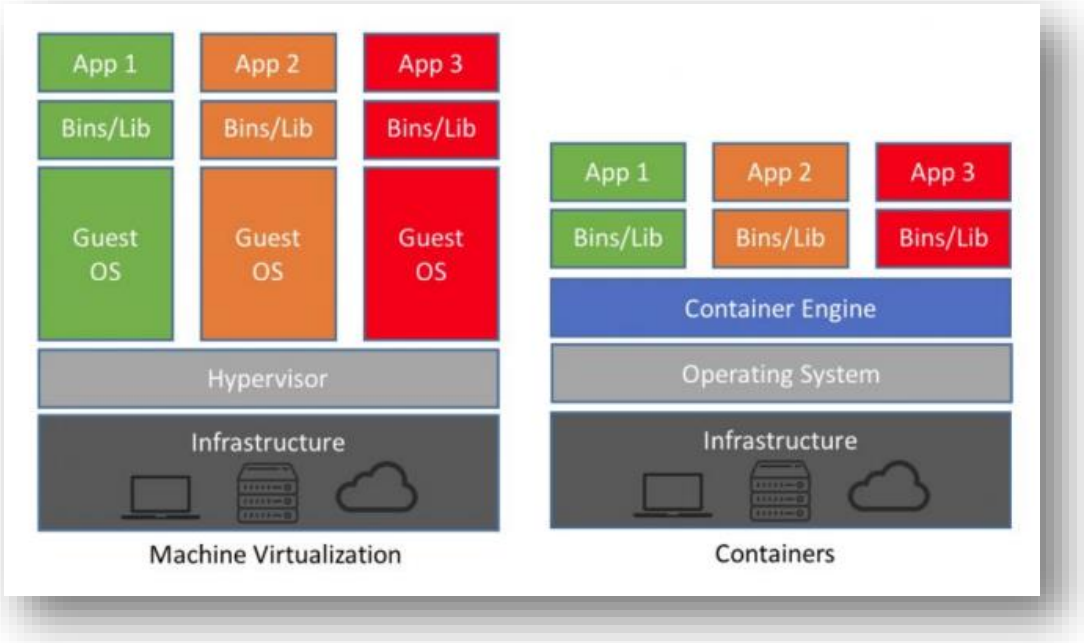


Рис 2.1 – Візуалізація рівнів ізоляції віртуалізації та контейнеризації

Docker використовує архітектуру «клієнт-сервер». Клієнт Docker

взаємодіє з демоном Docker, який виконує роботу створення, запуску та розповсюдження контейнерів Docker. Клієнт і демон Docker можуть працювати в одній системі, можливе підключення клієнта Docker до віддаленого демона Docker. Клієнт Docker і демон взаємодіють за допомогою REST API, через сокети UNIX або мережевий інтерфейс. Іншим клієнтом Docker є Docker Compose, який дозволяє працювати з програмами, що складаються з набору контейнерів.

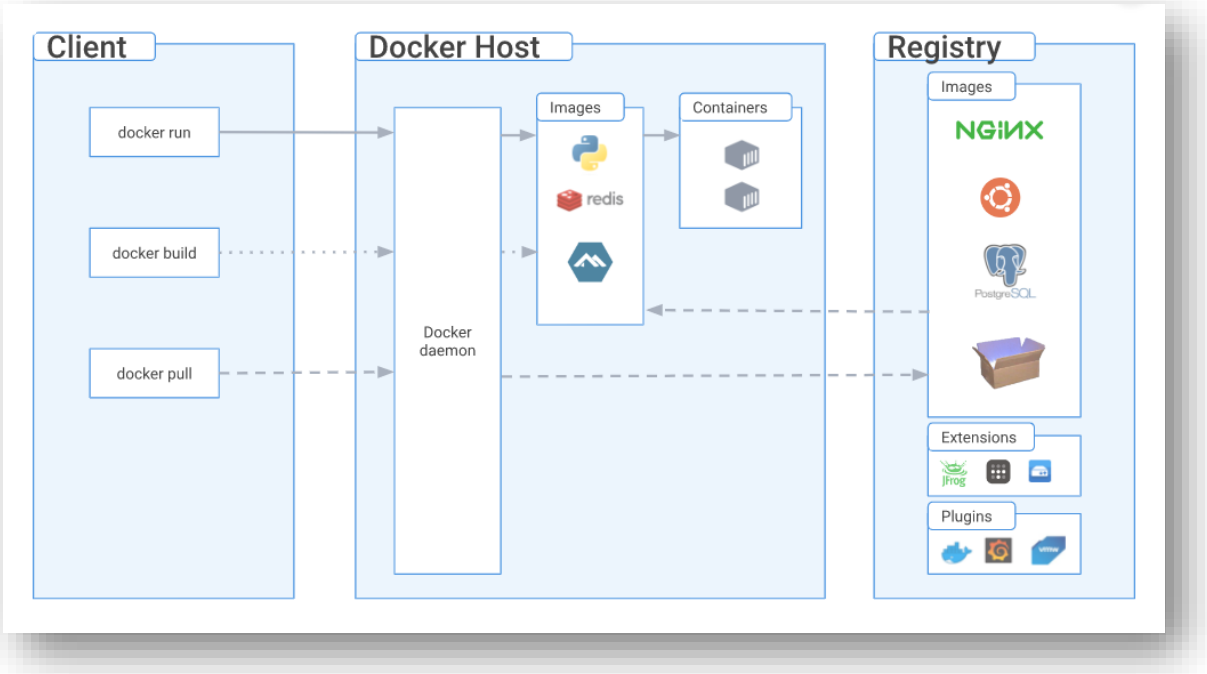


Рис 2.2 – Взаємодія компонентів Docker

Docker образ — це шаблон, доступний лише для читання, з інструкціями щодо створення контейнера Docker. Часто образ базується на іншому образі з деякими додатковими налаштуваннями. Наприклад, можливо створити образ, який базується на образі Ubuntu, але встановлює веб-сервер Apache і програму, а також деталі конфігурації, необхідні для запуску програми.

Користувач має можливість створювати особисті Docker образи за допомогою конфігураційних файлів Dockerfile. Файл являє собою скриптову

інструкцію, де обов’язковим є базовий образ, на основі якого буде створено новий. Таким чином розробник має можливість збирати образи та конфігурувати оточення так, як цього потребує додаток.

Головним сховищем Docker образів, що є публічно доступним є Docker Hub. При виконанні команд Docker: «docker pull» або «docker run», цільовим сховищем для пошуку образів за замовчуванням є саме Docker Hub.

3 ЗАСОБИ АВТОМАТИЗАЦІЇ РОЗГОРТАННЯ

3.1 Docker Compose та Docker Swarm

В той час як Docker standalone надає можливість запускати додатки або сервіси в поодиноких контейнерах, Docker має два потужних інструменти, які покращують управління та оркестрацію контейнерів.

Docker Compose – плагін Docker, що встановлюється окремо і є інструментом, який дозволяє визначати та керувати багатоконтейнерними додатками. Він використовує декларативний YAML-файл для лістингу сервісів, та їх конфігурації. За допомогою Docker Compose можна визначати зв'язки та залежності між контейнерами і легко запускати весь стек додатків за допомогою однієї команди «docker-compose up».

Основні можливості Docker Compose включають:

— Визначення сервісу: Docker Compose дозволяє визначити декілька служб, кожна з яких запускається у власному контейнері. Плагін надає можливості вказати різні параметри, такі як образ, порти, змінні середовища і розділи файлової системи для кожної служби.

— Створення мережі: Compose створює мережу за замовчуванням для кожного створеного додатку, уможливлюючи зв'язок між контейнерами. Розробник може створити власну мережу для ізоляції та контролю зв'язку між сервісами.

— Керування гучністю: Compose надає простий спосіб керування розділами даних для постійного зберігання. Плагін надає можливості створення іменованих розділів або прив'язувати локальні директорії до контейнерів.

— Налаштування середовища: Compose дозволяє встановлювати змінні оточення для сервісів, що дозволяє легко налаштовувати поведінку контейнерів без зміни основного коду.

Docker Swarm - це рішення Docker для кластеризації та оркестрування. Воно дозволяє створювати та керувати кластером Docker-вузлів, перетворюючи їх на розподілений обчислювальний ресурс. Swarm дозволяє розгортати і масштабувати додатки на декількох машинах, забезпечуючи високу доступність і відмовостійкість.

Основні можливості Docker Swarm включають:

— Режим Swarm: використовує вбудовану функцію режиму кластеризації Docker, яка перетворює групу хостів Docker в кластер. Цей кластер складається з вузлів-менеджерів, які керують оркестрацією та робочих вузлів, які виконують задачі запуску контейнери.

— Масштабування сервісів: Swarm дозволяє легко масштабувати сервіси вертикально або горизонтально, щоб задовольнити непостійні вимоги. Swarm розподілить контейнери між доступними вузлами в залежності від сконфігурованої кількості реплікацій.

— Балансування навантаження: Swarm включає вбудований розподілювач навантаження, який розподіляє вхідні запити між контейнерами, що виконують задачі сервісу. Це гарантує рівномірний розподіл навантаження і забезпечує високу доступність.

— Оновлення за принципом «rolling update»: Swarm підтримує оновлення на ходу, що дозволяє оновлювати сервіси без простоїв. Оновлювати контейнери можливо за декількома стратегіями, наприклад, один за одним, гарантуючи, що додаток залишатиметься доступним під час процесу оновлення.

— Swarm Config: Swarm надає спосіб керувати конфігураційними файлами, підключаючи їх до сервісів. Конфігурації надійно зберігаються і надаються лише службам, які мають відповідний доступ.

Docker Swarm спрощує управління та масштабування контейнерних додатків, надаючи уніфікований та інтуїтивно зрозумілий інтерфейс для управління кластером вузлів Docker.

І Docker Compose, і Docker Swarm сприяють загальному процесу розробки та розгортання, дозволяючи розробникам визначати складні мультиконтейнерні додатки та ефективно їх організовувати.

3.2 AWS і автоматизація розгортання у хмарному середовищі

Amazon Web Services (AWS) - це комплексна і широко використовувана платформа хмарних обчислень від Amazon, що надає широкий спектр хмарних сервісів та інструментів для створення, розгортання та управління різними типами додатків і сервісів. До основних переваг AWS можна віднести:

Масштабованість і гнучкість: AWS надає високомасштабовану інфраструктуру, яка дозволяє масштабувати додатки на вимогу за допомогою таких сервісів, як Amazon EC2 (Elastic Compute Cloud) і автоматичного масштабування, обчислювальні ресурси можна легко додавати або видаляти в міру зміни потреб додатків. Така масштабованість дозволяє додаткам справлятися із різкими стрибками трафіку та ефективно використовувати ресурси.

Обчислювальні сервіси AWS пропонує ряд обчислювальних сервісів, включаючи Amazon EC2, AWS Lambda, AWS Batch та AWS Fargate; EC2 дає повний контроль над обчислювальним середовищем, надаючи віртуальні сервери в хмарі. Lambda - це безсерверний обчислювальний сервіс, який дозволяє коду працювати без керування серверами, що робить його ідеальним для архітектурних рішень, керованих подіями, та мікросервісів;

Сховища та бази даних AWS пропонує кілька варіантів сховищ і баз даних для задоволення різних потреб Amazon S3 (Simple Storage Service) забезпечує масштабоване сховище об'єктів; Amazon EBS (Elastic Block Store) - постійне сховище на рівні блоків для екземплярів EC2; і Amazon S3 (Simple Storage Service) - масштабоване сховище і база даних для зберігання і пошуку будь-якого обсягу даних. Для баз даних AWS пропонує Amazon RDS (Relational Database Service) для керованих реляційних баз даних, Amazon DynamoDB для баз даних NoSQL, Amazon Redshift для зберігання даних та інші послуги.

Мережеві сервіси та сервіси розповсюдження контенту: AWS пропонує мережеві сервіси, які забезпечують безпечні та масштабовані з'єднання між

ресурсами; Amazon VPC (Virtual Private Cloud) дозволяє створювати ізольовані віртуальні мережі в хмарі. AWS Direct Connect забезпечує приватне мережеве з'єднання між локальними середовищами та AWS. Також надається доступ до сервісів ELB(Elastic Load Balancing), EIP (Elastic IP) та інші.

Як хмарна платформа, AWS надає організаціям масштабовану, надійну та гнучку інфраструктуру для впровадження інновацій та прискорення цифрової трансформації. AWS пропонує багатий набір послуг та глобальну присутність, що дозволяє організаціям зосередитися на своїх додатках, насолоджуючись перевагами хмарних обчислень.

За замовчуванням, користувачу AWS доступні декілька варіантів керування інфраструктурою:

— Консоль управління AWS: використання веб-інтерфейсу, який дозволяє вручну контролювати та керувати інфраструктурою;

— Командний рядок AWS: використання інтерфейсу командного рядка, що дозволяє виконувати команди по керуванню ресурсами, автоматизувати задачі на деякому рівні;

— Інструменти «інфраструктури як код»: AWS пропонує власний сервіс AWS CloudFormation, як рішення для створення конфігурації за допомогою коду та повну автоматизацію процесів керування хмарною інфраструктурою.

Можливе використання сторонніх інструментів, так як Terraform (TerraFormation) розробки HashiCorp, що є найбільш частим рішенням, бо є універсальним для багатьох цільових платформ, має широкий функціонал та спільноту.

3.3 Підхід «Інфраструктура як код»

Інфраструктура як код (IaaC) - це підхід до управління інфраструктурою, при якому ресурси інфраструктури (наприклад, сервери, мережі, бази даних) визначаються як код для автоматизації їх розгортання, конфігурації та управління. Основними принципами кодового підходу є:

— Декларативність: інфраструктурний код описує бажаний стан системи, а не послідовність кроків, необхідних для його досягнення. Він описує бажаний кінцевий стан інфраструктури, а не те, як будуть створені або налаштовані ресурси. Це дозволяє системі автоматично забезпечувати відповідність фактичного стану системи опису в коді;

— Версії та контроль версій: код інфраструктури може зберігатися в системі контролю версій, такій як Git. Це дозволяє відстежувати зміни в інфраструктурі, повертатися до попередніх версій та співпрацювати з іншими членами команди. Версіонування коду інфраструктури забезпечує послідовність і повторюваність управління інфраструктурою;

— Автоматизація: можливість автоматизувати розгортання, конфігурацію та управління інфраструктурою. Це знижує ризик помилок і підвищує швидкість і надійність управління інфраструктурою. Автоматизовані процеси також дозволяють швидше реагувати на зміни та ефективніше використовувати ресурси;

— Повторюваність: Інфраструктура як код дозволяє створювати код, який може працювати на різних платформах і хмарних сервісах. Це забезпечує гнучкість і можливість вибору середовища розгортання.

— Стабільність і надійність: представлення інфраструктури у вигляді коду забезпечує узгодженість і повторюваність в управлінні. Це запобігає людським помилкам і забезпечує стабільну та надійну роботу системи;

— Швидкість розгортання: автоматизоване розгортання з використанням інфраструктурного коду дозволяє швидко та ефективно створювати та налаштовувати нові ресурси;

— Масштабованість: завдяки інфраструктурному коду систему легко розширювати, а ресурси можна додавати або видаляти за потреби;

— Історія та аудит: можливість тримати код у системах контролю версій дозволяє відстежувати зміни та забезпечує аудиторський слід історії розгортання та конфігурації системи.

«Інфраструктура як код» революціонізує сучасні практики розробки, дозволяючи швидше, надійніше та ефективніше розгортати інфраструктуру та керувати нею. Завдяки цьому, у розробника більше часу для бізнес-цілей та інноваціях замість витрат часу на ручне керування інфраструктурою.

Загалом, інструменти підходу «Інфраструктура як код» можна поділити на типи:

— Інструменти менеджменту конфігурацій – часто використовують імперативно-декларативний формат, замість декларативного, часто використовуються саме для конфігурування створених ресурсів або керування ними після створення (Ansible);

— Інструменти управління інфраструктурою – використовуються для безпосереднього створення, зміни, знищення інфраструктури, як платформи для тих чи інших дій (Terraform).

Правильний підхід до поділу зон відповідальності відповідних інструментів є запорукою продуктивності у роботі із інфраструктурою, де є місце модульності, ідемпотентності та передбачуваності.

3.3.1 HashiCorp Terraform

HashiCorp Terraform - це інструмент підходу "Інфраструктура як код", який дозволяє описувати як хмарні, так і локальні ресурси за допомогою коду в зрозумілих для людини конфігураційних файлах, які можна змінювати, повторно використовувати і використовувати для розгортання. Terraform забезпечує послідовний робочий процес для представлення та управління всією інфраструктурою протягом її життєвого циклу. Terraform однаково добре справляється як із високорівневими, так і низькорівневими компонентами.

Terraform може створювати ресурси в хмарних середовищах та інших сервісах і керувати ними за допомогою API. Залежно від провайдера, Terraform може працювати практично з будь-якою платформою або сервісом з доступним йому API.

Рисунок 3.1 - Взаємодія між Terraform та API

Завдяки співпраці між HashiCorp і спільнотою Terraform, велика кількість провайдерів були написані для управління різними типами ресурсів і сервісів: Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, DataDog та інші.

Основний робочий процес Terraform складається з трьох етапів:

— Підготовка коду розробником: визначення ресурсів, які можуть бути доступні в різних хмарних середовищах.

— Планування: На основі існуючої інфраструктури та конфігурації, представленої кодом, Terraform створює план виконання, який визначає інфраструктуру, що має бути створена, модернізована або знищена.

— Виконання змін: після затвердження Terraform виконує запропоновані операції в правильному порядку з урахуванням ресурсних залежностей.

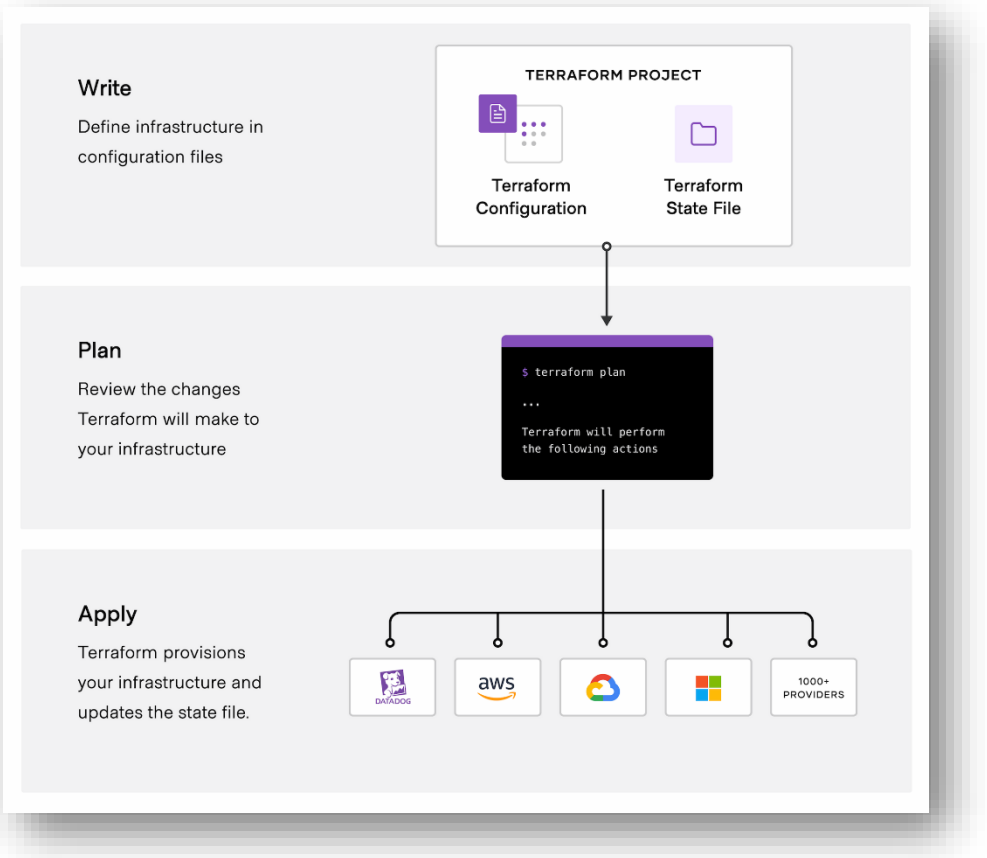


Рисунок 3.2 – Основні етапи роботи Terraform

Стан Terraform (TF State) - це файл або набір файлів, що зберігає інформацію про стан ресурсів, якими керує Terraform. Цей стан є важливим елементом для Terraform і використовується для визначення поточного стану

інфраструктури та визначення змін, необхідних для досягнення бажаного стану.

Неправильно визначений стан Terraform призводить до проблем із відсутністю відповідності реальних ресурсів та «уявленням» Terraform про них, така ситуація має назву «State Drift». Для вирішення такої проблеми потребуються додаткові маніпуляції, наприклад, «terraform import».

Основні особливості та функціональні можливості Terraform State полягають у наступному:

— Сховище стану Terraform State містить інформацію про всі керовані ресурси, їх стан та конфігурацію. Сюди входять ідентифікатори ресурсів, значення атрибутів та зв'язки між ресурсами у формі графів залежностей;

— Моніторинг залежностей: Terraform State відстежує залежності між ресурсами, що дозволяє точно визначити порядок створення, оновлення або видалення ресурсів;

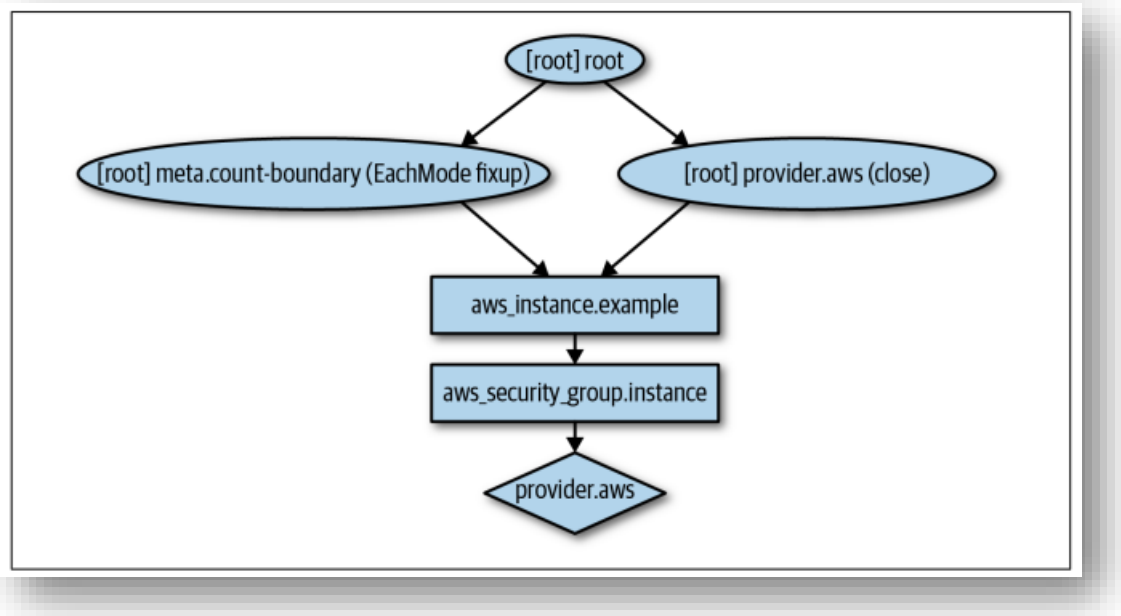


Рис 3. 3 – Приклад графу залежності ресурсів Terraform

— Планування змін: Terraform використовує Terraform State для

порівняння поточного стану з бажаним і визначення змін, необхідних для досягнення бажаного стану інфраструктури. Це дозволяє Terraform створювати плани, які вказують, які зміни необхідно внести в інфраструктуру;

— Безпека та конфіденційність: Terraform State може містити конфіденційну інформацію, таку як паролі та ключі доступу. Для забезпечення безпеки ці дані можна зашифрувати або використовувати зовнішню систему для управління конфіденційною інформацією;

— Terraform State можна зберігати локально на комп'ютері розробника або за допомогою різних рішень для зберігання даних, таких як Terraform Cloud, Amazon S3, Consul і т.д. Це дозволяє співпрацювати з декількома розробниками і забезпечує безпеку і цілісність стану інфраструктури.

HashiCorp Configuration Language (HCL) - це мова конфігурації, яка використовується для опису інфраструктури в Terraform. HCL - це спрощена, відкрита і декларативна мова, яка дозволяє розробникам явно визначати структуру і параметри ресурсів інфраструктури.

Основними особливостями HCL є:

— Простота: HCL має простий, зручний у використанні синтаксис, що сприяє швидкому вивченню мови та зменшує кількість помилок при написанні конфігураційних файлів;

— Декларативний підхід: HCL дозволяє явно описати бажаний стан ресурсу інфраструктури, а не вказувати кроки, необхідні для досягнення цього стану. Це спрощує розробку та управління інфраструктурою;

— Модульність: HCL підтримує використання модулів, що дозволяє створювати багаторазові конфігураційні блоки. Модулі уможливлюють стандартизацію та швидке впровадження окремих частин інфраструктури;

— Ефективність: HCL надає можливість використовувати вирази, змінні та функції для більш гнучкого та динамічного визначення параметрів конфігурації;

— Інтеграція з провайдерами: HCL підтримує інтеграцію з різними

провайдерами, включаючи AWS, Azure та GCP. Це дозволяє використовувати їхні ресурси та функціональність у конфігураційних файлах HCL;

— Підтримка коментарів: HCL дозволяє додавати коментарі до файлів конфігурації для полегшення документування та співпраці розробників.

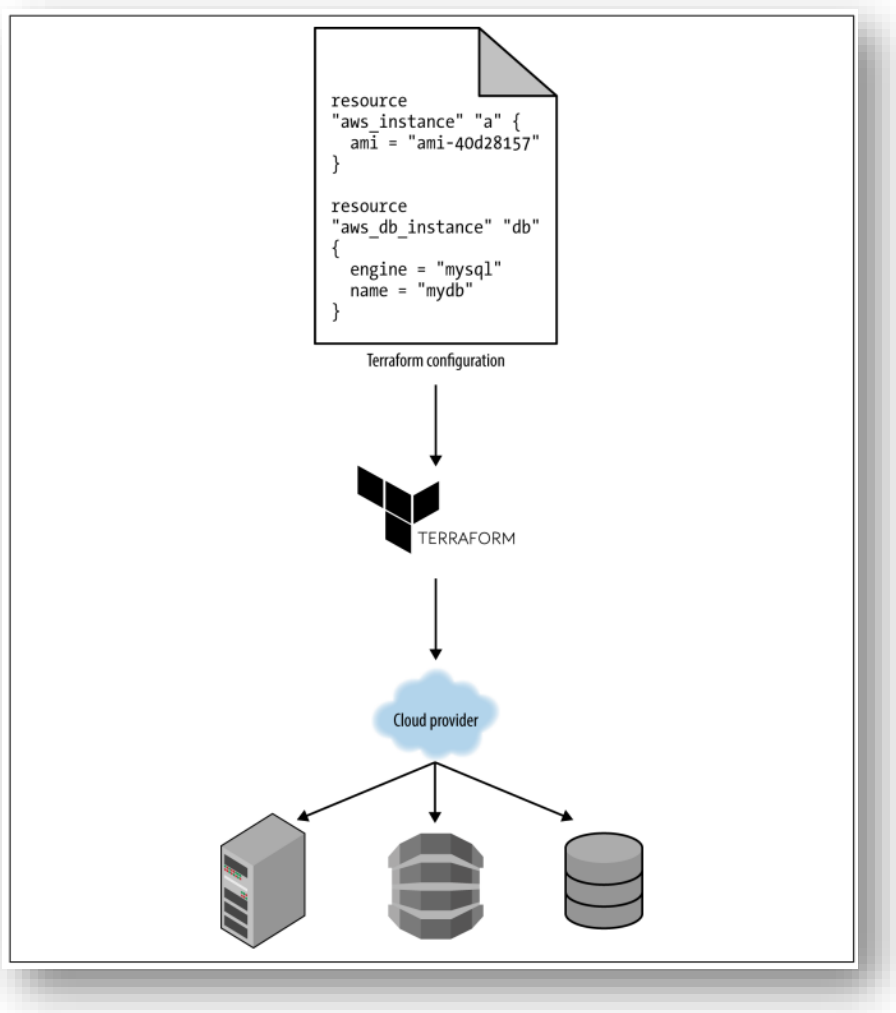


Рис. 3.4 - Принцип створення інфраструктури за допомогою коду Terraform

3.3.2 Ansible

Ansible є інструментом автоматизації, який дозволяє здійснювати конфігурацію та управління системами. Ansible використовує декларативний підхід і базується на мові YAML для визначення конфігураційних файлів. Ansible надає можливості розгортання та управління конфігурацією через підключення за допомогою протоколу Secure Shell(SSH).

Ansible - один з найпопулярніших інструментів автоматизації сьогодні з кількох причин:

— Простота використання: Завдяки простому, легкому у використанні синтаксису Ansible, його можна швидко вивчити для автоматизації конфігурації та управління системою; Ansible використовує декларативний підхід;

— "Безагентна" архітектура: Ansible не вимагає встановлення агентів або додаткового програмного забезпечення на цільовій системі; управління здійснюється через SSH, що спрощує розгортання та полегшує адміністрування;

— Ідемпотентність: архітектура Ansible базується на принципі виконання лише необхідних кроків для досягнення бажаного стану;

— Широка підтримка платформ: Ansible підтримує широкий спектр операційних систем і хмарних платформ, включаючи Linux, Windows, AWS, Azure і GCP. Це робить його універсальним інструментом для автоматизації широкого спектру задач;

— Інфраструктура як код: Ansible дозволяє записувати конфігурації системи у вигляді коду. Це означає, що код можна зберігати у системах контролю версій версії та повністю відтворювати інфраструктуру за допомогою файлів коду, що полегшує розгортання та управління інфраструктурою.

— Велика спільнота та підтримка: Ansible має велику та активну спільноту, яка постійно вдосконалює систему, розробляє різноманітні модулі та розширення, а також надає підтримку та допомогу користувачам. Це дозволяє швидко знаходити відповіді на питання та вирішувати проблеми, пов’язані з розробкою.

Основними поняттями Ansible є:

— Control node (вузол керування) – хост, де виконується Ansible скрипти;

— Managed nodes (керовані вузли) – цільові хости, з якими працює Ansible;

— Inventory (інвентар)– файл конфігурації для опису цільових хостів;

— Plays – контекст для виконання Ansible;

— Playbooks – головна одиниця виконання Ansible скрипта, являє собою файл з декларованими підключеннями компонентів Ansible;

— Roles (ролі) – юніти, що являть собою одиниці для імпорту і використання;

— Tasks(задачі) – визначення дій, які повинні бути виконані на керованому вузлі.

— Вузол керування: будь-який вузол, на якому встановлено Ansible.

Вузол керування - це мережевий пристрій (та/або сервер), яким керує Ansible. Керований вузол іноді називають "хостом". Ansible не встановлюється на керованому вузлі. Загалом, керований вузол має бути доступним через ssh з керуючого вузла і на ньому має бути встановлений Python.

Інвентар - список керованих вузлів. Файл інвентаризації іноді також називають "хост-файлом".

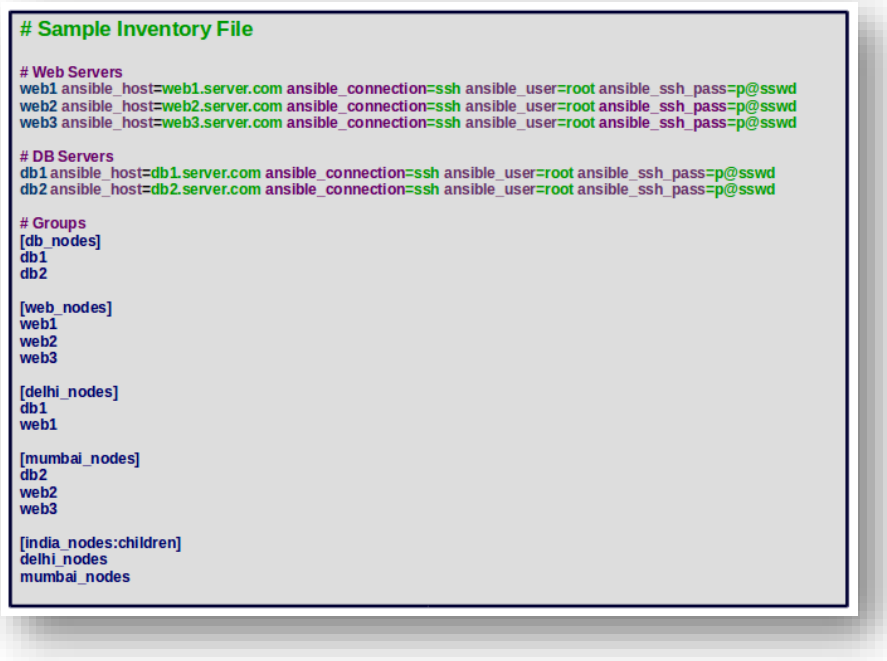


Рис. 3.5 – Приклад Ansible інвентаря

Інвентаризаційний файл може містити таку інформацію, як IP-адреса кожного керованого вузла, конфігурація SSH-підключень тощо. Інвентаризаційний файл також може впорядковувати керовані вузли, створюючи вкладеність груп для полегшення керування. Файли інвентаризації можуть бути у різних форматах, зокрема YAML та INI.

Запуск Ansible є виконанням команди «ansible-playbook», якій передають аргумент – ім’я файлу, де визначені послідовності задач. Для виконання задач потрібно правильно сконфігурувати інвентар та мати доступ до цільових хостів через протокол SSH.

Для зручності роботи, можна використовувати структуру директорій Ansible-galaxy, імплементуючи чітке упорядкування ролей Ansible.

```
yakimoro@wks4:~$ ansible-galaxy init example
- Role example was created successfully
```

Рис. 3.6 – Приклад використання команди для створення ролі Ansible

```
yakimoro@wks4:~$ tree example/
example/
├── README.md
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml

8 directories, 8 files
```

Рис. 3.7 – Структура директорій ролі Ansible

Завдяки такому підходу, до автоматизації з Ansible можна ставитися, як до модулів коду, створюючи ролі та підключаючи їх у різні сценарії для застосувань у різних середовищах.

Технології Ansible і Terraform не є взаємозамінними і можуть використовуватися разом. У багатьох випадках інфраструктура поділяється на зони відповідальності і закріплюється за Ansible і Terraform відповідно: Terraform використовується для створення інфраструктурної бази для розгортання, в той час як Ansible використовується для автоматизації конфігурації та встановлення програмного забезпечення.

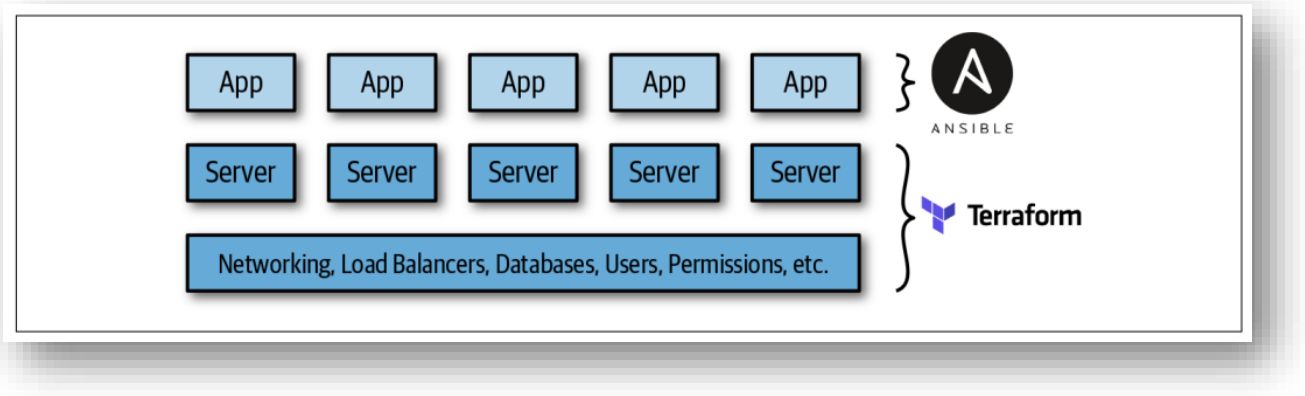


Рис. 3.8 - Розподіл "зон відповідальності" Ansible та Terraform

Для найбільш ефективної взаємодії двох технологій можливо створювати динамічні інвентарі Ansible на основі вихідних даних відпрацювання Terraform, автоматизуючи послідовність виконання задач двох технологій і об’єднуючи їх в ланцюг застосувань.

3.4 Gitlab, Gitlab CI/CD

GitLab - це веб-платформа DevOps, яка надає повний набір інструментів для управління життєвим циклом розробки програмного забезпечення. Вона побудована із використанням Git, популярній розподіленій системі контролю версій, і пропонує широкий спектр функцій, включаючи управління вихідним кодом, відстеження проблем, безперервну інтеграцію та доставку (CI/CD) і багато іншого.

Одним з ключових компонентів GitLab є вбудовані можливості CI/CD, відомі як GitLab CI/CD. CI/CD розшифровується як Continuous Integration (безперервна інтеграція) та Continuous Delivery/Deployment (безперервна доставка/розгортання), які є важливими практиками в сучасній розробці програмного забезпечення.

GitLab CI/CD дозволяє розробникам автоматизувати процес створення, та розгортання своїх додатків у формі артефактів.

GitLab CI/CD дозволяє розробникам регулярно інтегрувати зміни коду в загальний репозиторій. За допомогою конвеєрів (pipelines) GitLab CI/CD розробники можуть визначати завдання і робочі процеси, які автоматично збирають їхній код щоразу, коли зміни потрапляють до сховища. Це допомагає виявити проблеми інтеграції на ранніх стадіях і сприяє співпраці між членами команди.

GitLab CI/CD виходить за рамки безперервної інтеграції, надаючи засоби для автоматичного розгортання додатків у різних середовищах. Завдяки конфігурації конвеєра GitLab розробники можуть визначати етапи розгортання і вказувати умови для просування змін коду в різні середовища. Це дозволяє командам швидко та надійно випускати оновлення програмного забезпечення.

GitLab CI/CD інтегрується з інструментами інфраструктури, такими як Terraform та Kubernetes, що дозволяє командам керувати своєю інфраструктурою як кодом. Визначаючи конфігурації інфраструктури у файлах з контролем версій, розробники можуть легко створювати або видаляти ресурси інфраструктури в

рамках своїх конвеєрів CI/CD. Це сприяє узгодженості та відтворюваності при розгортанні додатків.

GitLab CI/CD надає надійну систему управління середовищем, що дозволяє командам визначати та керувати різними середовищами для своїх додатків. Розробники можуть налаштовувати певні змінні, секрети та конфігурації для кожного середовища, забезпечуючи послідовне розгортання додатків на різних етапах.

GitLab CI/CD пропонує вбудовані механізми моніторингу та зворотного зв'язку для відстеження стану та прогресу конвеєрів CI/CD. Розробники можуть переглядати детальні логи, звіти про тести та показники продуктивності, щоб виявити проблеми та усунути збої. Це допомагає командам забезпечувати якість та надійність своїх релізів програмного забезпечення.

GitLab CI/CD легко інтегрується з іншими інструментами та сервісами DevOps, такими як Docker, Terraform, AWS, Kubernetes тощо. Він надає API, що дозволяє командам розширювати функціональність та інтегрувати з існуючими робочими процесами та інструментами.

Для виконання конвеєрних задач, GitLab CI/CD використовує юніти під назвою Gitlab Runners, що є агентами GitLab для виконання поставлених задач, тобто, виступають середовищами виконання. Gitlab надає можливість використати загальнодоступного агента, або створити особистий, використовуючи персональні потужності для виконання задач. Існують багато типів агентів, але найбільш використовуваними є:

— агент типу «shell», який використовує оточення хоста, на якому його було зареєстровано;

— агент типу «docker», який використовує оточення вказаного Docker образу для кожної стадій конвеєру.

Для опису конвеєрів використовується файл, що за замовченнням називається «.gitlab-ci.yml». Файли конвеєрів використовують синтаксис yaml та надають можливості опису виконання послідовних задач при умові доступності агента GitLab. Якщо файл «.gitlab-ci.yml» потрапляє до репозиторія Git, GitLab

зчитує інструкції із нього та ініціює процес планування виконання на доступному йому агенті. GitLab може використовувати декілька агентів одночасно, а для специфікації виконання конвеєра на конкретному агенті, використовується спеціальний тег (тег в конфігурації агента та тег в конвеєрі), який дозволяє керувати плануванням задач для конкретних агентів. Синтаксис «.gitlab-ci.yml» для різних агентів є різним.

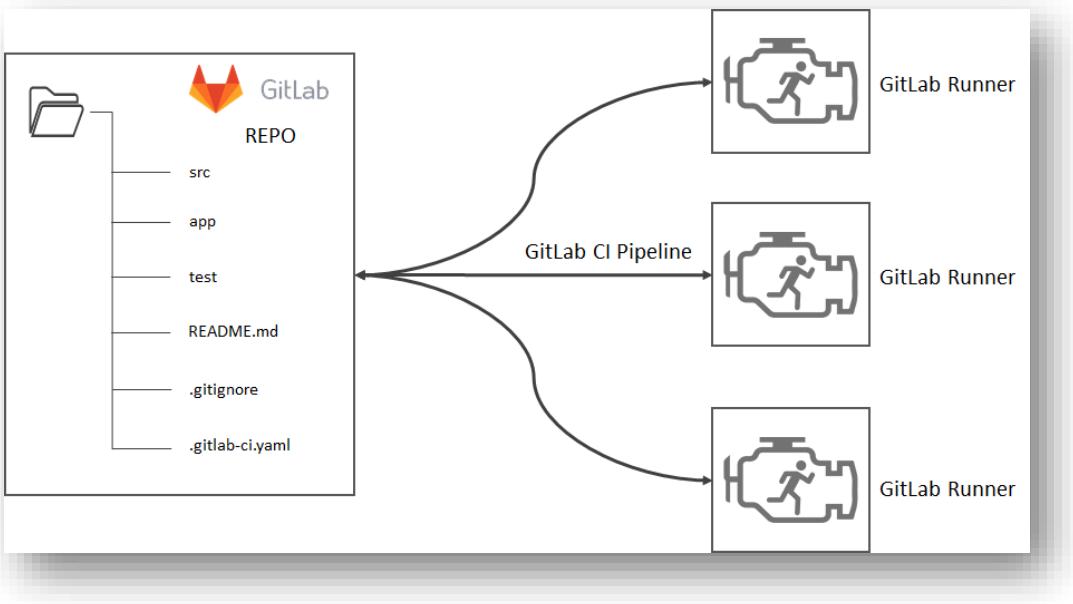


Рис. 3.9 – Візуалізація розподілу задач між декількома агентами GitLab

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ДОДАТКУ

4.1 Проєктування додатку

Зазвичай проєктування додатку починається із створення моделі даних для бази даних, аналізуючи поставлені задачі.

4.1.1 Діаграма бази даних

Для реалізації додатку не потребується складна схема даних, тому задля впровадження можливості додавання та доступу для тестувань, було вирішено використати 3 таблиці в базі даних.

Сутність Quiz та її атрибути:

— name: Відображає назву Тесту. Це CharField з максимальною довжиною 100 символів.

— short_description: Короткий опис Тесту. Це CharField з максимальною довжиною 200 символів.

— long_description: Відображає довгий опис Тесту, текстове поле.

Сутність Question та її атрибути:

— quiz: Представляє зв'язок зовнішнього ключа з сутністю Quiz, вказуючи, що кожне питання належить до певного Тесту.

— content: Відображає вміст запитання, текстове поле.

Сутність Answer та її атрибути:

— question: Представляє зв'язок зовнішнього ключа з сутністю Питання, вказуючи, що кожна відповідь належить до певного питання.

— content: Відображає вміст відповіді. Це текстове поле.

— is_correct: Відображає, чи є відповідь правильною чи ні. Це булеве поле.

Зв'язки:

— Сутність Quiz має зв'язок "один-до-багатьох" з сутністю Question, на що вказує поле зовнішнього ключа quiz у моделі Question.

— Сутність Question має зв'язок один-до-багатьох з сутністю Answer, на що вказує поле зовнішнього ключа question у моделі Answer.

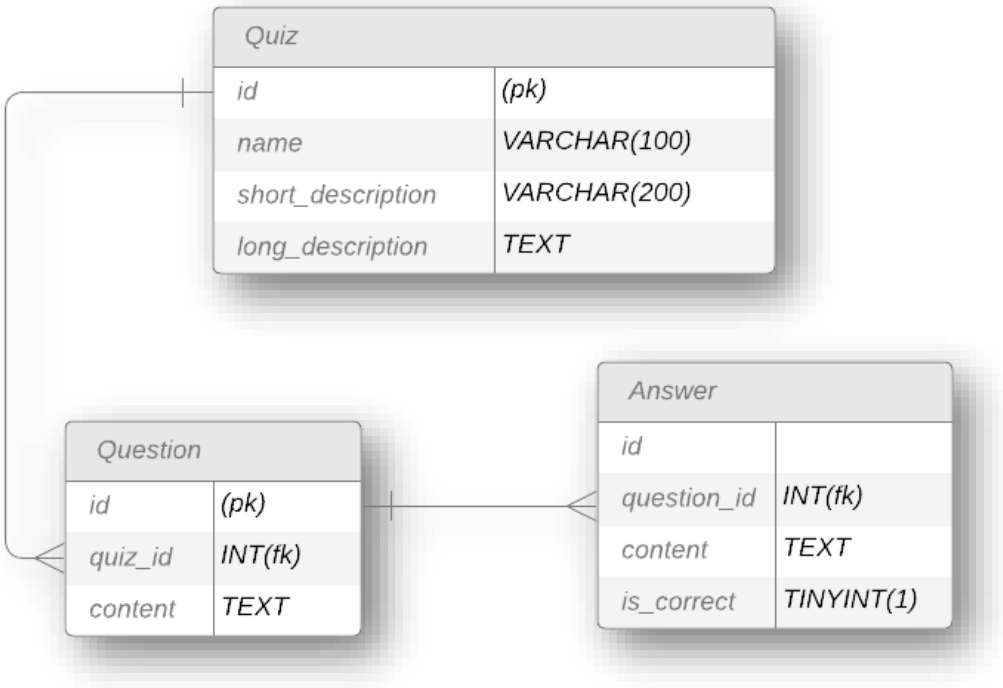


Рис. 4.1 – Діаграма бази даних

4.1.2 «Use case» діаграма

Мета «use case» діаграми – створення візуалізації динамічного аспекту системи для збору вимог, отримання загального вигляду системи, відслідковування взаємодій вимог та об’єктів.

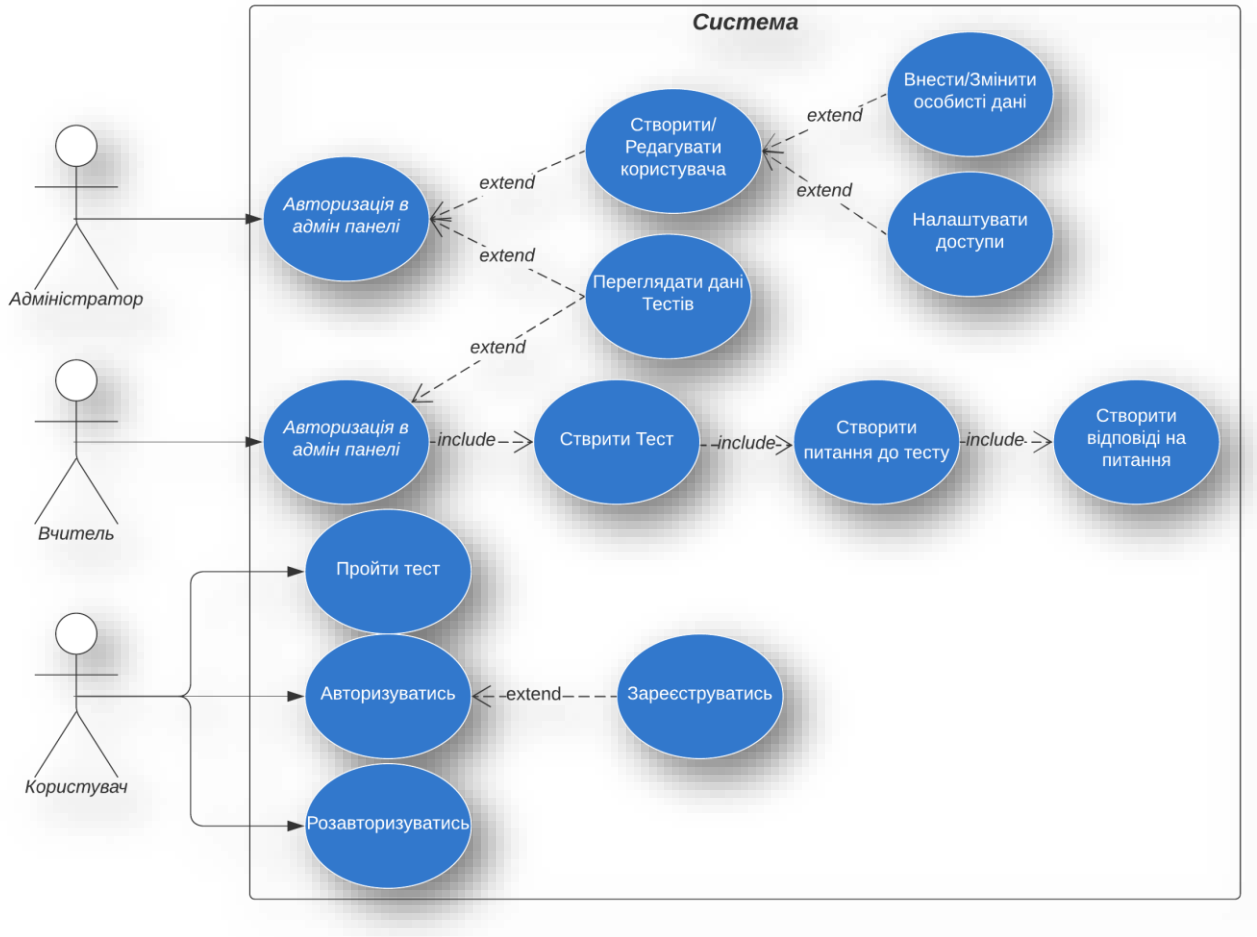


Рис. 4.2 – Діаграма «Use case»

4.2 Написання коду додатку

Подання (View): у коді визначено декілька функцій представлень, які обробляють HTTP-запити та повертають HTTP-відповіді. Ці подання відповідають за відображення шаблонів, обробку даних форми та перенаправлення користувачів. До них відносяться:

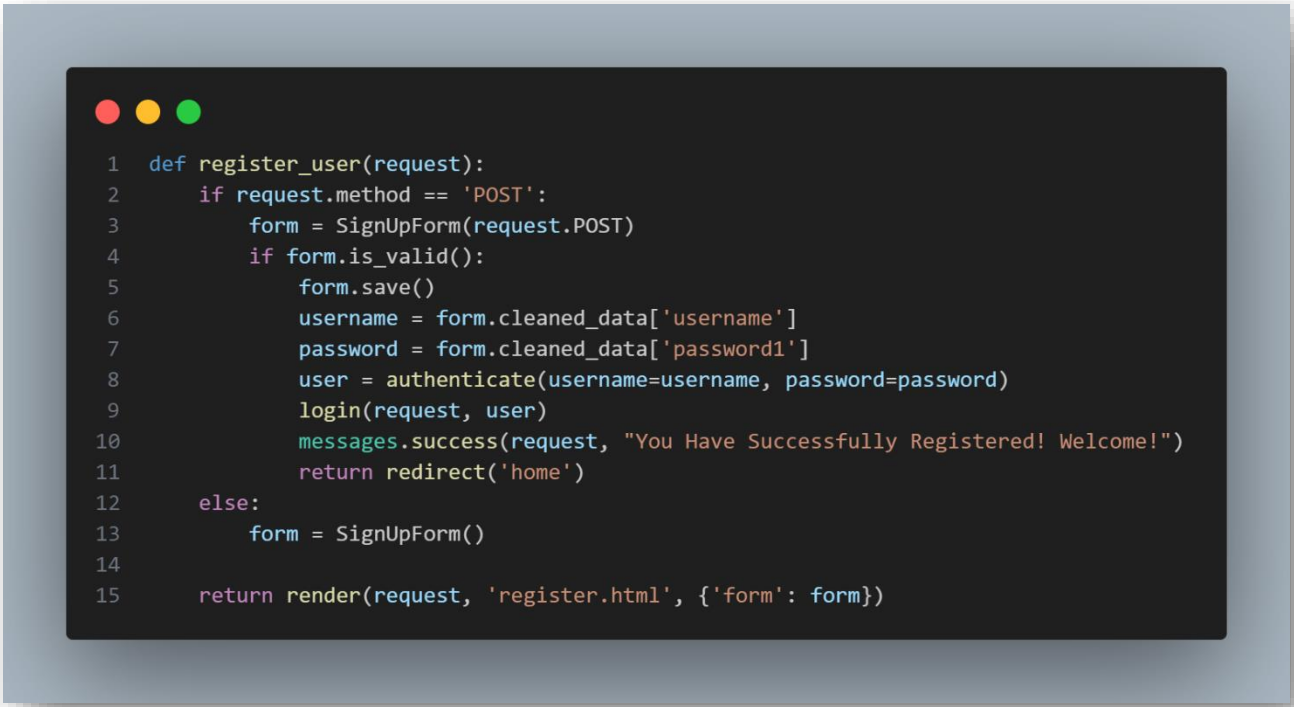
— `home`: Відображає шаблон домашньої сторінки та обробляє автентифікацію користувача.

```python
def home(request):
    quizzes = Quiz.objects.all()

    # Check if the user is authenticated
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            messages.success(request, "You Have Been Logged In!")
            return redirect('home')
        else:
            messages.success(request, "There Was An Error Logging In, Please Try Again...")
            return redirect('home')

    return render(request, 'quiz_home.html', {'quizzes': quizzes})
```

Рис. 4.3 – Код подання «home»

— `logout_user`: Виводить користувача з системи і перенаправляє на головну сторінку.

— `register_user`: Обробляє реєстрацію користувача, перевірку форми та автентифікацію.



```python
def register_user(request):
    if request.method == 'POST':
        form = SignUpForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data['username']
            password = form.cleaned_data['password1']
            user = authenticate(username=username, password=password)
            login(request, user)
            messages.success(request, "You Have Successfully Registered! Welcome!")
            return redirect('home')
    else:
        form = SignUpForm()

    return render(request, 'register.html', {'form': form})
```

Рис. 4.4 – Код подання «register_user»

— `quiz_list`: Відображає сторінку зі списком тестів, якщо користувач автентифікований, інакше перенаправляє на головну сторінку.

— `quiz_info`: Відображає сторінку з інформацією про вікторину для певної вікторини.

— `quiz_start`: Відображає стартову сторінку вікторини з питаннями для певного тесту.

— `quiz_result`: Обчислює результат тесту на основі надісланих відповідей користувача та виводить сторінку з результатом.

```python
def quiz_result(request, quiz_id):
    if request.user.is_authenticated:
        quiz = get_object_or_404(Quiz, id=quiz_id)
        questions = Question.objects.filter(quiz=quiz)
        total_questions = questions.count()
        correct_answers = 0

        for question in questions:
            selected_answer_id = request.POST.get(f'answer_{question.id}')
            if selected_answer_id:
                selected_answer = get_object_or_404(Answer, id=selected_answer_id)
                if selected_answer.is_correct:
                    correct_answers += 1

        grade = (correct_answers / total_questions) * 100 if total_questions > 0 else 0
        grade = round(grade, 2)  # Round grade to two decimal places


        return render(request, 'quiz_result.html', {'quiz': quiz, 'grade': grade})
    else:
        return redirect('home')
```

Рис. 4.5 – Код подання «quiz_result»

Моделі: код визначає моделі Django за допомогою класу `models.Model`. Моделі представляють таблиці бази даних і визначають структуру даних. Моделі включають в себе:

— `Quiz`: Представляє собою тест і містить такі поля, як `name`, `short_description` та `long_description`.

— `Question`: Представляє питання в тесті і має зв'язок зовнішнього ключа з моделлю `Quiz`. Містить поле `content`.

— `Answer`: Являє собою відповідь на питання і має зв'язок зовнішнього ключа з моделлю `Question`. Містить поле `content` та логічне поле `is_correct`.

```python
from django.db import models


class Quiz(models.Model):
    name = models.CharField(max_length=100)
    short_description = models.CharField(max_length=200)
    long_description = models.TextField()

    def __str__(self):
        return self.name


class Question(models.Model):
    quiz = models.ForeignKey(Quiz, on_delete=models.CASCADE)
    content = models.TextField()

    def __str__(self):
        return self.content


class Answer(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    content = models.TextField()
    is_correct = models.BooleanField()

    def __str__(self):
        return self.content
```

Рис. 4.6 – Код моделей

Шаблони: код посилається на декілька HTML-шаблонів, які визначають структуру та вміст веб-сторінок. Шаблони включають:

— `quiz_home.html`: Відображає домашню сторінку, на якій відображаються вікторини та обробляється автентифікація користувачів.

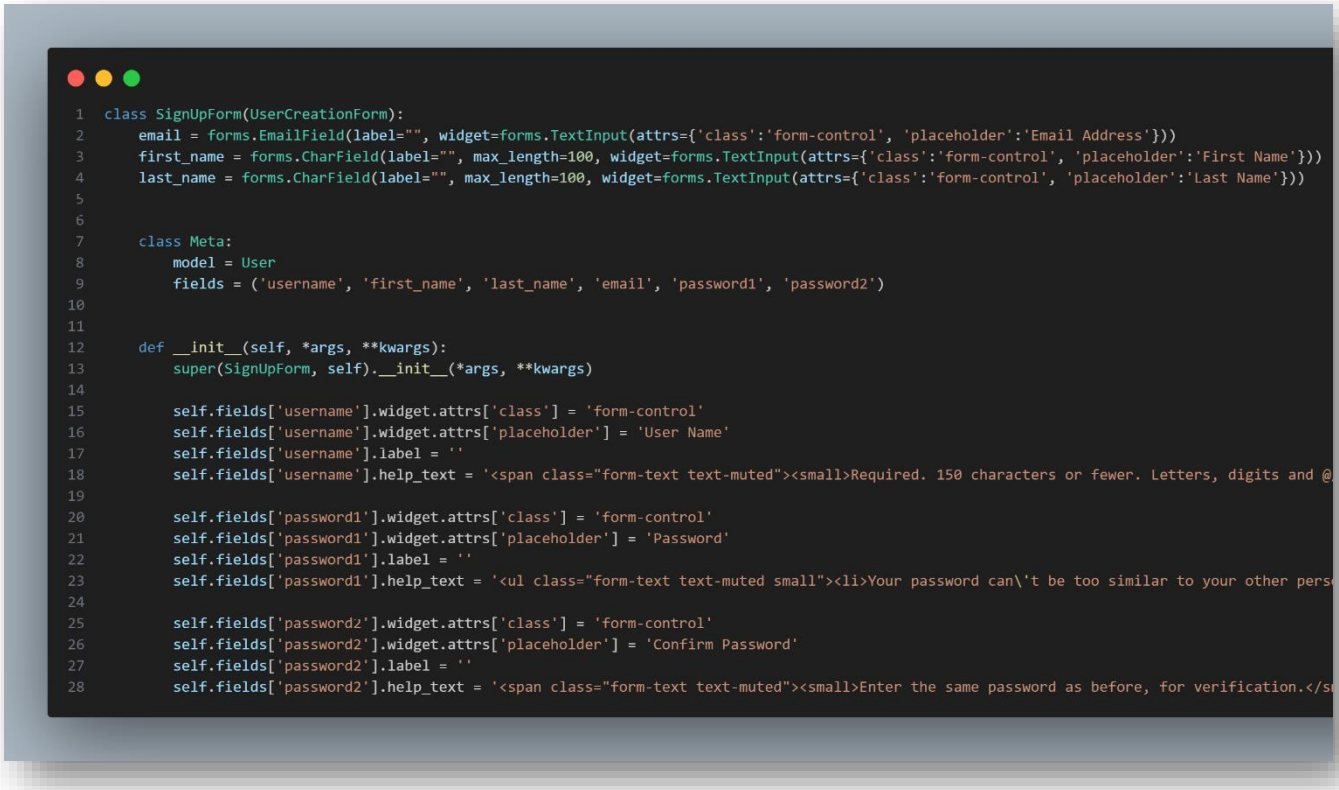
— `register.html`: Відображає форму реєстрації для нових користувачів.

— `quiz_info.html`: Відображає інформацію для конкретного тесту.

— `quiz_start.html`: Відображає питання вікторини і дозволяє користувачеві вибрати відповіді на них.

— `quiz_result.html`: Показує користувачеві результат тесту, включаючи оцінку.

Код файлу форми імпортує користувацьку реєстраційну форму `SignUpForm`, яка розширює `UserCreationForm` Django. Форма додає додаткові поля і змінює поведінку успадкованої форми за замовчуванням.



```
class SignUpForm(UserCreationForm):
    email = forms.EmailField(label="", widget=forms.TextInput(attrs={'class':'form-control', 'placeholder':'Email Address'}))
    first_name = forms.CharField(label="", max_length=100, widget=forms.TextInput(attrs={'class':'form-control', 'placeholder':'First Name'}))
    last_name = forms.CharField(label="", max_length=100, widget=forms.TextInput(attrs={'class':'form-control', 'placeholder':'Last Name'}))


    class Meta:
        model = User
        fields = ('username', 'first_name', 'last_name', 'email', 'password1', 'password2')


    def __init__(self, *args, **kwargs):
        super(SignUpForm, self).__init__(*args, **kwargs)

        self.fields['username'].widget.attrs['class'] = 'form-control'
        self.fields['username'].widget.attrs['placeholder'] = 'User Name'
        self.fields['username'].label = ''
        self.fields['username'].help_text = '<span class="form-text text-muted"><small>Required. 150 characters or fewer. Letters, digits and @

        self.fields['password1'].widget.attrs['class'] = 'form-control'
        self.fields['password1'].widget.attrs['placeholder'] = 'Password'
        self.fields['password1'].label = ''
        self.fields['password1'].help_text = '<ul class="form-text text-muted small"><li>Your password can\'t be too similar to your other pers

        self.fields['password2'].widget.attrs['class'] = 'form-control'
        self.fields['password2'].widget.attrs['placeholder'] = 'Confirm Password'
        self.fields['password2'].label = ''
        self.fields['password2'].help_text = '<span class="form-text text-muted"><small>Enter the same password as before, for verification.</s
```

Рис. 4.7 – Код реєстраційної форми

Форма містить наступні поля:

— `email`: Поле `EmailField` для отримання адреси електронної пошти користувача.

— `first_name`: Поле `CharField` для введення імені користувача.

— `last_name`: Поле `CharField` для отримання прізвища користувача.

— `username`: Поле `username` за замовчуванням, успадковане від `UserCreationForm`.

— `password1`: Поле за замовчуванням `password1`, успадковане від `UserCreationForm`.

— `password2`: Поле за замовчуванням `password2`, успадковане від `UserCreationForm`.

Форма налаштовує зовнішній вигляд полів форми шляхом додавання класів CSS та заповнювачів. Вона використовує віджет `TextInput` для відображення полів як елементів введення тексту.

Клас `Meta` форми визначає модель для використання (`User`) та поля для включення до форми (`username`, `first_name`, `last_name`, `email`, `password1`, `password2`).

Загалом, програмний код реалізує веб-додаток, який дозволяє користувачам реєструватися, входити в систему, переглядати список тестів, проходити тести і бачити результати. Він використовує моделі, представлення, шаблони та форми Django для обробки логіки додатку та представлення даних користувачам.

Панель адміністратора дозволяє отримувати доступ до управління даними бази даних та додавати новий контент.

5 РЕАЛІЗАЦІЯ СИСТЕМИ АВТОМАТИЗАЦІЇ ДЛЯ ДОДАТКУ

5.1 Інфраструктурне рішення AWS

Задля реалізації автоматизації розгортання додатку в хмарному середовищі, потрібно визначити необхідні ресурси.

Бажаним кінцевим результатом є додаток, що працює в кластері Docker Swarm, імплементація якого потребує, як мінімум, 3 сервери (віртуальні машини): 1 для ноди-менеджера та 2 для робочих нод. Відповідним ресурсом AWS є віртуальні машини EC2.

Задля уникнення ускладнення архітектури балансування трафіку всередині Docker Swarm та задля забезпечення розміщення віртуальних серверів виключно в приватній мережі, потрібно використати розподілювач навантаження. Такий підхід підвищує загальну безпеку системи та надає додаткові можливості по обмеженню доступу до серверів додатку. Відповідним ресурсом AWS є ELB, типу «application». ELB типу «application» повинен розміщуватися в двох публічних мережах, що перебувають в різних зонах доступності AWS, так як є сервісом високої доступності.

Імплементація розподілювача навантаження створює додатковий шар складності. Для віддаленого доступу до серверів зазвичай використовують 22 TCP порт для підключення по протоколу SSH (також його буде використовувати Ansible), саме тому необхідно спроєктувати відповідну модель доступу. Розподілювач навантаження не повинен бути включений в цей концепт, бо є відповідальним лише за обслуговування клієнтського трафіку. У таких випадках можливе використання VPN або бастіон хоста, який буде слугувати «хопом» для віддалених підключень до серверів додатку, що перебувають у приватній мережі. У даній роботі буде розглянуто саме варіант імплементації бастіону, роль якого буде відігравати іще один сервер EC2.

Бастіон хост повинен перебувати у публічній мережі задля забезпечення зовнішнього доступу до нього. Також необхідним є підключення бастіону до мережі, а тому необхідно додати ресурс AWS Internet Gateway.

Для забезпечення нормального функціонування серверів додатку, необхідно забезпечити їх доступ до ресурсів за межами VPC та відповідно до мережі. Типовою реалізацією для цієї ситуації є сервіс AWS NAT Gateway, що у парі із EIP (Elastic IP) надасть доступ до мережі Інтернет для серверів приватної мережі.

Гарною практикою є створення окремих політик Security Groups для кожного типу ресурсів, що того потребують. Таким чином можна відокремити 3 групи: група розподілювача навантаження, група серверів додатку та група бастіону.

Для забезпечення роботи розподілювача навантаження достатньо впровадити дозвіл перенаправлення трафіку із порту 80 tcp на той самий порт його цільової групи, тобто, робочих нод майбутнього кластеру Docker Swarm, так як нода-менеджер не призначена для прийому трафіку, хоча і може виконувати цю роль.

Для забезпечення роботи серверів додатку достатньо дозволити внутрішній трафік мережі і дозволити приймати підключення від бастіону (22 tcp) та від розподілювача навантаження (80 tcp).

Таким чином, сформоване уявлення про майбутню інфраструктуру, яку можна візуалізувати за допомогою веб-сервісу для побудови моделей інфраструктури у хмарних середовищах CloudCraft.

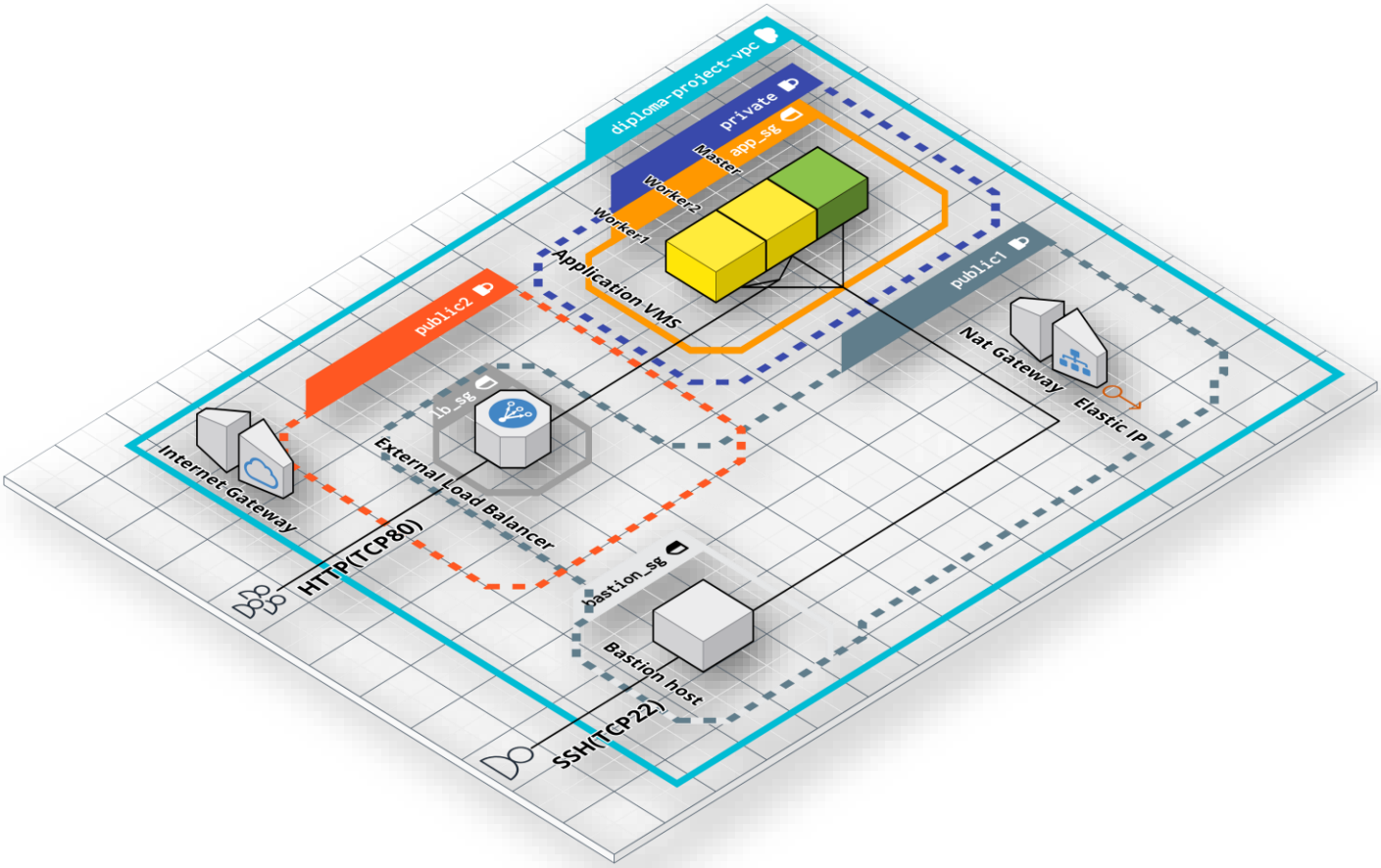


Рис. 5.1 – Модель майбутньої інфраструктури в хмарному середовищі Amazon

5.2 Опис інфраструктури за допомогою коду Terraform

Для створення необхідної інфраструктури за допомогою Terraform, необхідно описати ресурси за допомогою мови hcl та синтаксису відповідного провайдера.

Задля зручності написання коду Terraform, зазвичай створюються одразу декілька конфігураційних файлів, які називають відповідно до їх вмісту.

```
yakimoro@wks4:~/projects/terraform-aws-diploma$ tree -P "*.tf" -I "ansible"
.
├── app_instances.tf
├── app_sgs.tf
├── generation.tf
├── networking.tf
├── outputs.tf
├── providers.tf
├── service_instances.tf
├── service_sgs.tf
├── tf_templates
└── variables.tf
```

Рис. 5.2 – Обрані найменування для файлів конфігурацій

Обов’язковим є секція провайдерів, що часто знаходиться у файлі «providers.tf», у якій описані необхідні провайдери, їх версії та конфігурація Terraform Backend.

Terraform Backend є сховищем, яке використовує Terraform для збереження свого файлу стану. Важливим є те, що при використанні локальної розробки Terraform, у розробника є можливість зберігати файл стану локально у файловій системі, в той час як для автоматизації Terraform, необхідним є міграція стану до Remote Backend. Remote Backend може бути представленим різними ресурсами для зберігання інформації, але так як у даній роботі використовується саме AWS, доцільно буде використати AWS S3 сховище для збереження та версіонування файлу стану Terraform. Для цього було створено відповідне сховище S3.

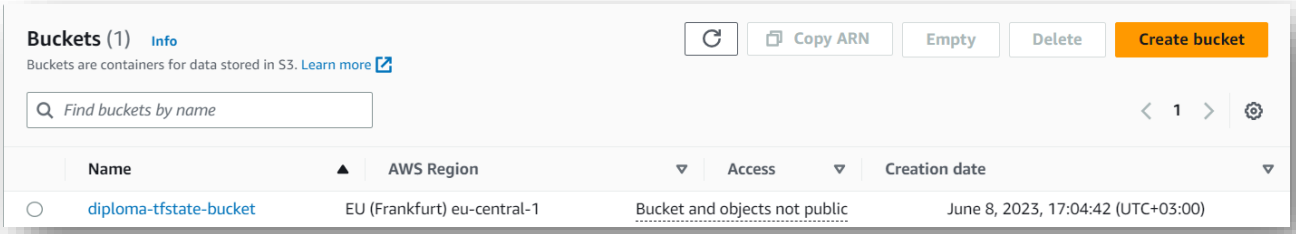


Рис. 5.3 – Список створених S3 сховищ у консолі AWS

Наступним кроком для міграції є додавання сховища до конфігурації Terraform Backend в секції провайдерів.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
  backend "s3" {
    bucket = "diploma-tfstate-bucket"
    key    = "terraform.tfstate"
    region = "eu-central-1"
  }
}

provider "aws" {

}
```

Рис. 5.4 – Конфігурація провайдерів та Terraform backend

Для застосування змін необхідно виконати команду «terraform init», що запускає процес пошуку потрібних провайдерів локально, завантажуючи їх при потребі та перевіряє налаштування Terraform Backend.

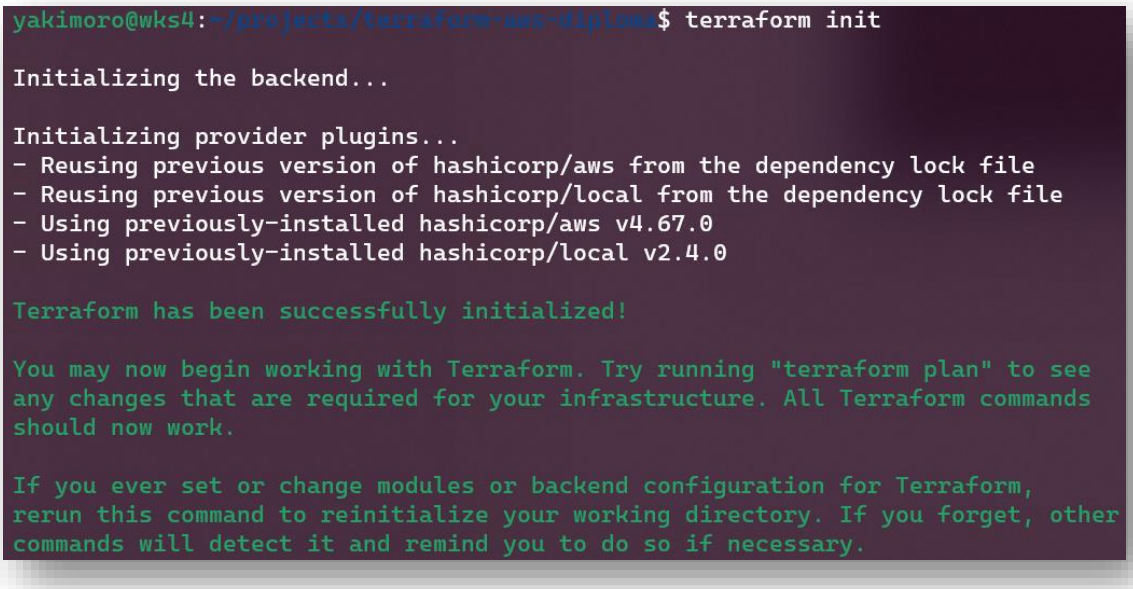
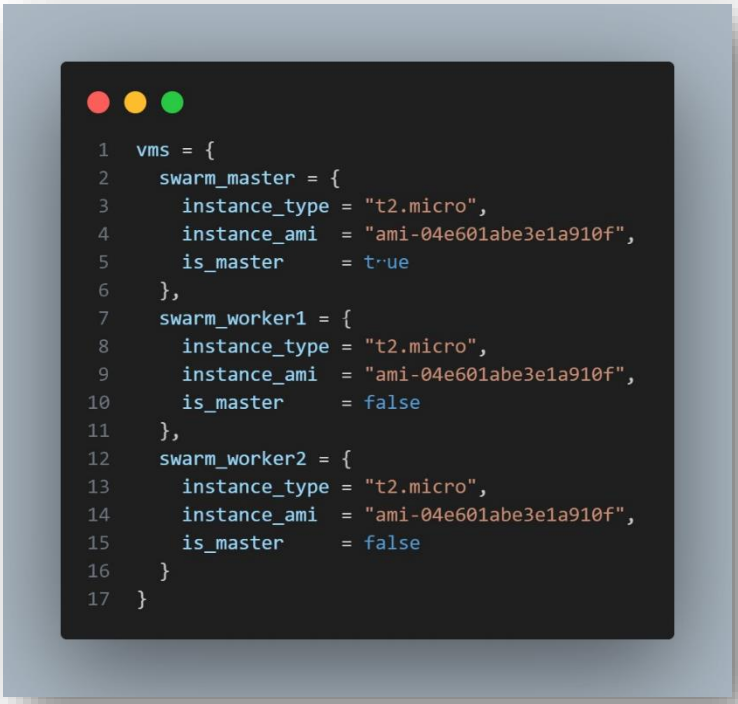


Рис. 5.5 – результат виконання «terraform init»

Важливим функціоналом Terraform є використання змінних, що спрощує розробку та надає можливість до скорочення об’ємів написаного коду. Стандартним іменем файлу для декларування очікуваних змінних на вхід є «variables.tf», в той час як файлом задання значень для змінних є «terraform.tfvars». Значення змінних може бути отримано із змінних середовища, окремих .env файлів тощо.

Terraform підтримує складні змінні різних типів, що надає можливості гнучкого подання конфігурацій. До функціоналу Terraform також можна віднести цикли, логічні вирази, мета-аргументи (depends_on, count, for_each), шаблонізатор для створення файлів з використанням шаблонів, різноманітні функції.

Використовуючи складні змінні для декларування значень конфігурації та мета-аргумент for_each, можна значно скоротити об’єм коду через відсутність потреби переписувати блоки коду для ресурсів одного типу.



```
vms = {
  swarm_master = {
    instance_type = "t2.micro",
    instance_ami  = "ami-04e601abe3e1a910f",
    is_master     = true
  },
  swarm_worker1 = {
    instance_type = "t2.micro",
    instance_ami  = "ami-04e601abe3e1a910f",
    is_master     = false
  },
  swarm_worker2 = {
    instance_type = "t2.micro",
    instance_ami  = "ami-04e601abe3e1a910f",
    is_master     = false
  }
}
```

Рис. 5.6 – використання змінної типу «map of object»

```
resource "aws_key_pair" "app_instance_key" {
  for_each   = var.vms
  key_name   = "${var.project_prefix}-${each.key}-app_instance_key"
  public_key = file("${var.ssh_keys_location}${var.pub_key_name}")
}

resource "aws_instance" "app_instance" {
  for_each               = var.vms
  ami                    = each.value.instance_ami
  instance_type          = each.value.instance_type
  key_name               = aws_key_pair.app_instance_key[each.key].key_name
  subnet_id              = aws_subnet.private_subnet.id
  vpc_security_group_ids = [aws_security_group.application_vms_sg.id]

  tags = {
    Name      = "${var.project_prefix}-${each.key}"
    is_master = each.value.is_master ? "true" : "false"
  }
}
```

Рис. 5.7 – використання мета-аргументу «for_each»

Після закінчення створення бажаної конфігурації, можна скористатися вбудованою командою «terraform fmt» для виправлення форматування та приведення коду до нормального стану.

Для валідації коду використовується вбудована команда «terraform validate», що перевіряє код на синтаксичні помилки.

Для створення плану для застосування змін, використовується команда «terraform plan», яка покаже, що саме у разі застосування «terraform apply» зміниться.

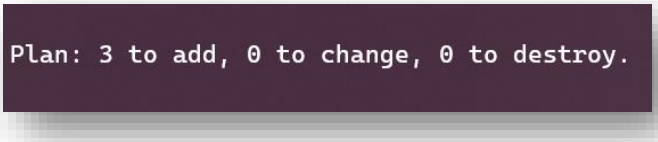


Рис. 5.8 – секція змін в результаті застосування команди «terraform plan»

У результаті успішного застосування «terraform apply», отримуємо розгорнуту за допомогою однієї команди інфраструктуру, яка може бути так само просто знищена і знову відтворена.

Для зручності, було реалізовано Terraform outputs, які передають в термінал значення, які зазначено у відповідних ресурсах «output»

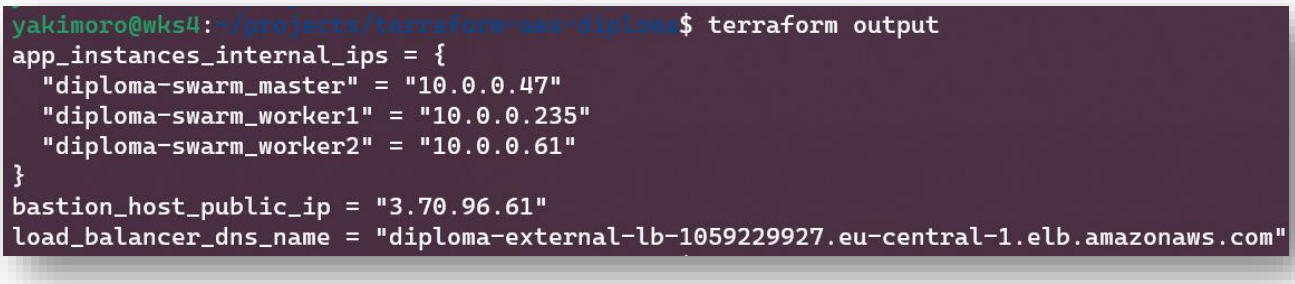


Рис. 5.9 – Terraform output

Для визначення залежностей ресурсів, Terraform використовує графи залежностей. Модель залежності компонентів інфраструктури для поставленої задачі можна візуалізувати за допомогою Digraph Visualiser, вставивши у поле коду результат команди «terraform graph».

5.3 Контейнеризаця додатку та створення моделі взаємодії в Docker

Для контейнеризації додатку, необхідно ознайомитись із кроками для контейнеризації додатків Django і створити Dockerfile.

Найкращими практиками є:

— відмова від використання важких базових образів на користь легких;

— зведення кількості кроків у файлі до мінімуму;

— розташування незмінних кроків вгорі, залишаючи непостійні внизу задля оптимізації кешованих етапів і прискорення збірки.

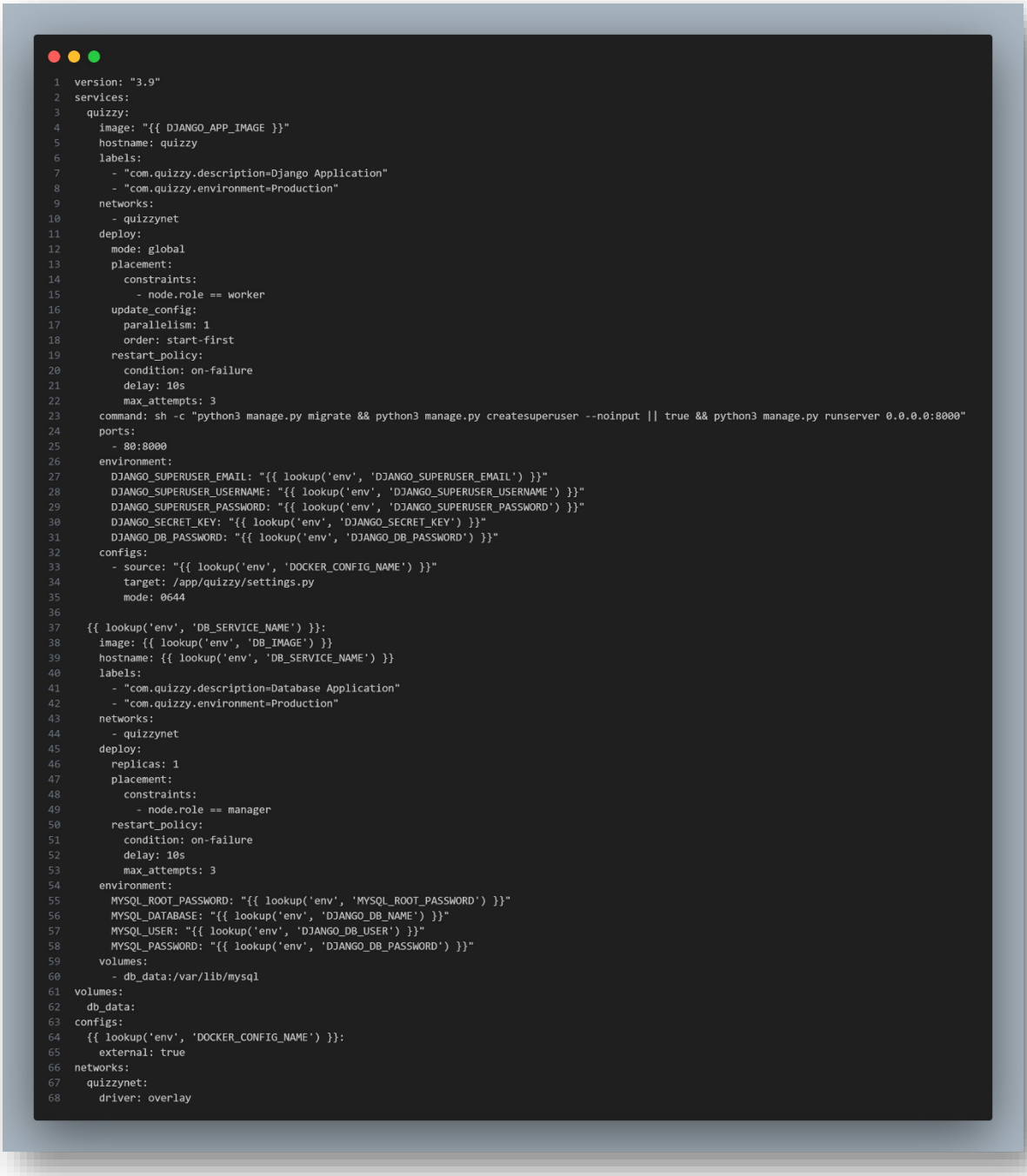
Базовим образом для додатка на Django є образ python. Dockerfile вміщує в себе крок встановлення необхідних бібліотек для повноцінного функціонування; встановлення необхідних залежностей із файлу «requirements.txt»; після чого локальна директорія із додатком копіюється всередину файлової системи майбутнього контейнера; відкривається порт, на якому працює додаток та встановлюється команда, що буде виконана при запуску контейнера за допомогою образа.

```
yakimoro@wks4:~/projects/django-app-diploma$ cat Dockerfile
FROM python:3.9-alpine

RUN apk update && apk add --no-cache mariadb-dev build-base

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["python3", "manage.py", "runserver", "0.0.0.0:8000"]
```

Рис. 5.10 – Зміст Dockerfile

Для взаємодії додатка та бази даних, при використанні оркестратора Docker Swarm, необхідно створити відповідний конфігураційний файл, що буде визначати модель взаємодії сервісів та опції їх запуску у середовищі контейнерів.

Одразу слід зазначити, що так як файл є конфігурацією, він не буде перебувати у форматі готового файлу «docker-compose.yml» у репозиторіях Git, так як буде потребувати генерації значень.



```
version: "3.9"
services:
  quizzy:
    image: "{{ DJANGO_APP_IMAGE }}"
    hostname: quizzy
    labels:
      - "com.quizzy.description=Django Application"
      - "com.quizzy.environment=Production"
    networks:
      - quizzynet
    deploy:
      mode: global
      placement:
        constraints:
          - node.role == worker
      update_config:
        parallelism: 1
        order: start-first
      restart_policy:
        condition: on-failure
        delay: 10s
        max_attempts: 3
    command: sh -c "python3 manage.py migrate && python3 manage.py createsuperuser --noinput || true && python3 manage.py runserver 0.0.0.0:8000"
    ports:
      - 80:8000
    environment:
      DJANGO_SUPERUSER_EMAIL: "{{ lookup('env', 'DJANGO_SUPERUSER_EMAIL') }}"
      DJANGO_SUPERUSER_USERNAME: "{{ lookup('env', 'DJANGO_SUPERUSER_USERNAME') }}"
      DJANGO_SUPERUSER_PASSWORD: "{{ lookup('env', 'DJANGO_SUPERUSER_PASSWORD') }}"
      DJANGO_SECRET_KEY: "{{ lookup('env', 'DJANGO_SECRET_KEY') }}"
      DJANGO_DB_PASSWORD: "{{ lookup('env', 'DJANGO_DB_PASSWORD') }}"
    configs:
      - source: "{{ lookup('env', 'DOCKER_CONFIG_NAME') }}"
        target: /app/quizzy/settings.py
        mode: 0644

  {{ lookup('env', 'DB_SERVICE_NAME') }}:
    image: {{ lookup('env', 'DB_IMAGE') }}
    hostname: {{ lookup('env', 'DB_SERVICE_NAME') }}
    labels:
      - "com.quizzy.description=Database Application"
      - "com.quizzy.environment=Production"
    networks:
      - quizzynet
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager
      restart_policy:
        condition: on-failure
        delay: 10s
        max_attempts: 3
    environment:
      MYSQL_ROOT_PASSWORD: "{{ lookup('env', 'MYSQL_ROOT_PASSWORD') }}"
      MYSQL_DATABASE: "{{ lookup('env', 'DJANGO_DB_NAME') }}"
      MYSQL_USER: "{{ lookup('env', 'DJANGO_DB_USER') }}"
      MYSQL_PASSWORD: "{{ lookup('env', 'DJANGO_DB_PASSWORD') }}"
    volumes:
      - db_data:/var/lib/mysql
volumes:
  db_data:
configs:
  {{ lookup('env', 'DOCKER_CONFIG_NAME') }}:
    external: true
networks:
  quizzynet:
    driver: overlay
```

Рис. 5.11 – Шаблон для майбутнього «docker-compose.yml»

Docker Compose описує мультисервісне налаштування для додатку Django під назвою "quizzy" та відповідного сервісу бази даних.

Основною перевагою використання docker-compose файлів в режимі Swarm є можливість використання функціоналу, доступного лише для режиму Swarm: налаштування режиму реплікаці, налаштування стратегії оновлення, використання конфігурацій Docker.

Ідея відділення конфігураційного файлу settings.py для додатка є ключовою причиною імплементації монтування розділу з відповідним файлом за допомогою секції «config».

Особливу увагу слід приділити визначенню команди для запуску контейнера додатку, де зазначено декілька команд:

— Для застосування міграцій;

— Для створення адміністратора;

— Для запуску сервера додатку.

Так як Docker Swarm частково використовує функціонал Docker compose, сервіси мають змогу взаємодіяти без використання адрес overlay мережі, за допомогою імен сервісів, задекларованих в файлі-конфігурації

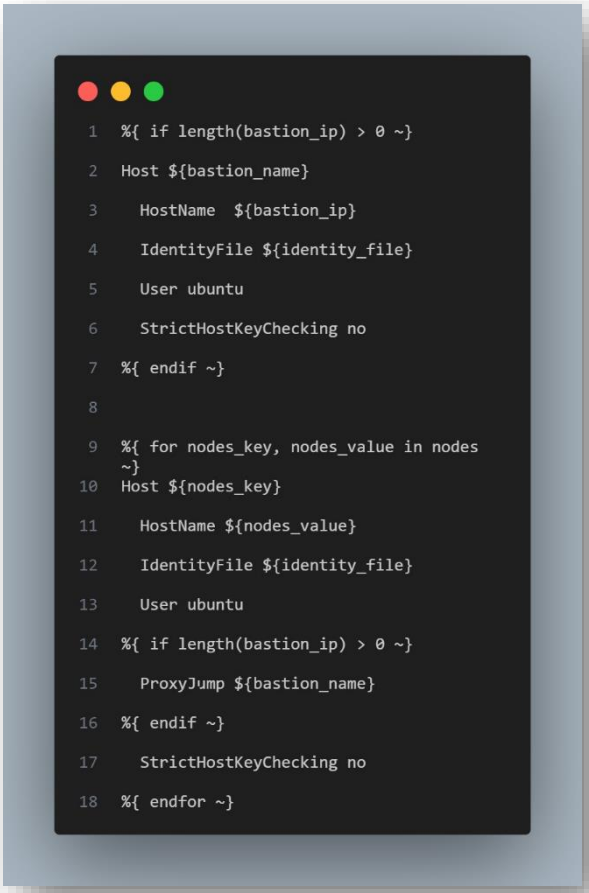
5.4 Підготовка платформи за допомогою Ansible

Підготовка платформи у контексті даної роботи являє собою створення та конфігурування Docker Swarm кластеру за допомогою інструменту Ansible.

Основне питання, що є першочерговим під час розгляду можливих реалізацій є побудова правильної взаємодії двох інструментів: Terraform та Ansible.

Як уже було зазначено, є можливість створення інвентарного файлу для Ansible за допомогою Terraform даних. Це допомагає не залежати від потреби вручну створювати необхідні файли, так як оновлення файлу інвентарю буде відбуватися автоматично при зміні даних Terraform шляхом генерації. У випадку із рішенням про використання бастіону як проміжного хоста, з'являється потреба у використанні додаткової конфігурації SSH підключення для proxy-стрибків. Для цього було створено відповідні шаблони для Terraform: «ansible_inventory.tpl», «ssh_proxy_conf.tpl», які будуть регенеруватись та заповнюватись даними при кожному застосуванні змін Terraform.

Рис. 5.12 – Шаблон конфігурації цільових хостів для Ansible



```
%{ if length(bastion_ip) > 0 ~}
Host ${bastion_name}
  HostName  ${bastion_ip}
  IdentityFile ${identity_file}
  User ubuntu
  StrictHostKeyChecking no
%{ endif ~}


%{ for nodes_key, nodes_value in nodes ~}
Host ${nodes_key}
  HostName ${nodes_value}
  IdentityFile ${identity_file}
  User ubuntu
%{ if length(bastion_ip) > 0 ~}
  ProxyJump ${bastion_name}
%{ endif ~}
  StrictHostKeyChecking no
%{ endfor ~}
```

Рис. 5.13 – Шаблон конфігурації SSH для доступу до цільових хостів

Для ініціювання Swarm, необхідно забезпечити встановлення Docker та додаткових пакетів на цільові ноди.

Було створено відповідні задачі Ansible із використанням структури директорій Ansible-galaxy.

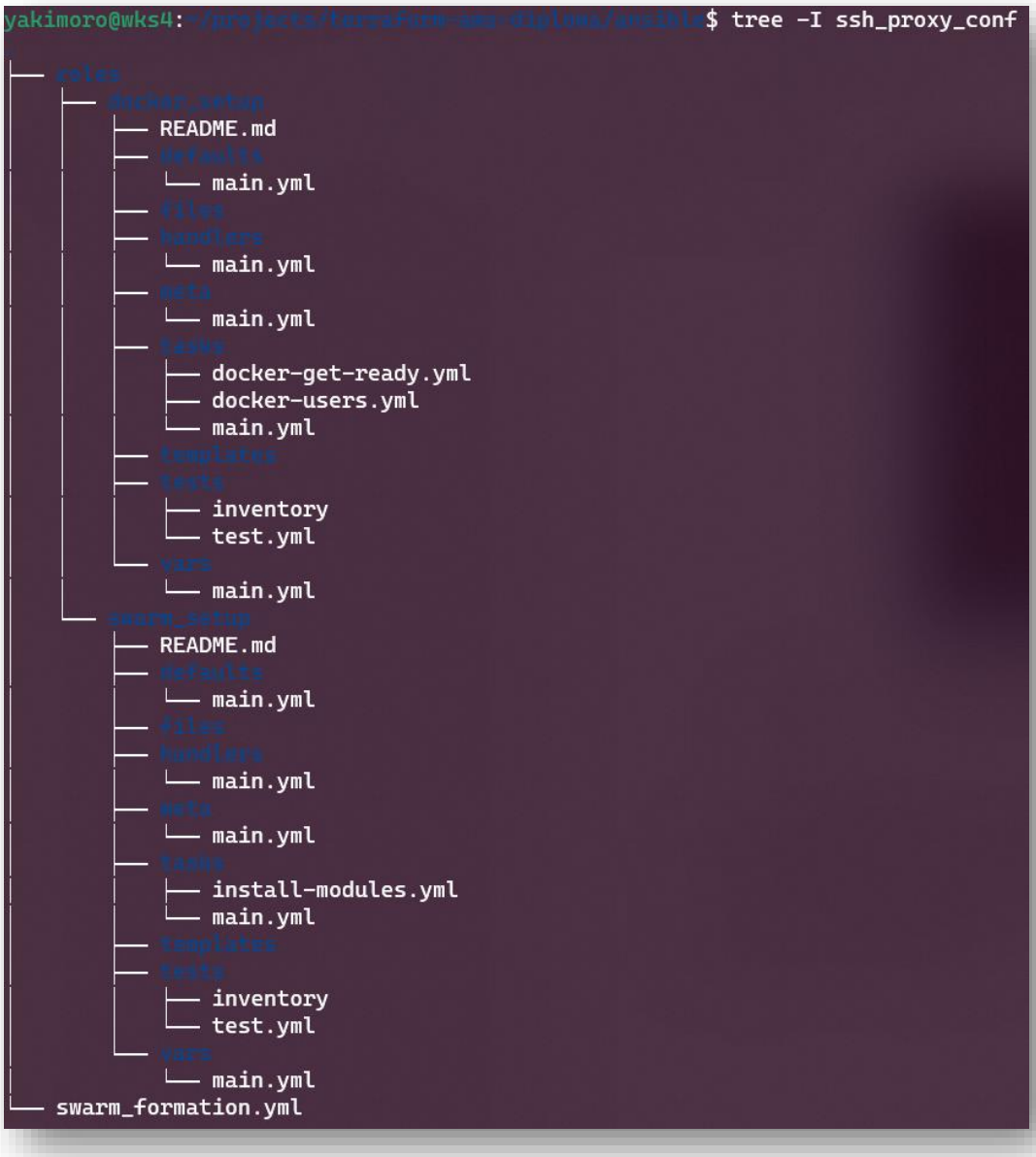
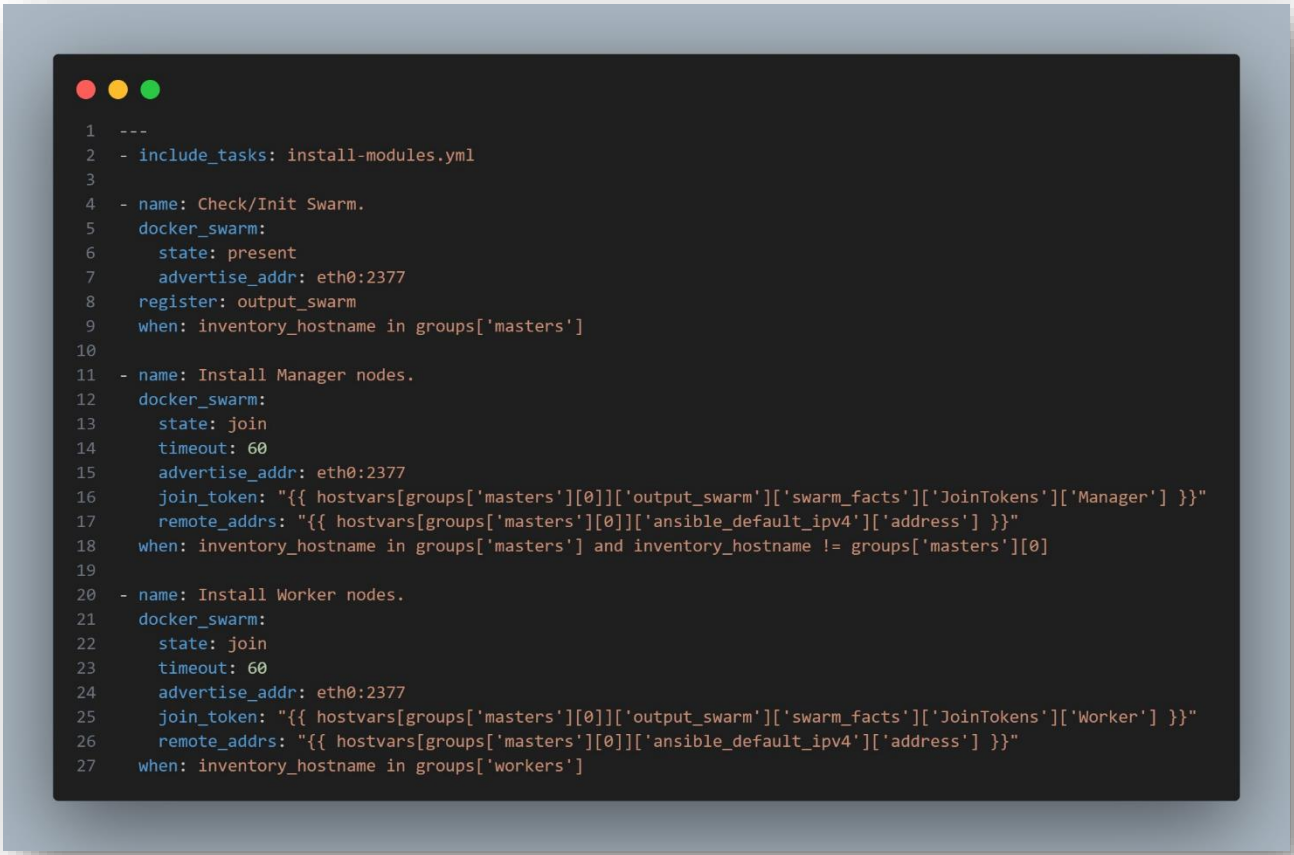


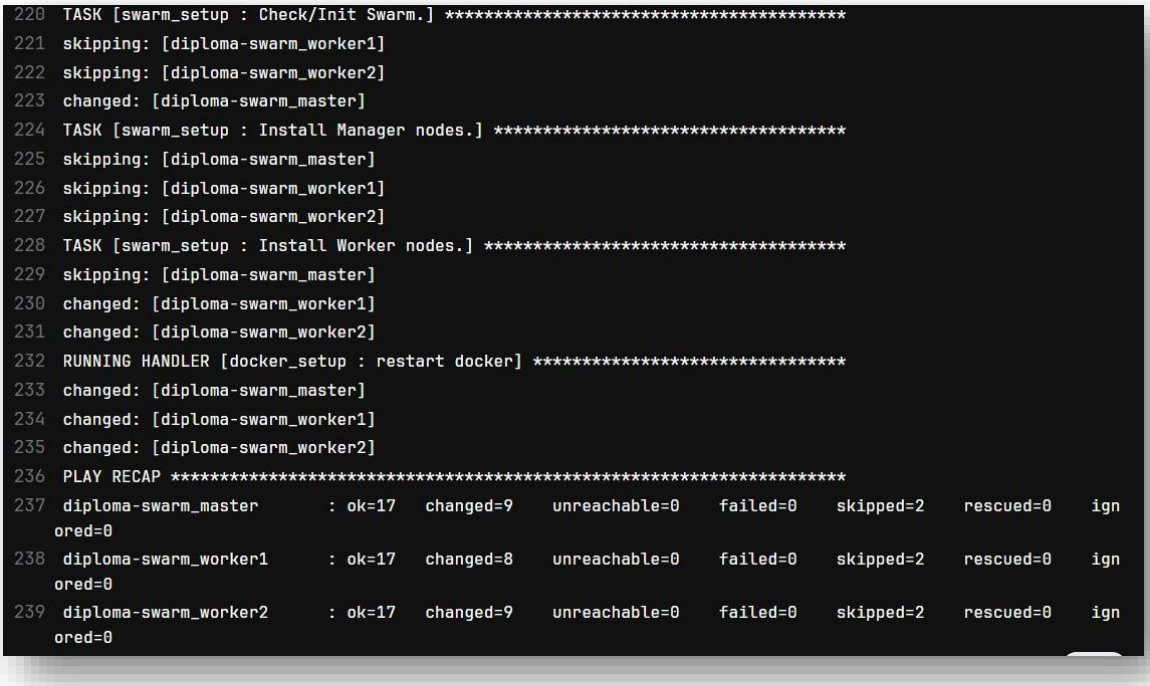
Рис. 5.14 – Структура директорій Ansible-galaxy для створення Swarm

Сценарій «swarm_formation.yml» викликає відповідні ролі, які підключаються до цільових хостів та виконують задачі по встановленню Docker, перевірці наявності потрібних пакетів та ініціюванню Docker Swarm.



```yaml
---
- include_tasks: install-modules.yml

- name: Check/Init Swarm.
  docker_swarm:
    state: present
    advertise_addr: eth0:2377
  register: output_swarm
  when: inventory_hostname in groups['masters']

- name: Install Manager nodes.
  docker_swarm:
    state: join
    timeout: 60
    advertise_addr: eth0:2377
    join_token: "{{ hostvars[groups['masters'][0]]['output_swarm']['swarm_facts']['JoinTokens']['Manager'] }}"
    remote_addrs: "{{ hostvars[groups['masters'][0]]['ansible_default_ipv4']['address'] }}"
  when: inventory_hostname in groups['masters'] and inventory_hostname != groups['masters'][0]

- name: Install Worker nodes.
  docker_swarm:
    state: join
    timeout: 60
    advertise_addr: eth0:2377
    join_token: "{{ hostvars[groups['masters'][0]]['output_swarm']['swarm_facts']['JoinTokens']['Worker'] }}"
    remote_addrs: "{{ hostvars[groups['masters'][0]]['ansible_default_ipv4']['address'] }}"
  when: inventory_hostname in groups['workers']
```

Рис. 5.14 – Ansible код для ініціювання Swarm та приєднання нод до кластеру



```
TASK [swarm_setup : Check/Init Swarm.] ****************************************
skipping: [diploma-swarm_worker1]
skipping: [diploma-swarm_worker2]
changed: [diploma-swarm_master]
TASK [swarm_setup : Install Manager nodes.] ***********************************
skipping: [diploma-swarm_master]
skipping: [diploma-swarm_worker1]
skipping: [diploma-swarm_worker2]
TASK [swarm_setup : Install Worker nodes.] ************************************
skipping: [diploma-swarm_master]
changed: [diploma-swarm_worker1]
changed: [diploma-swarm_worker2]
RUNNING HANDLER [docker_setup : restart docker] *******************************
changed: [diploma-swarm_master]
changed: [diploma-swarm_worker1]
changed: [diploma-swarm_worker2]
PLAY RECAP ********************************************************************
diploma-swarm_master       : ok=17   changed=9    unreachable=0    failed=0    skipped=2    rescued=0    ignored=0
diploma-swarm_worker1      : ok=17   changed=8    unreachable=0    failed=0    skipped=2    rescued=0    ignored=0
diploma-swarm_worker2      : ok=17   changed=9    unreachable=0    failed=0    skipped=2    rescued=0    ignored=0
```

Рис. 5.15 – Результат застосування Ansible

Після успішного відпрацювання Ansible, можна підключитися до менеджер ноди кластеру та перевірити результат для наочності.

```
ubuntu@ip-10-0-0-47:~$ docker node ls
ID                            HOSTNAME        STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
s16h3fdofm5qhuip0fzm79tdj *   ip-10-0-0-47    Ready     Active         Leader           24.0.2
jfax48f38faw8p5c7vkuh51pf     ip-10-0-0-61    Ready     Active                          24.0.2
jv1x45jetcs2w9w65upp07baa     ip-10-0-0-235   Ready     Active                          24.0.2
```

Рис. 5.16 – Коректна робота команд Docker Swarm на віддаленому хості

5.5 Проєктування автоматизації за допомогою GitLab CI/CD

Ключовою частиною проєктування системи автоматизованого розгортання та оновлення додатку є поєднання декількох етапів автоматизації за допомогою інструментів CI/CD.

Для реалізації було вирішено поділити загальний автоматичний процес на етапи:

— Етап створення та підготовки інфраструктури(Infra);

— Етап створення артефакту додатку(CI);

— Етап розгортання та оновлення додатку(CD);

Відповідний функціонал платформи GitLab надає можливість створити репозиторії для збереження коду для кожного із етапів.

Було створено групу проєкту та 3 репозиторії, що належать до неї:

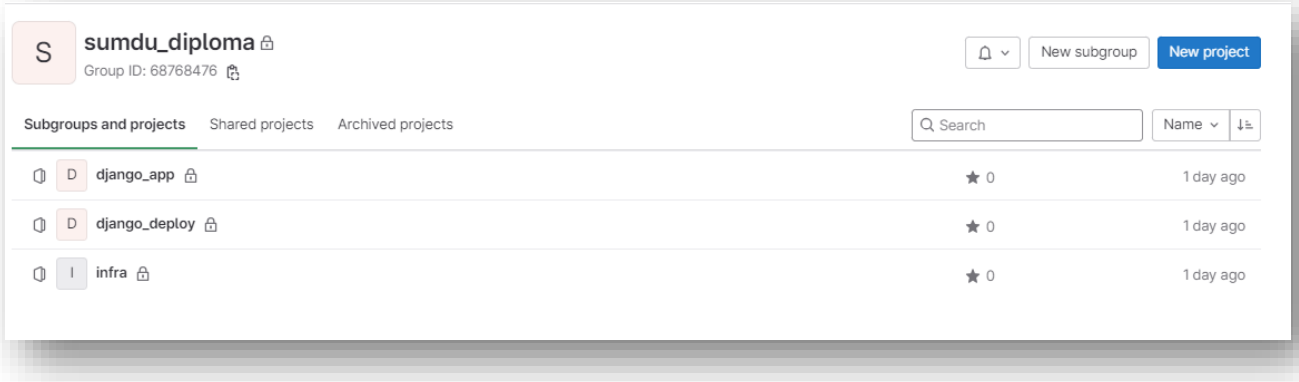


Рис. 5.17 – Створені проєктні репозиторії в Gitlab

Окреме збереження коду та конфігурації реалізоване за допомогою відокремлення репозиторіїв «django_app» та «django_deploy». В той час як інфраструктурний репозиторій «infra» відповідає за зберігання коду Terraform та Ansible для підготовки платформи.

Для виконання задач конвеєрів GitLab CI/CD було вирішено використовувати власний агент GitLab. Для цього було зареєстровано, сконфігуровано та підключено до групи проєктів агент типу «docker» із цільовим

тегом «core».

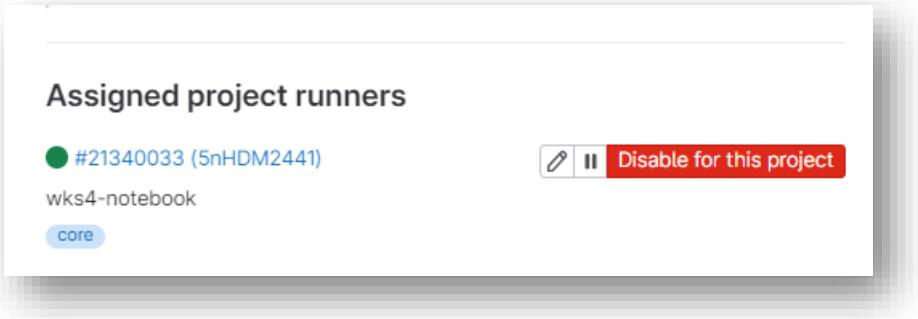


Рис. 5.18 – Верифікований GitLab агент

```
root@wks4:/home/yakimoro/projects/django-app-diploma# gitlab-runner health-check
Runtime platform                                    arch=amd64 os=linux pid=165837 revision=d540b510 version=15.9.1
```

Рис. 5.19 – Підтвердження функціонування агента GitLab на хості

Процес розгортання додатку планується реалізувати за допомогою Ansible, який так само потребує файлів конфігурації для роботи, тому необхідно забезпечити обмін артефактами поміж репозиторіїв для створення алгоритму передачі результатів роботи до репозиторію розгортання додатку. Для процесу автоматизації було побудовано таку схему конвеєрів:

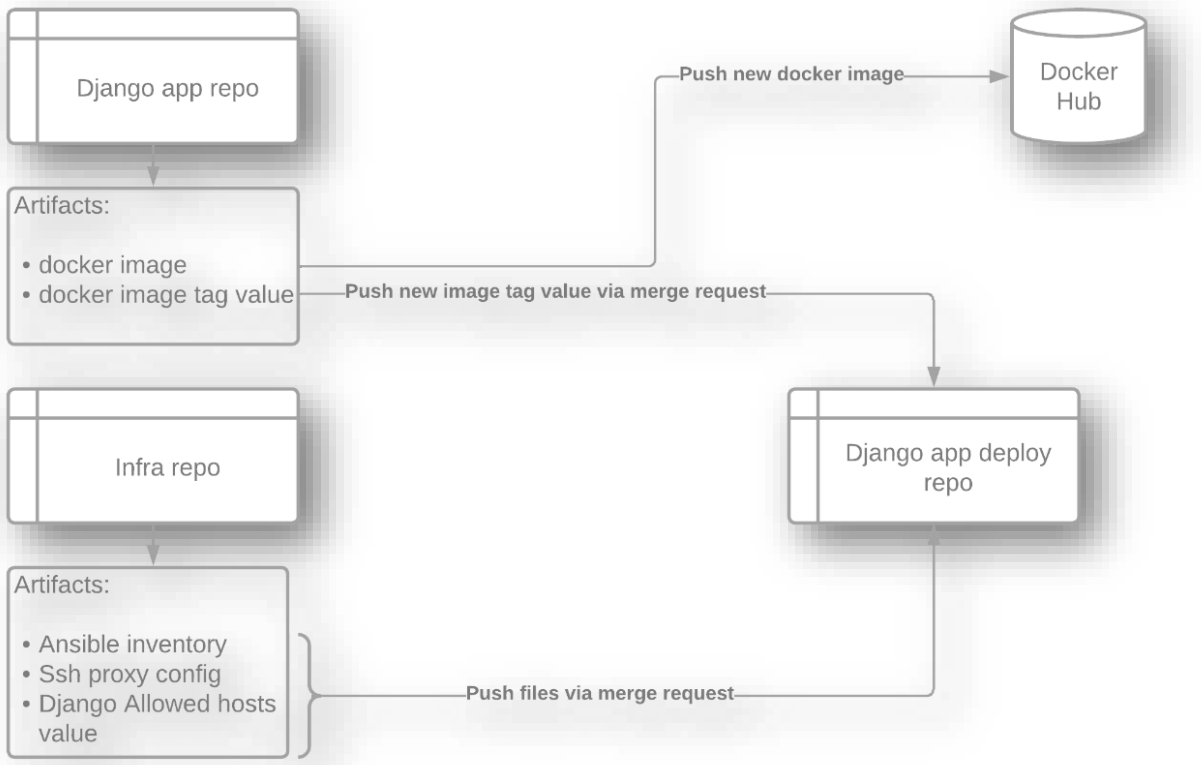


Рис. 5.20 – Схема розповсюдження артефактів поміж репозиторіями

Наданий варіант є реалізацією на основі функціоналу Git та GitLab, так як потребує використання GitLab API для створення запитів на злиття гілок.

5.5.1 Конвеєр розгортання інфраструктури

Для автоматизації роботи Terraform та Ansible в контейнеризованому середовищі роботи Gitlab агентів, необхідно подбати про декілька речей:

— Декларація змінних оточення;

— Реалізація конвеєру та запиту на злиття.

Для додавання змінних оточення достатньо додати змінну до Gitlab, зазначивши її ім'я, значення та вказавши тип змінної. Після цього, оточення, у якому відбувається запуск конвеєра CI/CD буде мати доступ до її використання. Gitlab надає можливість визначати унікальні змінні для репозиторію та змінні на рівні проєкту.

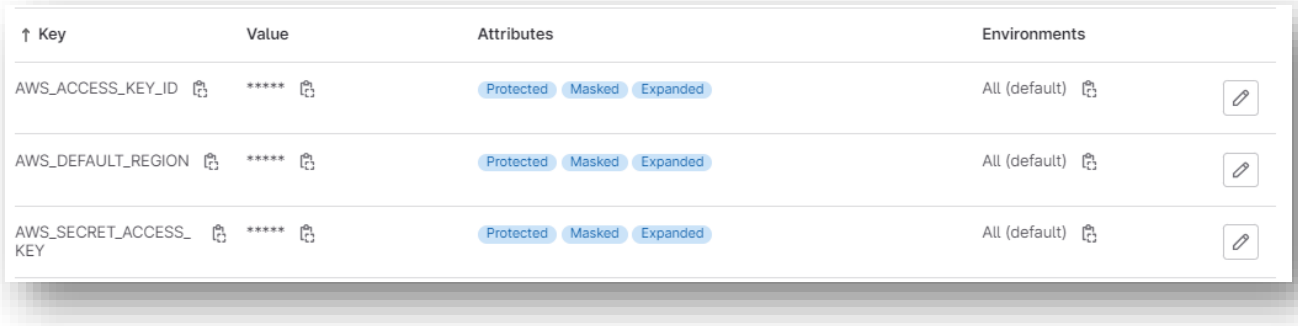


Рис. 5.21 – Змінні GitLab

До етапів у файлі «.gitlab-ci.yml» було додано:

— Етап валідації коду Terraform;

— Етап планування Tearraform;

— Етап застосування змін Terraform, де реєструються відповідні артефакти;

— Етап створення запиту на злиття (потребує створеного Access Token);

— Етап виконання Ansible для створення кластеру Swarm.

Важливо зазначити, що Terraform генерує значення ALLOWED_HOSTS для додатку Django, щоб правильно сконфігурувати знайомі хости.

```yaml
mr_job:
  stage: apply
  needs: ["apply"]
  image:
    name: registry.gitlab.com/sumdu_diploma/infra:curl-git
    pull_policy: if-not-present
    entrypoint: ['']
  tags:
    - core
  before_script:
    - current_datetime=$(date +%Y-%m-%d_%H-%M)
    - git config --global user.name "MR Robot"
    - git config --global user.email "mr.robot@robots.com"
  script:
    - |
     git clone
     https://${MR_ROBOT_ACCESS_TOKEN_NAME}:${MR_ROBOT_ACCESS_TOKEN}@gitlab.com/sumdu_diploma/django_deploy.git
    - cd django_deploy
    - git checkout -b mr-robot-branch-${CI_COMMIT_SHORT_SHA}-${current_datetime}
    - cp ${CI_PROJECT_DIR}/ansible/${ANSIBLE_INVENTORY_FILENAME} ${ANSIBLE_INVENTORY_FILENAME}
    - cp ${CI_PROJECT_DIR}/ansible/${SSH_CONF_FILENAME} ${SSH_CONF_FILENAME}
    - cp ${CI_PROJECT_DIR}/${DJANGO_CONFIG_FILENAME} ./roles/update_app/vars/${DJANGO_CONFIG_FILENAME}
    - git add ./roles/update_app/vars/${DJANGO_CONFIG_FILENAME} ${ANSIBLE_INVENTORY_FILENAME} ${SSH_CONF_FILENAME}
    - git commit -m "generated and overwritten --> ${DJANGO_CONFIG_FILENAME} ${ANSIBLE_INVENTORY_FILENAME} ${SSH_CONF_FILENAME}"
    - git push origin mr-robot-branch-${CI_COMMIT_SHORT_SHA}-${current_datetime}
    - |
      curl --request POST --header "PRIVATE-TOKEN: ${MR_ROBOT_ACCESS_TOKEN}" \
      --form "source_branch=mr-robot-branch-${CI_COMMIT_SHORT_SHA}-${current_datetime}" \
      --form "target_branch=main" \
      --form "title=Merge Request By Infra Pipeline $CI_COMMIT_SHORT_SHA-${current_datetime}" \
      --form "description=This MR includes changes for commit $CI_COMMIT_SHORT_SHA --> ${DJANGO_CONFIG_FILENAME} ${ANSIBLE_INVENTORY_FILENAME} ${SSH_CONF_FILENAME}" \
      --form "remove_source_branch=true" \
      "https://gitlab.com/api/v4/projects/46957519/merge_requests"
```

Рис. 5.22 – Код етап створення запиту на злиття

```yaml
ansible_swarm:
  stage: deploy_swarm
  needs: ["apply"]
  image:
    name: ${ANSIBLE_IMAGE}
    pull_policy: if-not-present
    entrypoint: ['']
  tags:
    - core
  before_script:
    - mkdir -p /home/.ssh
    - echo "${SSH_PRIV}" | tr -d '\r' > /home/.ssh/${PRIV_KEY_NAME} && chmod 600 /home/.ssh/${PRIV_KEY_NAME}
    - cd $CI_PROJECT_DIR/ansible
  script:
    - |
     ansible-playbook
     --inventory-file ${ANSIBLE_INVENTORY_FILENAME}
     --ssh-extra-args "-F ${SSH_CONF_FILENAME}"
     swarm_formation.yml
```

Рис. 5.23 – Код етапу виконання Ansible

5.5.2 Конвеєр збірки коду

Для конвеєра збірки коду визначено два етапи:

— Етап створення Docker образу та його відправка до сховища Docker Hub;

— Етап створення запиту на злиття (потребує створеного Access Token).

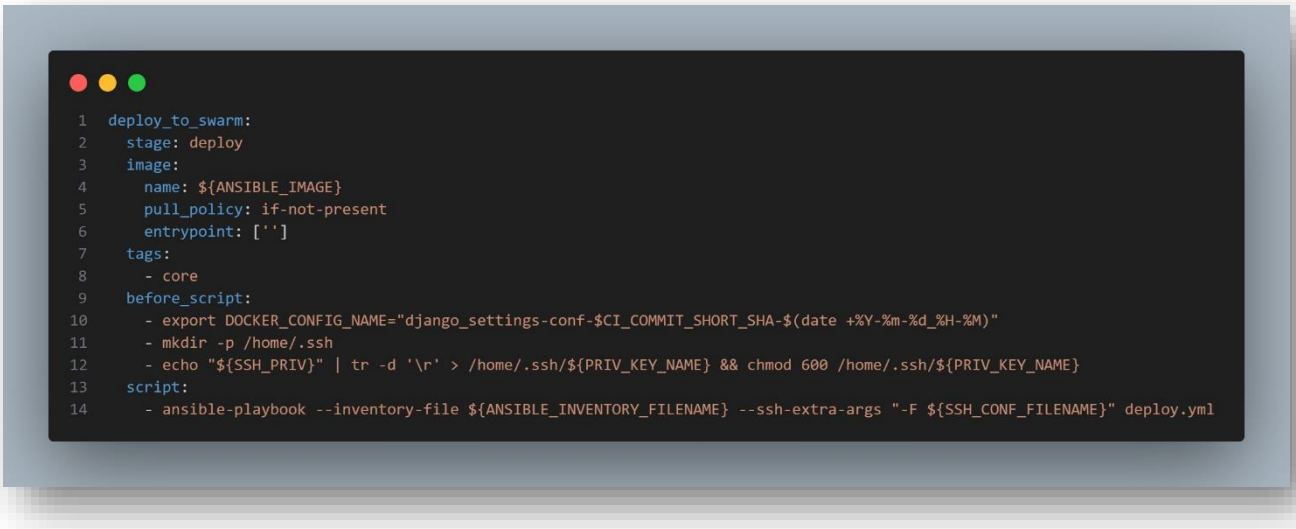
Зазвичай для створення Docker образів з використанням агентів типу «docker», використовують спеціальний сервіс DinD (Docker in Docker), що надає можливість «злиття» оточень. Для реалізації створення Docker образів у даній роботі було використано Kaniko Executor, що не потребує запуску контейнерів у привілейованому режимі.

```yaml
build_push_image:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
    pull_policy: if-not-present
  tags:
    - core
  before_script:
    - build_datetime=$(date +%Y-%m-%d_%H-%M)
    - |
      cat <<EOF > ${CI_PROJECT_DIR}/${DOCKER_TAG_FILENAME}
      ---
      DJANGO_APP_IMAGE: ${CI_REGISTRY_NAME}:${CI_COMMIT_SHORT_SHA}-${build_datetime}
      EOF
  script:
    - mkdir -p /kaniko/.docker
    - >-
      echo "{\"auths\":{\"${CI_REGISTRY}\":{\"auth\":\"$(printf "%s:%s" "${CI_REGISTRY_USER}" "${CI_REGISTRY_PASSWORD}" | base64 )\"}}}" >
      /kaniko/.docker/config.json
    - >-
      /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
      --destination "${CI_REGISTRY_NAME}:${CI_COMMIT_SHORT_SHA}-${build_datetime}"
  artifacts:
    when: on_success
    expire_in: 1 week
    paths:
      - ${CI_PROJECT_DIR}/${DOCKER_TAG_FILENAME}
```

Рис. 5.24 – Код етапу створення образу Docker

5.5.3 Конвеєр розгортання додатку

Останній компонент системи автоматизації – розгортання на цільовій платформі. Для даної задачі використано один етап, але саме цей етап збирає отримані артефакти, створює конфігураційний файл додатку із шаблону та файл для запуску у Docker Swarm.



```
deploy_to_swarm:
  stage: deploy
  image:
    name: ${ANSIBLE_IMAGE}
    pull_policy: if-not-present
    entrypoint: ['']
  tags:
    - core
  before_script:
    - export DOCKER_CONFIG_NAME="django_settings-conf-$CI_COMMIT_SHORT_SHA-$(date +%Y-%m-%d_%H-%M)"
    - mkdir -p /home/.ssh
    - echo "${SSH_PRIV}" | tr -d '\r' > /home/.ssh/${PRIV_KEY_NAME} && chmod 600 /home/.ssh/${PRIV_KEY_NAME}
  script:
    - ansible-playbook --inventory-file ${ANSIBLE_INVENTORY_FILENAME} --ssh-extra-args "-F ${SSH_CONF_FILENAME}" deploy.yml
```

Рис. 5.25 – Код етапу розгортання додатку

Основну роботу виконує Ansible, який виконує підстановку значень у шаблонах Jinja2 та використовує модуль docker для управління розгортанням на цільовому кластері. Ansible створює Docker config, який є файлом конфігурації для додатку і монтує його у потрібну директорію всередині контейнера.

При оновленні останньої версії Docker образу методом створення нової збірки, створюється відповідний запит на злиття із новим значенням Docker Tag в репозиторій розгортання, що автоматизує процес розгортання інтегрованих змін.

При зміні конфігурації інфраструктури, Terraform регенерує значення ALLOWED_HOSTS для додатку та за допомогою запиту на злиття, нова фігурація розгортається в кластері.

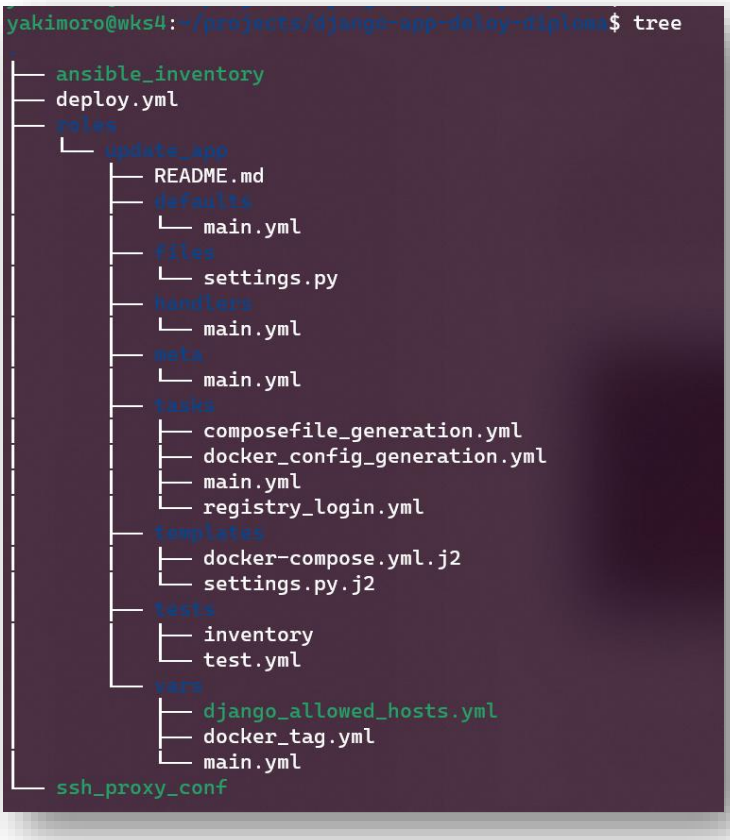
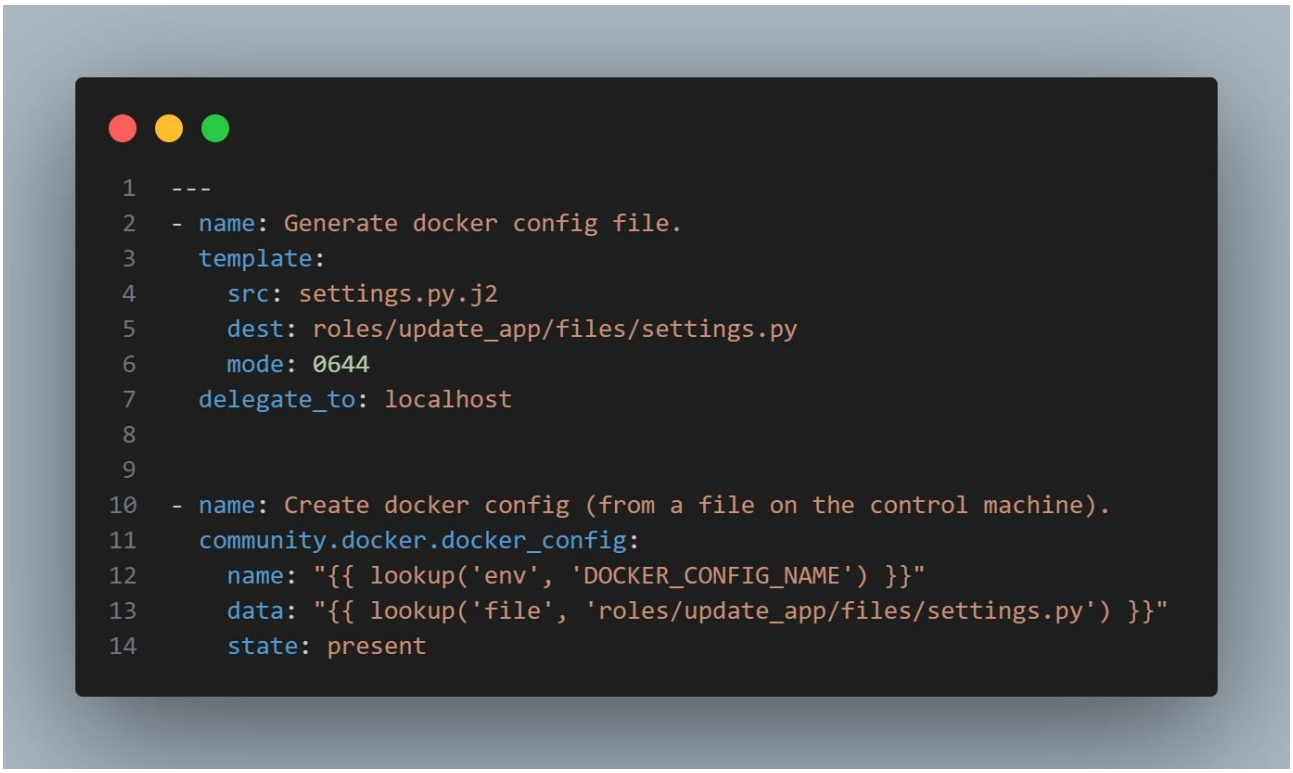


Рис. 5.25 – Структура директорій Ansible-galaxy для розгортання додатку

```
1
2  DATABASES = {
3      'default': {
4          'ENGINE': 'django.db.backends.mysql',
5          'HOST': '{{ lookup('env', 'DB_SERVICE_NAME') }}',
6          'NAME': '{{ lookup('env', 'DJANGO_DB_NAME') }}',
7          'USER': '{{ lookup('env', 'DJANGO_DB_USER') }}',
8          'PASSWORD': os.environ.get('DJANGO_DB_PASSWORD'),
9          'PORT': '{{ lookup('env', 'DJANGO_DB_PORT') }}',
10     }
11 }
```

Рис. 5.26 – Конфігурація підключення до бази даних в шаблоні «settings.py.j2»



```yaml
---
- name: Generate docker config file.
  template:
    src: settings.py.j2
    dest: roles/update_app/files/settings.py
    mode: 0644
  delegate_to: localhost


- name: Create docker config (from a file on the control machine).
  community.docker.docker_config:
    name: "{{ lookup('env', 'DOCKER_CONFIG_NAME') }}"
    data: "{{ lookup('file', 'roles/update_app/files/settings.py') }}"
    state: present
```

Рис. 5.27 – Код Ansible для генерації файлу конфігурації додатку та створення Docker Config

5.6 Розгляд і оцінка результатів

Результатом виконання конвеєру інфраструктури є розгорнуті в повному обсязі та готові до використання компоненти, визначені кодом Terraform та сконфігурований кластер Docker Swarm, готовий до використання.

Результатом виконання конвеєру створення збірки є артефакт, що успішно відправлено до Docker Hub.

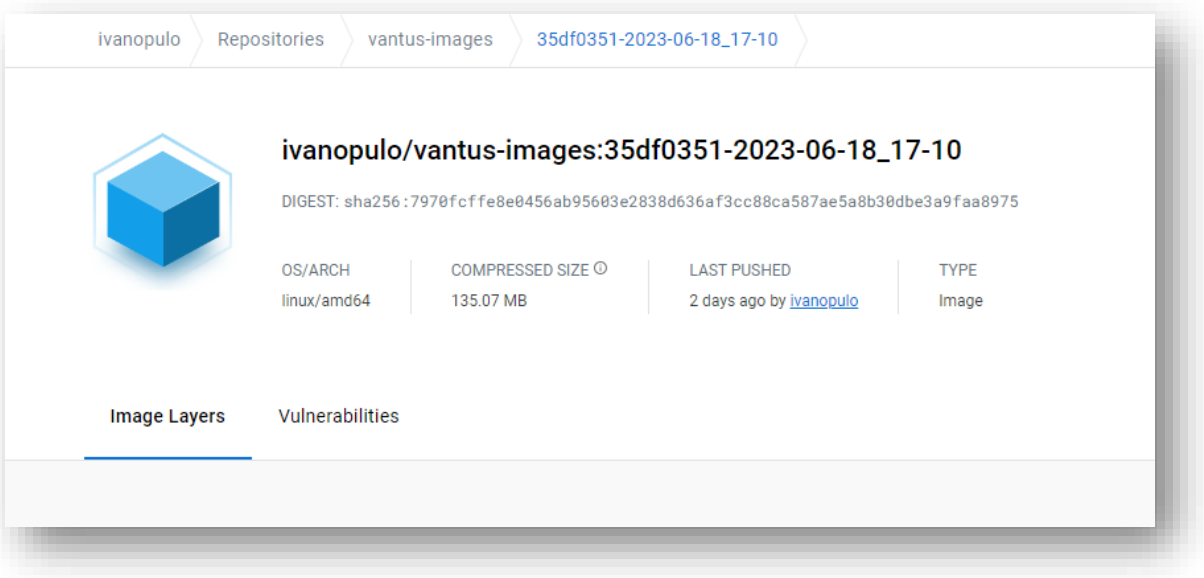


Рис. 5.28 – Образ додатку у сховищі Docker Hub

Результатом виконання конвеєру розгортання додатку є функціонуючий додаток, що розгорнуто у Docker Swarm із повними можливостями, закладеними на етапі проєктування.

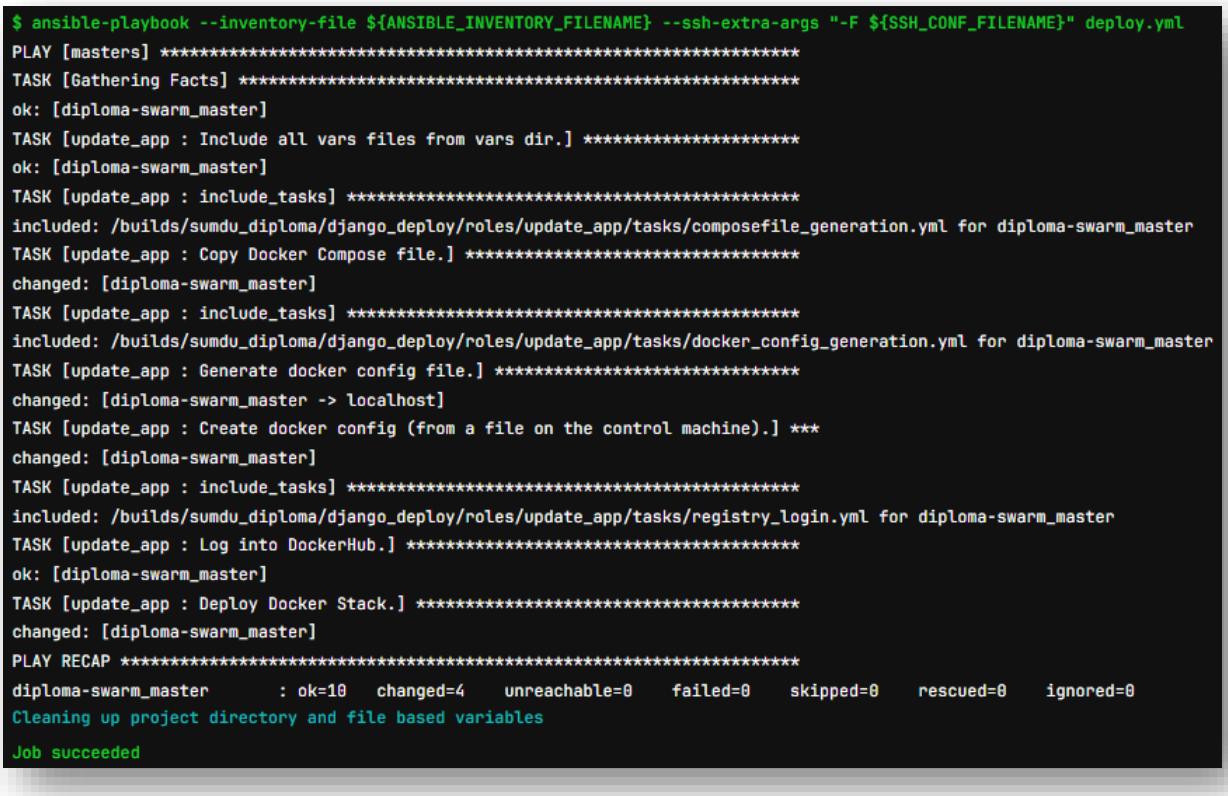


Рис. 5.29 – Результат виконання конвеєру розгортання

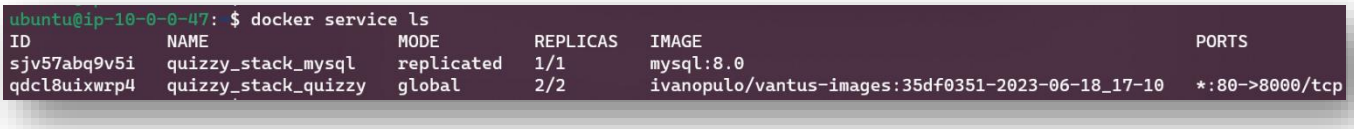


Рис. 5.30 – Статус сервісів у кластері

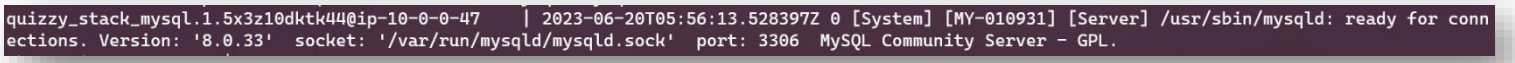


Рис. 5.31 – Логи бази даних

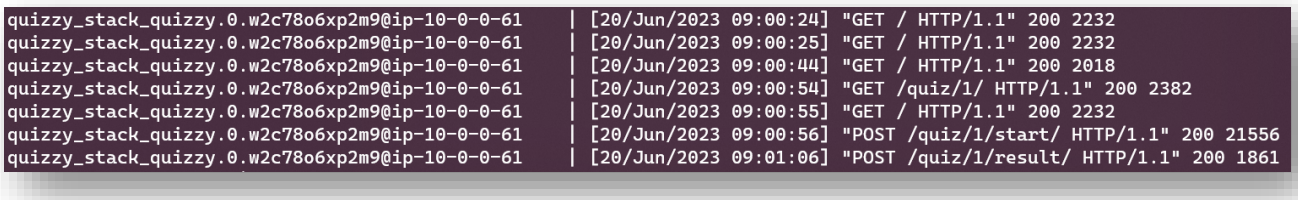


Рис. 5.32 – Логи додатку

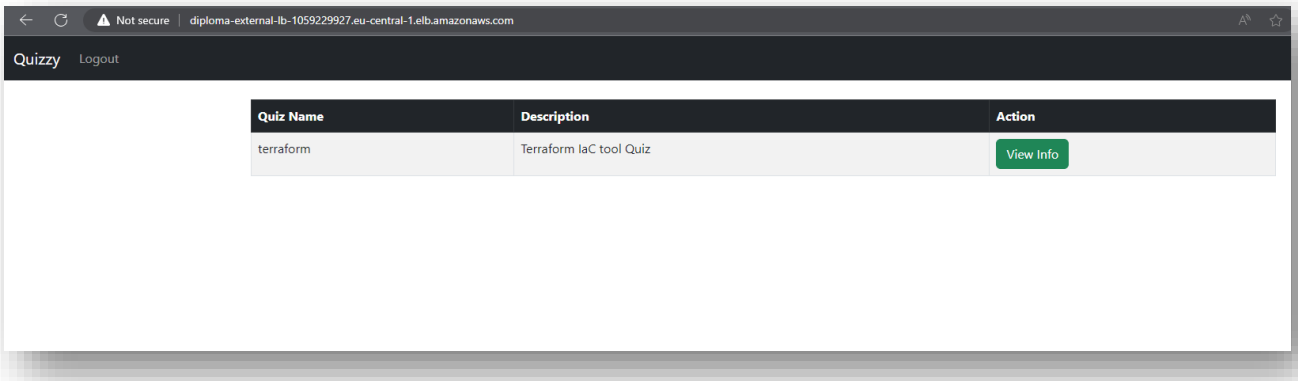


Рис. 5.33 – Головна сторінка додатку

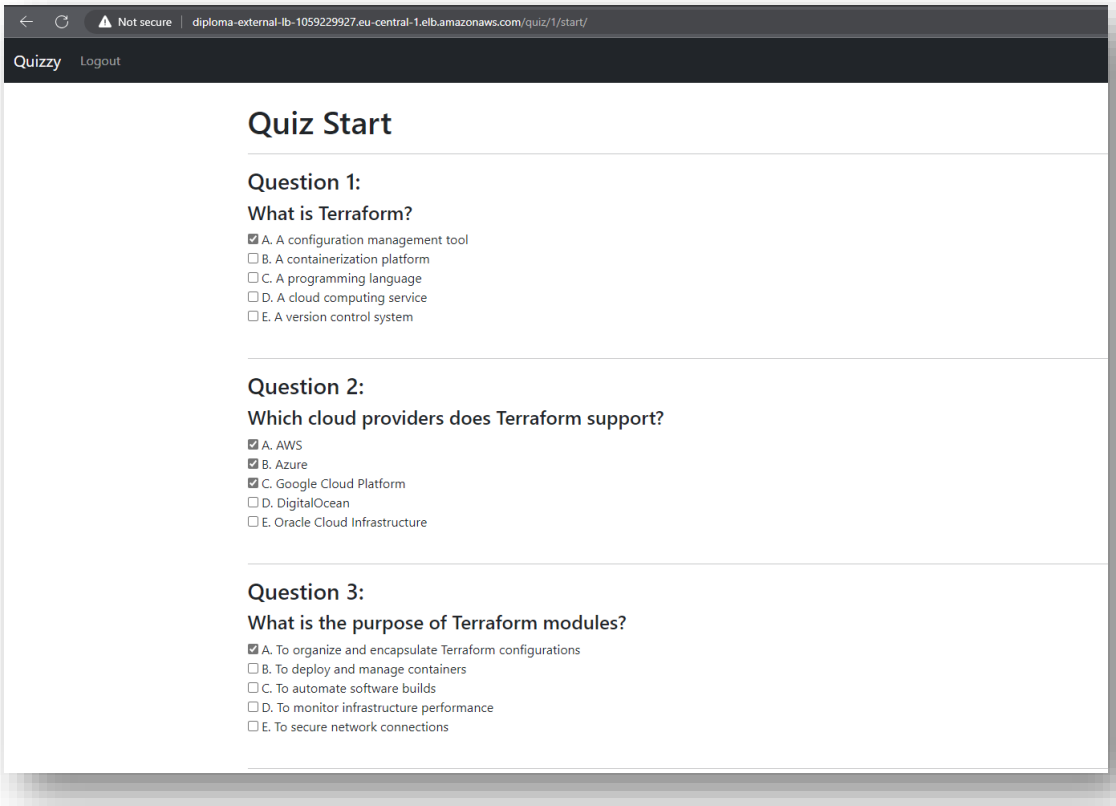


Рис. 5.34 – Сторінка проходження тесту

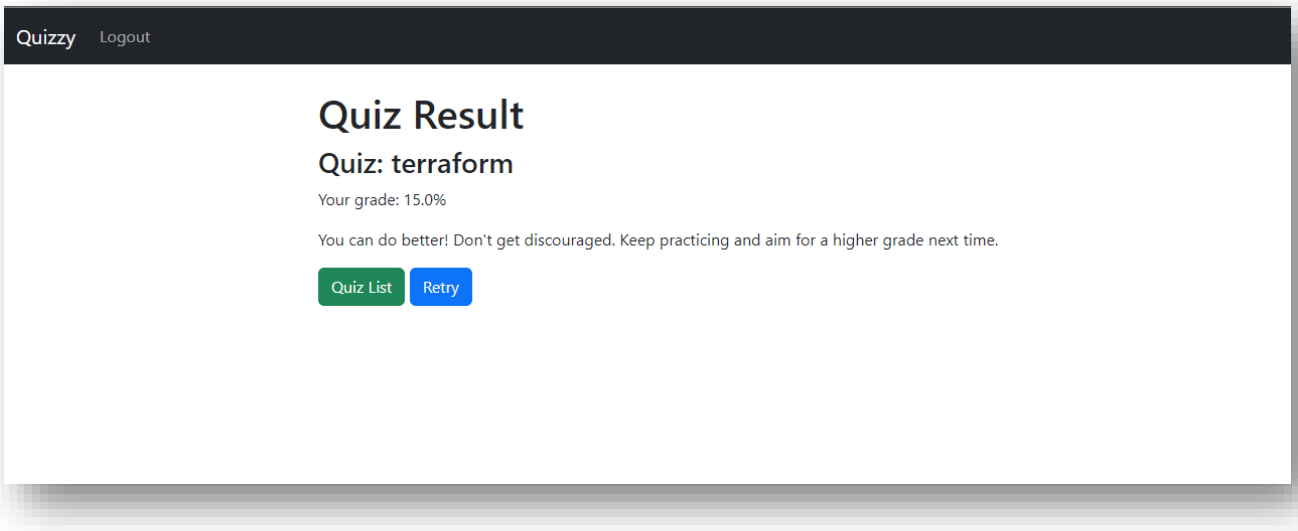


Рис. 5.35 – Сторінка результату тесту

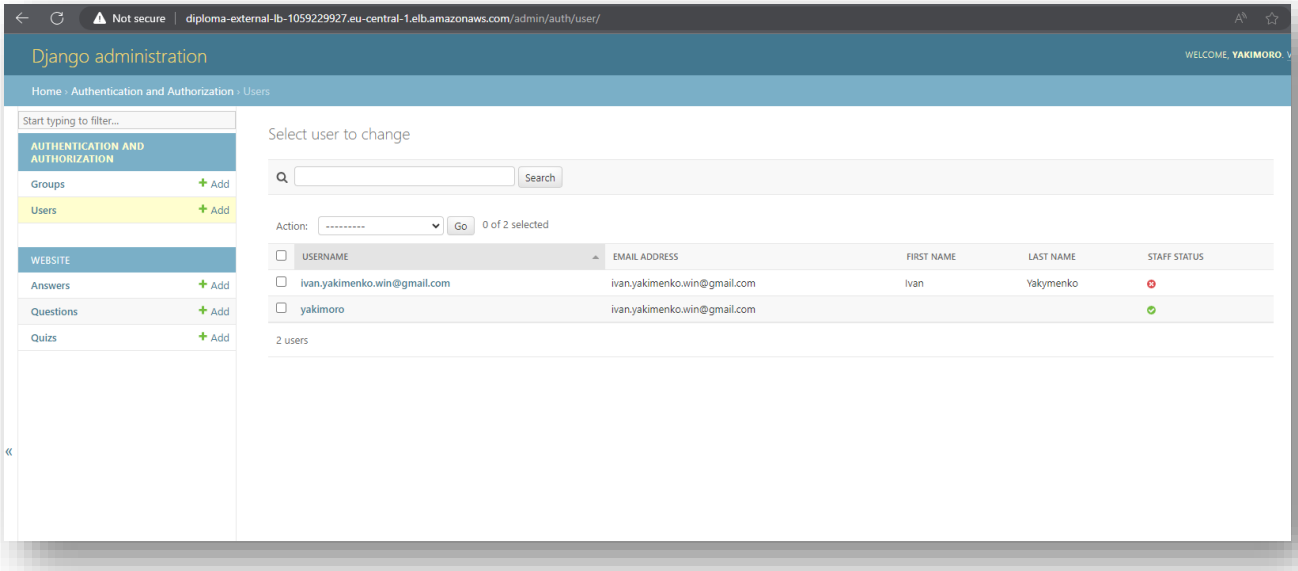


Рис. 5.36 – Сторінка панелі адміністратора

ВИСНОВКИ

У процесі виконання роботи було досліджено сучасні методи та підходи розробки веб-додатків, розглянуто можливості сучасних інструментів автоматизації розгортання, розробки, інтеграції та підтримки програмного забезпечення.

Результатом виконання роботи є веб-додаток для контролю знань у формі тестувань та система його автоматичного розгортання, оновлення та підтримки.

Веб-додаток може використовуватися як для локального застосування так і для організацій чи установ.

СПИСОК ЛІТЕРАТУРИ

1. Djangoproject - Django Framework [Електронний ресурс] // www.djangoproject.com. Режим доступу до ресурсу: https://www.djangoproject.com.

2. Mozilla Developer - Django [Електронний ресурс] // developer.mozilla.org. Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django.

3. W3Schools - Django [Електронний ресурс] // www.w3schools.com. Режим доступу до ресурсу: https://www.w3schools.com/django.

4. Mozilla Developer - HTML [Електронний ресурс] // developer.mozilla.org. Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/HTML.

5. Mozilla Developer - CSS [Електронний ресурс] // developer.mozilla.org. Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics.

6. Getbootstrap - Bootsrap 5 [Електронний ресурс] // getbootstrap.com. Режим доступу до ресурсу: https://getbootstrap.com/docs/5.0/getting-started/introduction.

7. Mysql - Mysql [Електронний ресурс] // www.mysql.com. Режим доступу до ресурсу: https://www.mysql.com.

8. W3Schools - Mysql [Електронний ресурс] // www.w3schools.com. Режим доступу до ресурсу: https://www.w3schools.com/MySQL/default.asp.

9. Docker Docks - Docker [Електронний ресурс] // docs.docker.com. Режим доступу до ресурсу: https://docs.docker.com/get-started/overview.

10. AWS Docks - AWS [Електронний ресурс] // docs.aws.amazon.com. Режим доступу до ресурсу: https://docs.aws.amazon.com.

11. Microsift Leard - IaC [Електронний ресурс] // learn.microsoft.com. Режим доступу до ресурсу: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code.

12. Redhat - IaC [Електронний ресурс] // www.redhat.com. Режим доступу до ресурсу: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac.

13. HashiCorp Developer - Terraform [Електронний ресурс] // developer.hashicorp.com. Режим доступу до ресурсу: https://developer.hashicorp.com/terraform/intro.

14. Teraform Provider Registry - Terraform AWS [Електронний ресурс] // registry.terraform.io. Режим доступу до ресурсу: https://registry.terraform.io/providers/hashicorp/aws/latest/docs.

15. Ansible Documentation - Ansible [Електронний ресурс] // www.ansible.com. Режим доступу до ресурсу: https://www.ansible.com/overview/how-ansible-works.

16. GitLab Docs - Gitlab [Електронний ресурс] // docs.gitlab.com. Режим доступу до ресурсу: https://docs.gitlab.com.

17. Baeldung - SSH proxying [Електронний ресурс] // www.baeldung.com. Режим доступу до ресурсу: https://www.baeldung.com/linux/ssh-tunneling-and-proxying.

18. Git Documentation - Git [Електронний ресурс] // git-scm.com. Режим доступу до ресурсу: https://git-scm.com/doc.

19. 12 Factor - Creating Application [Електронний ресурс] // 12factor.net. Режим доступу до ресурсу: https://12factor.net.

ДОДАТОК А

```
from django.shortcuts import render, redirect, get_object_or_404
from django.contrib.auth import authenticate, login, logout
from django.contrib import messages
from django.db.models import Count
from .forms import SignUpForm
from .models import Quiz, Question, Answer



def home(request):
    quizzes = Quiz.objects.all()

    # Check if the user is authenticated
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            messages.success(request, "You Have Been Logged In!")
            return redirect('home')
        else:
            messages.success(request, "There Was An Error Logging In, Please Try
Again...")
            return redirect('home')

    return render(request, 'quiz_home.html', {'quizzes': quizzes})
```

```
def logout_user(request):
    logout(request)
    messages.success(request, "You Have Been Logged Out...")
    return redirect('home')


def register_user(request):
    if request.method == 'POST':
        form = SignUpForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data['username']
            password = form.cleaned_data['password1']
            user = authenticate(username=username, password=password)
            login(request, user)
            messages.success(request, "You Have Successfully Registered! Welcome!")
            return redirect('home')
    else:
        form = SignUpForm()

    return render(request, 'register.html', {'form': form})


def quiz_list(request):
    if request.user.is_authenticated:
        quizzes = Quiz.objects.all()
        return render(request, 'quiz_home.html', {'quizzes': quizzes})
    else:
        return redirect('home')
```

```
def quiz_info(request, quiz_id):
    if request.user.is_authenticated:
        quiz = get_object_or_404(Quiz, id=quiz_id)
        return render(request, 'quiz_info.html', {'quiz': quiz})
    else:
        return redirect('home')

def quiz_start(request, quiz_id):
    if request.user.is_authenticated:
        quiz = get_object_or_404(Quiz, id=quiz_id)
        questions = Question.objects.filter(quiz=quiz)
        return render(request, 'quiz_start.html', {'quiz': quiz, 'questions': questions})
    else:
        return redirect('home')



def quiz_result(request, quiz_id):
    if request.user.is_authenticated:
        quiz = get_object_or_404(Quiz, id=quiz_id)
        questions = Question.objects.filter(quiz=quiz)
        total_questions = questions.count()
        correct_answers = 0

        for question in questions:
            selected_answer_id = request.POST.get(f'answer_{question.id}')
            if selected_answer_id:
                selected_answer = get_object_or_404(Answer, id=selected_answer_id)
```

```
            if selected_answer.is_correct:
                correct_answers += 1

        grade = (correct_answers / total_questions) * 100 if total_questions > 0 else 0
        grade = round(grade, 2)  # Round grade to two decimal places


        return render(request, 'quiz_result.html', {'quiz': quiz, 'grade': grade})
    else:
        return redirect('home')


from django.db import models


class Quiz(models.Model):
    name = models.CharField(max_length=100)
    short_description = models.CharField(max_length=200)
    long_description = models.TextField()

    def __str__(self):
        return self.name


class Question(models.Model):
    quiz = models.ForeignKey(Quiz, on_delete=models.CASCADE)
    content = models.TextField()

    def __str__(self):
```

```
        return self.content


class Answer(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    content = models.TextField()
    is_correct = models.BooleanField()

    def __str__(self):
        return self.content



from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User
from django import forms

class SignUpForm(UserCreationForm):
        email = forms.EmailField(label="",
widget=forms.TextInput(attrs={'class':'form-control', 'placeholder':'Email Address'}))
        first_name = forms.CharField(label="", max_length=100,
widget=forms.TextInput(attrs={'class':'form-control', 'placeholder':'First Name'}))
        last_name = forms.CharField(label="", max_length=100,
widget=forms.TextInput(attrs={'class':'form-control', 'placeholder':'Last Name'}))



        class Meta:
                model = User
                fields = ('username', 'first_name', 'last_name', 'email', 'password1',
'password2')
```

```
    def __init__(self, *args, **kwargs):
        super(SignUpForm, self).__init__(*args, **kwargs)

        self.fields['username'].widget.attrs['class'] = 'form-control'
        self.fields['username'].widget.attrs['placeholder'] = 'User Name'
        self.fields['username'].label = ''
        self.fields['username'].help_text = '<span class="form-text text-muted"><small>Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.</small></span>'

        self.fields['password1'].widget.attrs['class'] = 'form-control'
        self.fields['password1'].widget.attrs['placeholder'] = 'Password'
        self.fields['password1'].label = ''
        self.fields['password1'].help_text = '<ul class="form-text text-muted small"><li>Your password can\'t be too similar to your other personal information.</li><li>Your password must contain at least 8 characters.</li><li>Your password can\'t be a commonly used password.</li><li>Your password can\'t be entirely numeric.</li></ul>'

        self.fields['password2'].widget.attrs['class'] = 'form-control'
        self.fields['password2'].widget.attrs['placeholder'] = 'Confirm Password'
        self.fields['password2'].label = ''
        self.fields['password2'].help_text = '<span class="form-text text-muted"><small>Enter the same password as before, for verification.</small></span>'
```

ДОДАТОК Б

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
  backend "s3" {
    bucket = "diploma-tfstate-bucket"
    key    = "terraform.tfstate"
    region = "eu-central-1"
  }
}

provider "aws" {
  #  region                  = "eu-central-1"
  #  shared_credentials_files = ["${var.credfile_location}${var.credfile_name}"]
}
resource "aws_key_pair" "app_instance_key" {
  for_each   = var.vms
  key_name   = "${var.project_prefix}-${each.key}-app_instance_key"
  public_key = file("${var.ssh_keys_location}${var.pub_key_name}")
}

resource "aws_instance" "app_instance" {
  for_each              = var.vms
  ami                   = each.value.instance_ami
```

```
  instance_type          = each.value.instance_type
  key_name               = aws_key_pair.app_instance_key[each.key].key_name
  subnet_id              = aws_subnet.private_subnet.id
  vpc_security_group_ids = [aws_security_group.application_vms_sg.id]

  tags = {
    Name      = "${var.project_prefix}-${each.key}"
    is_master = each.value.is_master ? "true" : "false"
  }
}
resource "aws_security_group" "application_vms_sg" {
  name        = "${var.project_prefix}-application-vms-sg"
  description = "Security group for the application node"
  vpc_id      = aws_vpc.project_vpc.id

  ingress {
    from_port       = 80
    to_port         = 80
    protocol        = "tcp"
    security_groups = [aws_security_group.lb_sg.id]
  }

  ingress {
    from_port       = 22
    to_port         = 22
    protocol        = "tcp"
    security_groups = [aws_security_group.bastion_sg.id]
  }
```

```
  ingress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    self      = true
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "local_file" "ssh_proxy_conf" {
  content = templatefile("${path.module}/tf_templates/ssh_proxy_conf.tpl", {
    bastion_ip    = aws_instance.bastion.public_ip
    bastion_name  = aws_instance.bastion.tags.Name
    identity_file = "/home/.ssh/${var.priv_key_name}"
    nodes = tomap({
      for instance in aws_instance.app_instance : instance.tags.Name =>
instance.private_ip
    })
  })
  filename = "${path.module}/ansible/${var.ssh_conf_filename}"
}

resource "local_file" "ansible_inventory" {
```

```
  content = templatefile("${path.module}/tf_templates/ansible_inventory.tpl",
    {
      masters = [for instance in aws_instance.app_instance : instance.tags.Name if instance.tags.is_master == "true"]
      workers = [for instance in aws_instance.app_instance : instance.tags.Name if instance.tags.is_master == "false"]
    }
  )
  filename = "${path.module}/ansible/${var.ansible_inventory_filename}"
}

resource "local_file" "django_allowed_hosts" {
  content = templatefile("${path.module}/tf_templates/django_allowed_hosts.tpl", {
    load_balancer_dns_name = aws_lb.external_lb.dns_name
    nodes = tomap({
      for instance in aws_instance.app_instance : instance.tags.Name => instance.private_ip
    })
  })
  filename = "${path.module}/${var.django_config_filename}"
}

resource "aws_vpc" "project_vpc" {
  cidr_block = var.vpc_cidr
  tags = {
    Name = "${var.project_prefix}-project-vpc"
  }
}

resource "aws_eip" "nat_gateway_eip" {
```

```
  vpc = true
}

resource "aws_nat_gateway" "vpc_nat_gateway" {
  allocation_id = aws_eip.nat_gateway_eip.id
  subnet_id     = aws_subnet.public_subnet1.id
}

resource "aws_internet_gateway" "vpc_internet_gateway" {
  vpc_id = aws_vpc.project_vpc.id

  tags = {
    Name = "${var.project_prefix}-internet-gateway"
  }
}

resource "aws_route_table" "public_route_table" {
  vpc_id = aws_vpc.project_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.vpc_internet_gateway.id
  }
}

resource "aws_route_table" "private_route_table" {
  vpc_id = aws_vpc.project_vpc.id

  route {
    cidr_block     = "0.0.0.0/0"
```

```
    nat_gateway_id = aws_nat_gateway.vpc_nat_gateway.id
  }
}

resource "aws_route_table_association" "public_subnet1_routing_association" {
  subnet_id      = aws_subnet.public_subnet1.id
  route_table_id = aws_route_table.public_route_table.id
}

resource "aws_route_table_association" "private_subnet_routing_association" {
  subnet_id      = aws_subnet.private_subnet.id
  route_table_id = aws_route_table.private_route_table.id
}

resource "aws_subnet" "private_subnet" {
  vpc_id            = aws_vpc.project_vpc.id
  cidr_block        = var.private_subnet_cidr
  availability_zone = var.az[0]

  tags = {
    Name = "${var.project_prefix}-private-subnet"
  }
}

resource "aws_subnet" "public_subnet1" {
  vpc_id                  = aws_vpc.project_vpc.id
  cidr_block              = var.public_subnet1_cidr
  availability_zone       = var.az[0]
  map_public_ip_on_launch = true
```

```
  tags = {
    Name = "${var.project_prefix}-public-subnet1"
  }
}

resource "aws_subnet" "public_subnet2" {
  vpc_id                  = aws_vpc.project_vpc.id
  cidr_block              = var.public_subnet2_cidr
  availability_zone       = var.az[1]
  map_public_ip_on_launch = true

  tags = {
    Name = "${var.project_prefix}-public-subnet2"
  }
}
output "app_instances_internal_ips" {
  description = "Internal IP addresses of the app instances"
  value       = { for instance in aws_instance.app_instance : instance.tags.Name => instance.private_ip }
}

output "bastion_host_public_ip" {
  description = "Public IP address of the bastion instance"
  value       = aws_instance.bastion.public_ip
}

output "load_balancer_dns_name" {
  description = "Public DNS name of the load balancer"
```

```
  value       = aws_lb.external_lb.dns_name
}
resource "aws_key_pair" "bastion_ssh_key" {
  key_name   = "${var.project_prefix}-bastion_ssh_key"
  public_key = file("${var.ssh_keys_location}${var.pub_key_name}")
}

resource "aws_instance" "bastion" {
  ami           = var.bastion_instance_ami
  instance_type = var.bastion_instance_type
  key_name      = aws_key_pair.bastion_ssh_key.key_name
  subnet_id     = aws_subnet.public_subnet1.id

  vpc_security_group_ids = [
    aws_security_group.bastion_sg.id
  ]

  tags = {
    Name = "${var.project_prefix}-bastion"
  }
}

resource "aws_lb" "external_lb" {
  name               = "${var.project_prefix}-external-lb"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.lb_sg.id]
  subnets = [
    aws_subnet.public_subnet1.id,
```

```
    aws_subnet.public_subnet2.id
  ]

  tags = {
    Name = "${var.project_prefix}-external-lb"
  }
}

resource "aws_lb_target_group" "external_lb_target_group" {
  name_prefix = "target"
  port        = 80
  protocol    = "HTTP"
  vpc_id      = aws_vpc.project_vpc.id

  health_check {
    path     = "/"
    protocol = "HTTP"
  }
}

resource "aws_lb_listener" "external_lb_listener" {
  load_balancer_arn = aws_lb.external_lb.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    target_group_arn = aws_lb_target_group.external_lb_target_group.arn
    type             = "forward"
  }
```

```
}

resource "aws_lb_listener_rule" "external_lb_listener_rule" {
  listener_arn = aws_lb_listener.external_lb_listener.arn
  priority     = 1

  action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.external_lb_target_group.arn
  }

  condition {
    path_pattern {
      values = ["/"]
    }
  }
}

resource "aws_lb_target_group_attachment" "external_lb_target_group_attachment"
{
  for_each         = { for k, v in aws_instance.app_instance : k => v if v.tags.is_master
== "false" }
  target_group_arn = aws_lb_target_group.external_lb_target_group.arn
  target_id        = aws_instance.app_instance[each.key].id
}
resource "aws_security_group" "bastion_sg" {
  name        = "${var.project_prefix}-bastion-sg"
  description = "Security group for the bastion host"
  vpc_id      = aws_vpc.project_vpc.id
```

```
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "lb_sg" {
  name        = "${var.project_prefix}-lb-sg"
  description = "Security group for the load balancer"
  vpc_id      = aws_vpc.project_vpc.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
```

```
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
vms = {
  swarm_master = {
    instance_type = "t2.micro",
    instance_ami  = "ami-04e601abe3e1a910f",
    is_master     = true
  },
  swarm_worker1 = {
    instance_type = "t2.micro",
    instance_ami  = "ami-04e601abe3e1a910f",
    is_master     = false
  },
  swarm_worker2 = {
    instance_type = "t2.micro",
    instance_ami  = "ami-04e601abe3e1a910f",
    is_master     = false
  }
}

bastion_instance_type = "t2.micro"

bastion_instance_ami = "ami-04e601abe3e1a910f"
#----------------------Naming-Vars
variable "project_prefix" {
  description = "Resource naming prefix"
```

```
  type        = string
  default     = "diploma"
}

#-----------------------Credential_Vars

variable "credfile_location" {
  type    = string
  default = "~/.aws/"
}

variable "credfile_name" {
  type    = string
  default = "credentials"
}

#-----------------------Placement-Vars
variable "region" {
  type    = string
  default = "eu-central-1"
}

variable "az" {
  type        = list(string)
  description = "avaliability zone"
  default     = ["eu-central-1a", "eu-central-1b", "eu-central-1c"]
}

#-----------------------Network-Vars
```

```
variable "vpc_cidr" {
  type        = string
  description = "CIDR vlock in vpc"
  default     = "10.0.0.0/16"
}

variable "private_subnet_cidr" {
  type        = string
  description = "CIDR for the public subnet"
  default     = "10.0.0.0/24"
}

variable "public_subnet1_cidr" {
  type        = string
  description = "CIDR block for the public subnet"
  default     = "10.0.1.0/24"
}

variable "public_subnet2_cidr" {
  type        = string
  description = "CIDR block for the public subnet"
  default     = "10.0.2.0/24"
}

#----------------------Instances-conf-Vars

variable "vms" {
  description = "Map of vm names/configuration."
  type = map(object({
```

```
    instance_type = string,
    instance_ami  = string,
    is_master     = bool
  }))
}

variable "bastion_instance_type" {
  description = "An instance type for Bastion host"
  type        = string
  default     = "t2.micro"
}

variable "bastion_instance_ami" {
  description = "An ami to use for Bastion host"
  type        = string
  default     = "ami-04e601abe3e1a910f"
}

variable "ssh_keys_location" {
  description = "A directory with ssh keys"
  type        = string
  default     = "/home/yakimoro/.ssh/"
}

variable "priv_key_name" {
  description = "Private ssh key name"
  type        = string
  default     = "aws_diploma"
}
```

```
variable "pub_key_name" {
  description = "Public ssh key name"
  type        = string
  default     = "aws_diploma.pub"
}

#----------------------Generation-opt-Vars

variable "ansible_inventory_filename" {
  description = "The name for the inventory will be generated"
  type        = string
  default     = "ansible_inventory"
}

variable "ssh_conf_filename" {
  description = "The name for the ssh config will be generated"
  type        = string
  default     = "ssh_proxy_conf"
}

variable "django_config_filename" {
  description = "The name for the Django values yml file"
  type        = string
  default     = "django_allowed_hosts.yml"
}
```

ДОДАТОК В

```
image:
  name: ${TERRAFORM_IMAGE}
  pull_policy: if-not-present
  entrypoint:
    - '/usr/bin/env'
    - 'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
variables:
  AWS_SECRET_ACCESS_KEY: ${AWS_SECRET_ACCESS_KEY}
  AWS_ACCESS_KEY_ID: ${AWS_ACCESS_KEY_ID}
  AWS_REGION: ${AWS_DEFAULT_REGION}
  TF_VAR_project_prefix: ${PROJECT_PREFIX}
  TF_VAR_credfile_location: ${CREDFILE_LOCATION}
  TF_VAR_credfile_name: ${CREDFILE_NAME}
  TF_VAR_region: ${REGION}
  TF_VAR_az: ${AZ}
  TF_VAR_vpc_cidr: ${VPC_CIDR}
  TF_VAR_private_subnet_cidr: ${PRIVATE_SUBNET_CIDR}
  TF_VAR_public_subnet1_cidr: ${PUBLIC_SUBNET1_CIDR}
  TF_VAR_public_subnet2_cidr: ${PUBLIC_SUBNET2_CIDR}
  TF_VAR_ssh_keys_location: ${SSH_KEYS_LOCATION}
  TF_VAR_pub_key_name: ${PUB_KEY_NAME}
  TF_VAR_priv_key_name: ${PRIV_KEY_NAME}
```

```
  TF_VAR_ansible_inventory_filename: ${ANSIBLE_INVENTORY_FILENAME}
  TF_VAR_ssh_conf_filename: ${SSH_CONF_FILENAME}
  TF_VAR_django_config_filename: ${DJANGO_CONFIG_FILENAME}
before_script:
  - mkdir -p ${SSH_KEYS_LOCATION}
  - echo "${SSH_PUB}" > ${SSH_KEYS_LOCATION}${PUB_KEY_NAME}
  - terraform version
  - terraform init
cache:
  key: "$CI_COMMIT_SHA"
  paths:
    - .terraform

stages:
  - validate
  - plan
  - apply
  - deploy_swarm
  - push_artifacts

fmt:
  stage: validate
  tags:
```

```
    - core
  script:
    - terraform fmt -check
  only:
    changes:
      - "*.tf"
  allow_failure: true


validate:
  stage: validate
  tags:
    - core
  script:
    - terraform validate


plan:
  stage: plan
  needs: ["validate"]
  tags:
    - core
  script:
    - terraform plan -out "planfile"
  artifacts:
```

```
    when: on_success
    paths:
      - planfile
    expire_in: 1 week

apply:
  stage: apply
  needs: ["plan"]
  tags:
    - core
  script:
    - terraform apply -input=false "planfile"
  artifacts:
    when: on_success
    expire_in: 1 week
    paths:
      - ${CI_PROJECT_DIR}/ansible/${ANSIBLE_INVENTORY_FILENAME}
      - ${CI_PROJECT_DIR}/ansible/${SSH_CONF_FILENAME}
      - ${CI_PROJECT_DIR}/${DJANGO_CONFIG_FILENAME}
  when: manual

mr_job:
  stage: apply
```

```
  needs: ["apply"]
  image:
    name: registry.gitlab.com/sumdu_diploma/infra:curl-git
    pull_policy: if-not-present
    entrypoint: ['']
  tags:
    - core
  before_script:
    - current_datetime=$(date +%Y-%m-%d_%H-%M)
    - git config --global user.name "MR Robot"
    - git config --global user.email "mr.robot@robots.com"
  script:
    - |
     git clone

https://${MR_ROBOT_ACCESS_TOKEN_NAME}:${MR_ROBOT_ACCESS_TOKEN}@gitlab.com/sumdu_diploma/django_deploy.git
    - cd django_deploy
    - git checkout -b mr-robot-branch-${CI_COMMIT_SHORT_SHA}-${current_datetime}
    - cp ${CI_PROJECT_DIR}/ansible/${ANSIBLE_INVENTORY_FILENAME} ${ANSIBLE_INVENTORY_FILENAME}
    - cp ${CI_PROJECT_DIR}/ansible/${SSH_CONF_FILENAME} ${SSH_CONF_FILENAME}
```

```
    - cp ${CI_PROJECT_DIR}/${DJANGO_CONFIG_FILENAME}
./roles/update_app/vars/${DJANGO_CONFIG_FILENAME}

    - git add ./roles/update_app/vars/${DJANGO_CONFIG_FILENAME}
${ANSIBLE_INVENTORY_FILENAME} ${SSH_CONF_FILENAME}

    - git commit -m "generated and overwritten -->
${DJANGO_CONFIG_FILENAME} ${ANSIBLE_INVENTORY_FILENAME}
${SSH_CONF_FILENAME}"

    - git push origin mr-robot-branch-${CI_COMMIT_SHORT_SHA}-
${current_datetime}

    - |

      curl --request POST --header "PRIVATE-TOKEN:
${MR_ROBOT_ACCESS_TOKEN}" \

      --form "source_branch=mr-robot-branch-${CI_COMMIT_SHORT_SHA}-
${current_datetime}" \

      --form "target_branch=main" \

      --form "title=Merge Request By Infra Pipeline $CI_COMMIT_SHORT_SHA-
${current_datetime}" \

      --form "description=This MR includes changes for commit
$CI_COMMIT_SHORT_SHA --> ${DJANGO_CONFIG_FILENAME}
${ANSIBLE_INVENTORY_FILENAME} ${SSH_CONF_FILENAME}" \

      --form "remove_source_branch=true" \

      "https://gitlab.com/api/v4/projects/46957519/merge_requests"


ansible_swarm:

  stage: deploy_swarm
```

```
  needs: ["apply"]
  image:
    name: ${ANSIBLE_IMAGE}
    pull_policy: if-not-present
    entrypoint: ['']
  tags:
    - core
  before_script:
    - mkdir -p /home/.ssh
    - echo "${SSH_PRIV}" | tr -d '\r' > /home/.ssh/${PRIV_KEY_NAME} && chmod
600 /home/.ssh/${PRIV_KEY_NAME}
    - cd $CI_PROJECT_DIR/ansible
  script:
    - |
     ansible-playbook
     --inventory-file ${ANSIBLE_INVENTORY_FILENAME}
     --ssh-extra-args "-F ${SSH_CONF_FILENAME}"
     swarm_formation.yml


stages:
  - build
  - push_artifacts
```

```
build_push_image:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
    pull_policy: if-not-present
  tags:
    - core
  before_script:
    - build_datetime=$(date +%Y-%m-%d_%H-%M)
    - |
      cat <<EOF > ${CI_PROJECT_DIR}/${DOCKER_TAG_FILENAME}
      ---
      DJANGO_APP_IMAGE:
${CI_REGISTRY_NAME}:${CI_COMMIT_SHORT_SHA}-${build_datetime}
      EOF
  script:
    - mkdir -p /kaniko/.docker
    - >-
      echo "{\"auths\":{\"${CI_REGISTRY}\":{\"auth\":\"$(printf "%s:%s"
"${CI_REGISTRY_USER}" "${CI_REGISTRY_PASSWORD}" | base64 )\"}}}" >
      /kaniko/.docker/config.json
    - >-
```

```
    /kaniko/executor
    --context "${CI_PROJECT_DIR}"
    --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
    --destination "${CI_REGISTRY_NAME}:${CI_COMMIT_SHORT_SHA}-
${build_datetime}"
  artifacts:
    when: on_success
    expire_in: 1 week
    paths:
    - ${CI_PROJECT_DIR}/${DOCKER_TAG_FILENAME}

mr_job:
  stage: push_artifacts
  needs: ["build_push_image"]
  image:
    name: registry.gitlab.com/sumdu_diploma/infra:curl-git
    pull_policy: if-not-present
    entrypoint: ['']
  tags:
    - core
  before_script:
    - current_datetime=$(date +%Y-%m-%d_%H-%M)
    - git config --global user.name "MR Robot"
```

```
    - git config --global user.email "mr.robot@robots.com"
  script:
    - |
      git clone \
https://${MR_ROBOT_ACCESS_TOKEN_NAME}:${MR_ROBOT_ACCESS_TOKEN}@gitlab.com/sumdu_diploma/django_deploy.git
    - cd django_deploy
    - git checkout -b mr-robot-branch-${CI_COMMIT_SHORT_SHA}-${current_datetime}
    - touch ./roles/update_app/vars/${DOCKER_TAG_FILENAME}
    - cp ${CI_PROJECT_DIR}/${DOCKER_TAG_FILENAME} ./roles/update_app/vars/${DOCKER_TAG_FILENAME}
    - git add ./roles/update_app/vars/${DOCKER_TAG_FILENAME}
    - git commit -m "generated and overwritten --> ${DOCKER_TAG_FILENAME}"
    - git push origin mr-robot-branch-${CI_COMMIT_SHORT_SHA}-${current_datetime}
    - |
      curl --request POST --header "PRIVATE-TOKEN: ${MR_ROBOT_ACCESS_TOKEN}" \
      --form "source_branch=mr-robot-branch-${CI_COMMIT_SHORT_SHA}-${current_datetime}" \
      --form "target_branch=main" \
      --form "title=Merge Request By Build Pipeline $CI_COMMIT_SHORT_SHA-${current_datetime}" \
```

```
    --form "description=This MR includes changes for commit
$CI_COMMIT_SHORT_SHA --> ${DOCKER_TAG_FILENAME}" \
    --form "remove_source_branch=true" \
    https://gitlab.com/api/v4/projects/46957519/merge_requests
stages:
  - deploy


deploy_to_swarm:
  stage: deploy
  image:
    name: ${ANSIBLE_IMAGE}
    pull_policy: if-not-present
    entrypoint: ['']
  tags:
    - core
  before_script:
    - export DOCKER_CONFIG_NAME="django_settings-conf-
$CI_COMMIT_SHORT_SHA-$(date +%Y-%m-%d_%H-%M)"
    - mkdir -p /home/.ssh
    - echo "${SSH_PRIV}" | tr -d '\r' > /home/.ssh/${PRIV_KEY_NAME} && chmod
600 /home/.ssh/${PRIV_KEY_NAME}
  script:
    - ansible-playbook --inventory-file ${ANSIBLE_INVENTORY_FILENAME} --
ssh-extra-args "-F ${SSH_CONF_FILENAME}" deploy.yml
```