МІНІСТЕРСТВО ОБОРОНИ УКРАЇНИ
ВІЙСЬКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ ТА ІНФОРМАТИЗАЦІЇ ІМЕНІ ГЕРОЇВ КРУТ

Д.Ю. Меркотан, О.Я. Сова, О.О. Троцько,
О.А. Симоненко, О.В. Гаман,
О.Є. Степаненко, Г.Г. Мягких, В.П. Величко

Технології автоматизації системних процесів

Київ - 2021

МІНІСТЕРСТВО ОБОРОНИ УКРАЇНИ

ВІЙСЬКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ ТА ІНФОРМАТИЗАЦІЇ
імені ГЕРОЇВ КРУТ

Д.Ю. Меркотан, О.Я. Сова, О.О. Троцько, О.А. Симоненко,
О.В. Гаман, О.Є. Степаненко, Г.Г. Мягких, В.П. Величко

ТЕХНОЛОГІЇ АВТОМАТИЗАЦІЇ СИСТЕМНИХ ПРОЦЕСІВ

Навчальний посібник

Київ – 2021

УДК 004.715(075)
П 69

Рецензенти:
В. О. Сілко – кандидат технічних наук, доцент;
М. М. Нестеренко – кандидат технічних наук, доцент.

Технології автоматизації системних процесів: Навч. Посібник/Д.Ю. Меркотан, О.Я. Сова, О.О. Троцько, О.А. Симоненко, О.В. Гаман, О.Є. Степаненко, Г.Г. Мягких, В.П. Величко. Київ: ВІТІ, 2021. – 231 с.

Видання містить необхідний теоретичний матеріал щодо технологій автоматизації системних процесів, що є невід'ємною частиною розвитку інформаційних систем та технологій.

Матеріал навчального посібника забезпечить тим, хто навчається, можливість орієнтуватися та використовувати в подальшій професійній діяльності технології віртуалізації та контейнеризації, інструменти управління конфігурацією сервісів, мережевої та серверної інфраструктури, безперервну інтеграцію та розгортання програмного забезпечення.

Навчальний посібник призначений для курсантів (курсантів заочної форми навчання) та ад'юнктів інституту, які навчаються за спеціальністю 126 "Інформаційні системи та технології" і може бути корисним для курсантів та ад'юнктів суміжних спеціальностей під час вивчення ними дисциплін за профілем інформаційних систем та технологій.

Рекомендовано до друку Вченою радою Військового інституту телекомунікацій та інформатизації імені Героїв Крут (протокол № 14 від 25 травня 2021 року).

ЗМІСТ

СПИСОК УМОВНИХ СКОРОЧЕНЬ

ВМ – віртуальна електронна обчислювальна машина
ЕОМ – електронна обчислювальна машина
ЖЦ – життєвий цикл
ЗЕ – запам'ятовуючий елемент
ЗЗП – зовнішній запам'ятовуючий пристрій
ЗП – запам'ятовуючий пристрій
IaC – інфраструктура як код
МК – мікрокоманда
МО – мікрооперація
МП – мікропроцесор
ОМ – обчислювальна машина
ОП – основна пам'ять
Опр – операційний пристрій
ОС – операційна система
ПЗ – програмне забезпечення
ПК – персональний комп'ютер
СКВ – система контролю версій
СУ – сигнал управління
ЦОД – центр оброблення даних
ЦП – центральний процесор

API – Aplication program interface – програмний інтерфейс додатка
AWS – Amazon Web Services - Вебсервіси Амазон
CD – безперервна доставка
CD – Compact Disk – компакт-диск
CI – постійна інтеграція
CLI – інтерфейс командного рядка
COM – Communication – послідовний порт
CPU – Central Processing Unit – центральний процесор
DNS – Domain Name System – доменна система імен
DSL – Domain Specific Language – предметно-орієнтована мова опису станів серверів
Git – система контролю версій
Go – мова програмування
HCL – HashiCorp Configuration Language – мова конфігурації Terraform
HTML – HyperText Markup Language – мова розмітки гіпертексту
HTTP – HyperText Transfer Protocol – протокол передачі гіпертексту
IaaS – Infrastructure as a Service – інфраструктура як послуга
IAM – Identity and Access Management інструмент ідентифікації та доступу до сервісів Amazon
ISA – Industry Standard Architecture – стандартна індустріальна архітектура (системна шина)
JSON – JavaScript Object Notation – текстовий формат обміну даними між комп'ютерами
РКІ – інфраструктура відкритих ключів
REST – Representational State Transfer – передача репрезентативного стану
RDS – Radio Data System – багатоцільовий стандарт для передачі інформаційних повідомлень
SCCM – software configuration and change management – управління конфігурацією та змінами
SCM – software configuration management – управління конфігурацією
SSH – Secure Shell – протокол безпечного з'єднання на прикладному рівні моделі OSI
SQA – Software Quality Assurance – забезпечення якості програмного забезпечення
TCP – Transmission Control Protocol – протокол керування передаванням
TLS – Transport Layer Security – протокол захисту даних на транспортному рівні моделі OSI
tmpfs – тимчасове сховище інформації
YAML – формат серіалізації даних
VLAN – virtual local area network – віртуальна локальна мережа
VLIW – Very Long Instruction Word – архітектура процесора з довгим командним словом
VM – віртуальна ЕОМ
VPC – Virtual Private Cloud – віртуальна приватна хмара
WAR – запис після читання
WAW – запис після запису

ВСТУП

Навчальний посібник “Технології автоматизації системних процесів” підготовлений відповідно до першого, другого, третього та четвертого змістових модулів навчальної програми дисципліни “Технології автоматизації системних процесів”, яка передбачена освітньо-професійною та освітньо-науковою програмами підготовки здобувачів вищої освіти на першому та третьому рівні вищої освіти, за спеціальністю 126 “Інформаційні системи та технології”.

Навчальний посібник призначений для набуття курсантами та ад’юнктами вміння користуватися технологіями віртуалізації, інструментами управління конфігурацією сервісів, мережевої та серверної інфраструктури, інструментами, які забезпечують безперервну інтеграцію та розгортання програмного забезпечення.

Навчальний посібник складається з чотирьох розділів.

Перший розділ посібника присвячений технології контейнеризації.

Контейнери Docker надають прості швидкі і надійні методи розробки, поширення і запуску програмного забезпечення, особливо в динамічних і розподілених середовищах. В першому розділі посібника дізнаєтеся, чому контейнери так важливі, які переваги ви отримаєте від застосування Docker і як зробити Docker частиною процесу розробки. Ви послідовно пройдете по всіх етапах, необхідним для створення, тестування і розгортання будь-якого програмного забезпечення, що використовує Docker.

У другому розділі розглядаються поняття інфраструктури як коду та системи управління конфігураціями. Велика увага у розділі придається системі управління конфігураціями Ansible. Вона мінімалістична, не вимагає установки програмного забезпечення на вузлах, і легка в освоєнні. Ви дізнаєтеся, як написати скрипт управління конфігураціями, встановити контроль над віддаленими серверами, а також задіяти потужний функціонал вбудованих модулів.

У третьому розділі розглядається питання автоматизованого розгортання мережевої інфраструктури за допомогою Terraform. Terraform є справжньою зіркою в світі DevOps. Це технологія, що дозволяє конфігурувати, запускати і керувати хмарною інфраструктурою. “Інфраструктура як код” (IaC) дозволяє оптимально використовувати віртуалізовані платформи таких технологічних гігантів, як AWS, Google Cloud, Azure і інші. В цьому розділі представлені прості і лаконічні приклади коду, який використовується в Terraform для розгортання інфраструктури та управління нею.

Зміст четвертого розділу ознайомить вас з системою та інструментами контролю версій, за матеріалом цього розділу ви дізнаєтеся про поняття CI/CD, зрозумієте, як працювати з інструментом, що забезпечує постійну інтеграцію (CI) та безперервне доставлення (CD) програмного забезпечення Jenkins.

Постійна інтеграція та безперервна доставка є значною частиною культури DevOps. Jenkins – це повнофункціональна технологічна платформа, яка дозволяє користувачам застосовувати CI та CD. Це допомагає користувачам надавати кращі програми, автоматизуючи життєвий цикл доставки додатків. CI включає автоматизацію процесів побудови, тестування та пакетування. Jenkins дозволяє користувачеві використовувати послуги постійної інтеграції для розробки програмного забезпечення в гнучкому середовищі. Системи безперервної інтеграції є життєво важливою частиною продуктивної команди. Безперервна інтеграція є значною частиною культури DevOps, а отже, багато відкритих джерел та комерційні інструменти для безперервної доставки використовують Jenkins. Jenkins дозволяє командам зосередитись на роботі та інноваціях, автоматизуючи процеси збирання, управління артефактами та розгортання, а не турбуючись про ручні процеси.

За кожним з розділів навчального посібника підготовлено перелік контрольних питань для самоконтролю, з метою розвитку творчого та критичного мислення в процесі вивчення матеріалу посібника.

Навчальний посібник призначений для курсантів та ад'юнктів, які вивчають дисципліну "Технології автоматизації системних процесів", а також може використовуватись для розробки курсових, кваліфікаційних, дипломних та дисертаційних робіт.

Автори висловлюють подяку рецензентам кандидату технічних наук, доценту Сілку О.В., а також кандидату технічних наук, доценту Нестеренку М.М. за цінні зауваження та рекомендації, які сприяли покращенню цього видання і позбавленню низки помилок.

1. ТЕХНОЛОГІЇ КОНТЕЙНЕРИЗАЦІЇ

1.1. Ознайомлення з технологіями контейнеризації

Контейнери докорінно змінюють спосіб розробки, поширення і функціонування програмного забезпечення. Розробники можуть створювати програмне забезпечення на локальній системі, точно знаючи, що воно буде працювати однаково в будь-якому операційному середовищі, на ноутбуці користувача або в хмарному кластері. Інженери з експлуатації можуть зосередитися на підтримці роботи в мережі, на надання ресурсів і на забезпеченні безперебійної роботи і витрачати менше часу на конфігурацію оточення і на «боротьбу» з системними залежностями. Масштаби переходу до практичного застосування контейнерів стрімко ростуть у всій індустрії інформаційних технологій, від невеликих стартапів до великих підприємств. Розробники та інженери з експлуатації повинні розуміти, що необхідність постійного використання контейнерів буде зростати протягом кількох наступних років.

***Контейнери* (*containers*)** представляють собою засоби інкапсуляції додатка разом з його залежностями. На перший погляд контейнери можуть здатися всього лише спрощеною формою віртуальних машин (*virtual machines - VM*), але це не так.

Віртуальні машини емулюють весь комп'ютер, включаючи апаратне забезпечення. Для виртуалізації (тобто емуляції) процесору, пам'яті, жорсткого диску та мережі запускається гіпервізор, такий як *VMWare, VirtualBox, Proxmox*. Перевага цього підходу – будь-який образ віртуальної машини, який працює поверх гіпервізора може бачити тільки віртуальне обладнання, тому він повністю ізольований від фізичного комп'ютера та будь-яких інших образів ВМ. І він виконується аналогічно у всіх середовищах (наприклад, на домашньому пк або на робочому сервері). Недолік у тому, що віртуалізація всього обладнання та запуск окремої ОС для кожної ВМ потребує більшої кількості ресурсу процесора та пам'яті, що впливає на час запуску. Образи віртуальних машин можливо описувати у вигляді коду, використовуються такі інструменти як *Packet* та *Vagrant*.

Контейнери емулюють користувальницьке середовище ОС. Для ізоляції процесів, пам'яті, точок монтування та мережі запускається середовище виконання контейнерів, таке як Docker. Перевагою цього методу полягає в тому, що будь-який контейнер, який використовується у даному середовищі, може бачити лише свій користувальницький простір, тому він ізольований від основного комп'ютера та інших контейнерів. При цьому він веде себе однаково у будь-якому середовищі. Але є і недолік, всі контейнери запущені на одному хості, одночасно використовують ядро його ОС та його обладнання, тому досягти того рівня ізоляції та безпеки, який ви отримуєте, використовуючи технології віртуалізації – набагато важче. Оскільки використовується спільне ядро та обладнання, ваші контейнери можуть завантажуватися за лічені мілісекунди і, практично, не будуть

вимагати додаткових ресурсів процесору або пам’яті. Образи контейнерів можливо описати у вигляді коду, використовуючи *Docker*.

У сучасних ОС, код виконується в одному із двох “середовищ”:

1. ***Середовище ядра***. Код який працює у середовищі ядра має прямий та необмежений доступ до всього обладнання. На нього не діють обмеження безпеки, ви маєте змогу виконувати будь-які процесорні інструкції, звертатися до будь-якої ділянки жорсткого диску, записувати у будь-яку адресу пам’яті. При цьому збій у середовищі ядра зазвичай призводить до збою всього комп’ютера. У зв’язку з цим воно відводиться для самих низькорівневих та довірених функцій ОС.
2. ***Середовище користувача***. Код у середовищі користувача не має безпосереднього доступу до апаратного забезпечення і замість цього повинен використовувати API (*application program interface*), які надає ядро ОС. Ці інтерфейси можуть накладати обмеження безпеки (наприклад, права доступу) та локалізувати збої у користувальницьких додатках, тому весь прикладний код працює у користувалбницькому середовищі.

Контейнери мають деякі переваги, що забезпечують такі варіанти використання, які важко або неможливо реалізувати в звичайних віртуальних машинах:

– контейнери спільно використовують ресурси основної ОС, що робить їх на порядок більш ефективними. Контейнери можна запускати і зупиняти за частки секунди;

– для додатків, що запускаються в контейнерах, накладні витрати мінімальні або взагалі відсутні, в порівнянні з додатками, що запускаються безпосередньо під управлінням основної ОС;

– сумісність контейнерів забезпечує потенційну можливість усунення цілого класу програмних помилок, що викликаються незначними змінами робочого середовища – позбавляється обґрунтування древній аргумент розробника: «але це працює на моєму комп'ютері»;

– спрощена сутність контейнера означає, що розробники можуть одночасно запускати десятки контейнерів, що дає можливість імітації роботи промислової розподіленої системи.

Інженери з експлуатації можуть запустити на одному хості набагато більше контейнерів, ніж при використанні окремих віртуальних машин. Крім того, контейнери надають переваги кінцевим користувачам і розробникам без необхідності розгортання програми в хмарі. Користувачі можуть завантажувати і запускати складні додатки без багатогодинної метушні з конфігурацією і проблемами при установці і при цьому не турбуватися про будь-які зміни в їх локальних системах. У свою чергу, розробники подібних додатків можуть уникнути проблем, пов'язаних з відмінностями в конфігураціях призначених для користувача середовищ і з доступністю залежностей для цих додатків. І що більш важливо, існують принципові відмінності в цілях використання віртуальних машин і контейнерів – ***метою застосування віртуальної машини є повна емуляція***

чужорідного програмного (операційного) середовища, тоді як мета застосування контейнера – зробити додатки кросплатформенними і самодостатніми.

1.2. Порівняння контейнерів з віртуальними машинами

Незважаючи на те що контейнери і віртуальні машини на перший погляд здаються схожими, між ними існують важливі відмінності, які найпростіше продемонструвати на графічних схемах.

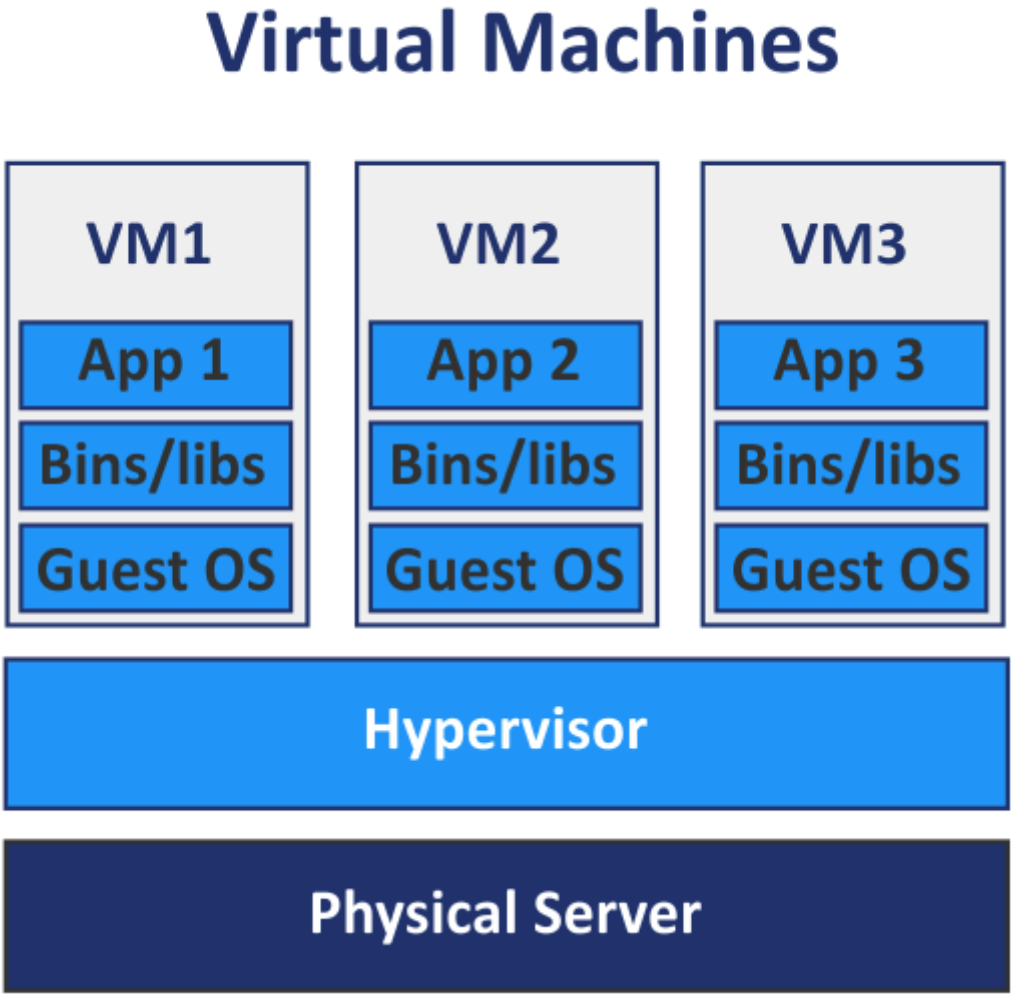


Рис. 1.1. Розгортання додатку з використанням віртуальних машин

На рисунку 1.1 представлено три додатки, що працюють в окремих віртуальних машинах на одному хості. Тут потрібен гіпервізор для створення і запуску віртуальних машин, керуючий доступом до хостової ОС і до апаратури, а також при необхідності інтерпретування системних викликів. Для кожної віртуальної машини необхідні повна копія ОС, що запускається і всі бібліотеки підтримки.

Гіпервізори можуть бути двох типів: тип 1 – такий як *Virtualbox* або *VMWare Workstation*, який працює поверх основної ОС, типу 2 – такий як *Xen*, що працює безпосередньо на «голому залізі».

***Ядро* (*kernel*)** є головним компонентом будь-якої ОС і відповідає за надання додаткам доступу до найважливіших системних функцій (ресурсів), пов'язаних з оперативною пам'яттю, процесором і доступом до різних пристроїв. Повноцінна ОС складається з ядра і різноманітних системних програм, таких як програма ініціалізації системи, компілятори і віконні менеджери та ін.

На противагу описаній схемі, на рис. 1.2 показано, як ті ж самі три додатки можуть працювати в системі з використанням технології контейнеризації.

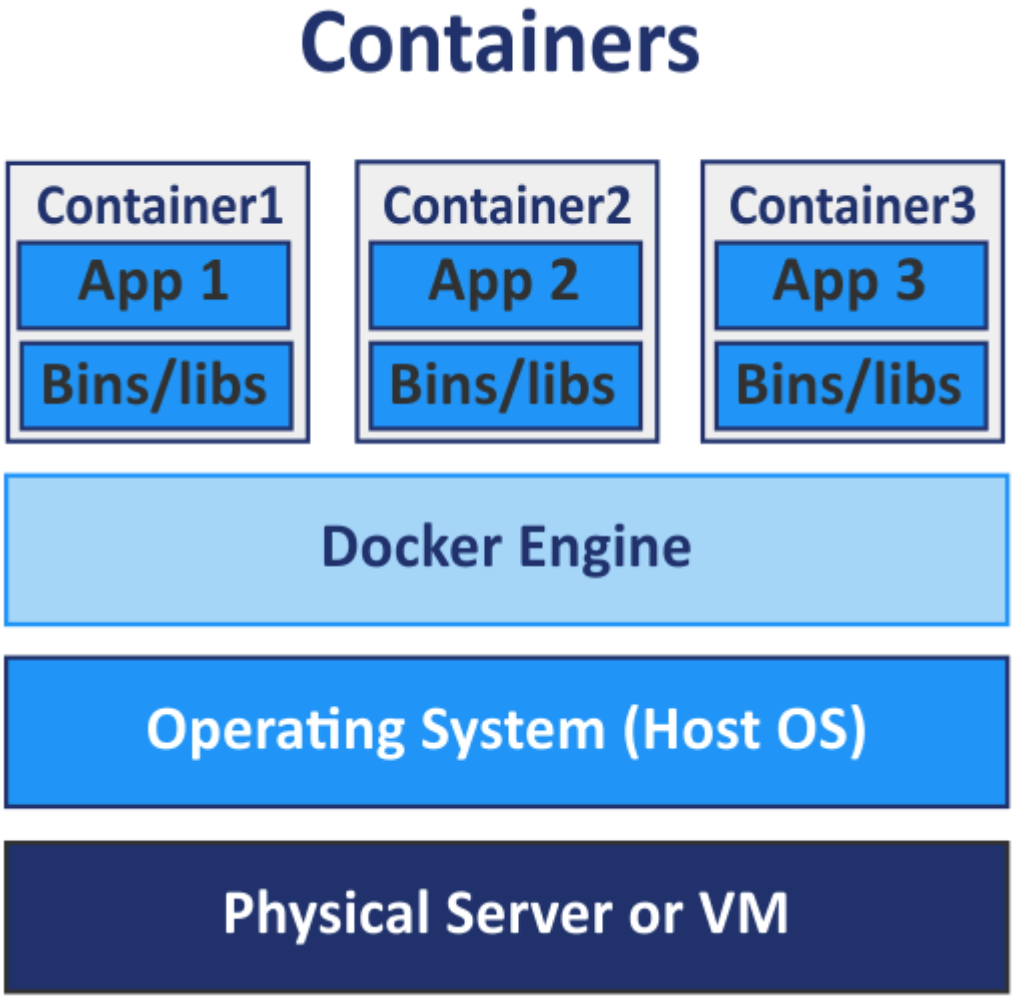


Рис. 1.2. Розгортання додатку з використанням технології контейнеризації

На відміну від віртуальних машин, ядро хоста спільно використовується (розділяється) працюючими контейнерами. Це означає, що контейнери завжди обмежуються використанням того ж ядра, яке функціонує на хості. Внутрішній механізм контейнера відповідає за пуск і зупинку контейнерів так само, як гіпервізор у віртуальній машині. Проте процеси всередині контейнерів рівнозначні власним процесам ОС хоста і не

вимагають додаткових накладних витрат, пов'язаних з роботою гіпервізора. Як віртуальні машини, так і контейнери можна використовувати для ізоляції додатків один від одного при роботі на одному хості. Додаткова ступінь ізоляції в віртуальних машинах забезпечується гіпервізором, і віртуальні машини є багаторазово перевірені в реальних умовах. Контейнери являють собою порівняно нову технологію, і багато організацій не цілком довіряють методиці ізоляції функціональних компонентів у контейнерах, поки не побачать на власні очі вагомих доказів переваг контейнерів. З цієї причини часто зустрічаються гібридні системи, в яких контейнери працюють усередині віртуальних машин для поєднання переваг обох технологій.

1.3. Технологія контейнеризації Docker

Філософію *Docker* часто описують за допомогою метафори “доставки вантажів в контейнерах”, яка цілком очевидно пояснює походження імені *Docker*. Нижче наводиться найбільш типовий виклад цієї філософії.

При транспортуванні вантажів використовуються різноманітні засоби, включаючи трейлери, автонавантажувачі, крани, поїзди і кораблі. Ці засоби повинні мати можливості роботи з широким спектром вантажів різних розмірів і дотримуватися численних спеціальних вимог (наприклад, при транспортуванні упаковок кави, бочок з небезпечними хімікатами, коробок з електронними товарами, парків дорогих автомобілів і стелажів із замороженими м’ясними продуктами). Протягом багатьох років це був важкий і дорогий процес, що вимагав великих трудовитрат багатьох людей, в тому числі і портових докерів, для навантаження і вивантаження вручну різних предметів у кожному транзитному пункті.

Корінний переворот в транспортній промисловості справила поява універсальних вантажних контейнерів. Ці контейнери мали стандартні розміри і були спеціально спроектовані таким чином, щоб для їх переміщення між різними видами транспортних засобів був потрібний мінімум ручної праці. Всі види вантажного транспорту створювалися з урахуванням характеристик цих контейнерів – від автонавантажувачів і кранів до вантажівок, поїздів і кораблів. Контейнери-рефрижератори та ізольовані контейнери призначені для перевезення вантажів з жорстко заданим температурним режимом, наприклад, деяких продуктів харчування і фармацевтичних товарів. Крім того, переваги стандартизації поширилися і на інші допоміжні системи, такі як система маркування вантажів і система герметизації і опечатування контейнерів.

Це означає, що відповідальність за вміст контейнерів повністю перекладається на виробників товарів, що транспортуються, а працівники транспорту можуть повністю зосередитися на перевезенні та зберіганні самих контейнерів.

Основна мета програмного середовища Docker – перенести переваги стандартизації контейнерів в область інформаційних технологій.

В останні роки програмні системи відрізняються вражаючою різноманітністю. Пройшли часи технології *LAMP*, реалізованої на одному комп'ютері. Типова сучасна система може складатися з фреймворків *JavaScript*, баз даних *MySQL*, черг повідомлень, прикладних програмних інтерфейсів *REST* і внутрішніх компонентів, написаних різними мовами програмування. Такий стек повинен частково або повністю працювати на різноманітній апаратурі – від особистого комп'ютера або ноутбука розробника і невеликого тестового кластера, до великого промислового вузла провайдера хмарних сервісів. Всі ці середовища відрізняються різноманітністю, використовують різні операційні системи з різними версіями бібліотек і працюють на різній апаратурі.

Отже, проблема точно така ж, як і в транспортній промисловості – для переміщення коду додатків між різними середовищами потрібно значний обсяг ручної праці. Подібно до того, як універсальні контейнери спрощують перевезення вантажів, контейнери *Docker* спрощують переміщення (перенесення) програмних додатків.

Розробники можуть повністю зосередитися на створенні програми, на проведенні циклу тестування і на вводі додатків в експлуатацію, не турбуючись про відмінності в програмних середовищах і забезпеченні необхідних залежностей.

Інженери з експлуатації можуть приділити всю увагу специфічним питанням забезпечення роботи контейнерів, таким як розподіл ресурсів, запуск і зупинка контейнерів, переміщення контейнерів між серверами.

Контейнери є старою теоретичною концепцією. Ще кілька десятків років тому в *Unix*-системах існувала команда *chroot*, що забезпечує найпростішу форму ізоляції частини файлової системи. З 1998 року в ОС *FreeBSD* з'явилася утиліта *jail*, яка поширює ізоляційні можливості chroot на процеси. У 2001 році реалізація *Solaris Zones* забезпечувала повну технологію контейнеризації, але її застосування було обмежено тільки *ОС Solaris*. У тому ж 2001 році компанія *Parrallels Inc* (надалі *SWsoft*) випустила комерційну версію технології контейнерів *Virtuozzo* для ОС *Linux*, а в 2005 році відкрила вихідні коди ядра цієї технології під ім'ям *OpenVZ1*. Потім компанія *Google* почала розробку *CGroups* для ядра ОС *Linux* і приступила до переміщення своєї інфраструктури в контейнери.

Проект *Linux Containers (LXC*) був створений в 2008 році і незабаром об'єднав *CGroups*, простір імен ядра, технологію *chroot* і деякі інші технології, щоб надати повністю завершене рішення щодо забезпечення контейнеризації. Нарешті, в 2013 році *Docker* став останнім штрихом в загальній картині стану контейнеризації і ця технологія по праву зайняла своє місце як одне з головних напрямків розвитку IT-індустрії.

В основу *Docker* була закладена існуюча технологія *Linux*-контейнерів з різноманітними обгортками і розширеннями, в основному використовуючи переносимі образи і зручний для користувача інтерфейс –

для створення повністю готового до застосування рішення, що забезпечує створення і поширення контейнерів.

Платформа *Docker* складається з двох окремих компонентів: ***Docker Engine*** – механізму, що відповідає за створення і функціонування контейнерів, *і* ***Docker Hub*** – хмарного сервісу для зберігання і поширення контейнерів.

Механізм *Docker Engine* (рис.1.3) надає ефективний і зручний інтерфейс для запуску контейнерів. Він складається з:

- серверу, що працює у фоновому режимі (як демон);
- REST API, що використовують програми для взаємодії з сервером;
- інтерфейсу командного рядка (CLI) клієнт.

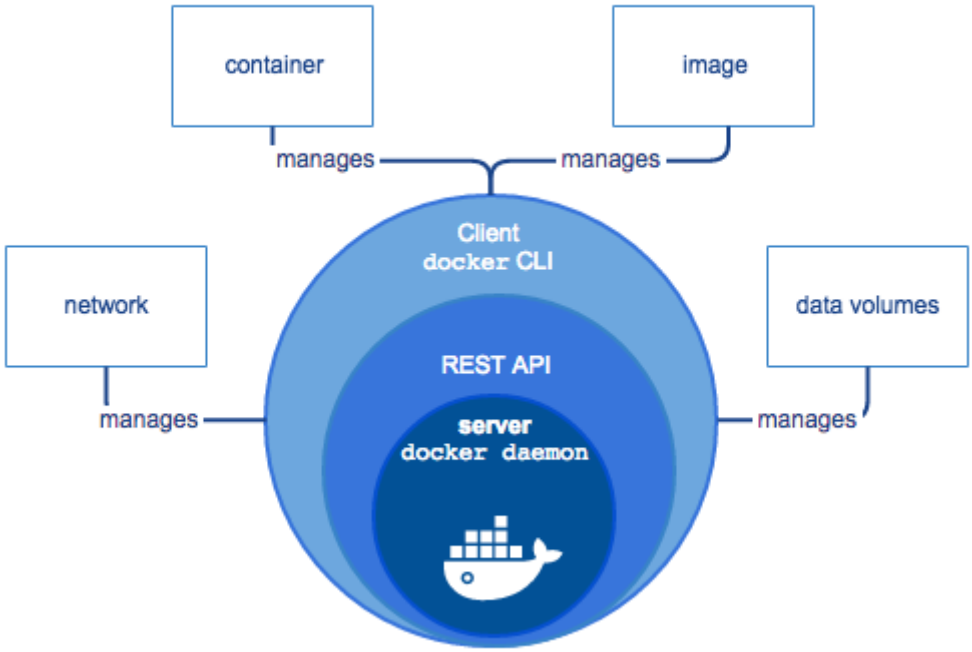


Рис. 1.3. Docker engine

До цього для запуску контейнерів, що використовують таку технологію, як, наприклад, *LXC*, були потрібні неабиякий запас спеціальних знань в цій області і великий обсяг ручної роботи.

Docker Hub надає величезну кількість образів контейнерів з відкритим доступом для завантаження, дозволяючи користувачам швидко почати роботу з ними і уникнути рутинної роботи, раніше вже виконаної іншими людьми. Розробники, інженери, тестувальники мають можливість безкоштовно завантажити так званий образ на базі якого розгортається контейнер (базу данних, вебсервер і тп). Завдяки чому, мають змогу завантажувати та ділитися контейнерами на базі яких розгорнуто додатки. (рис.1.4)

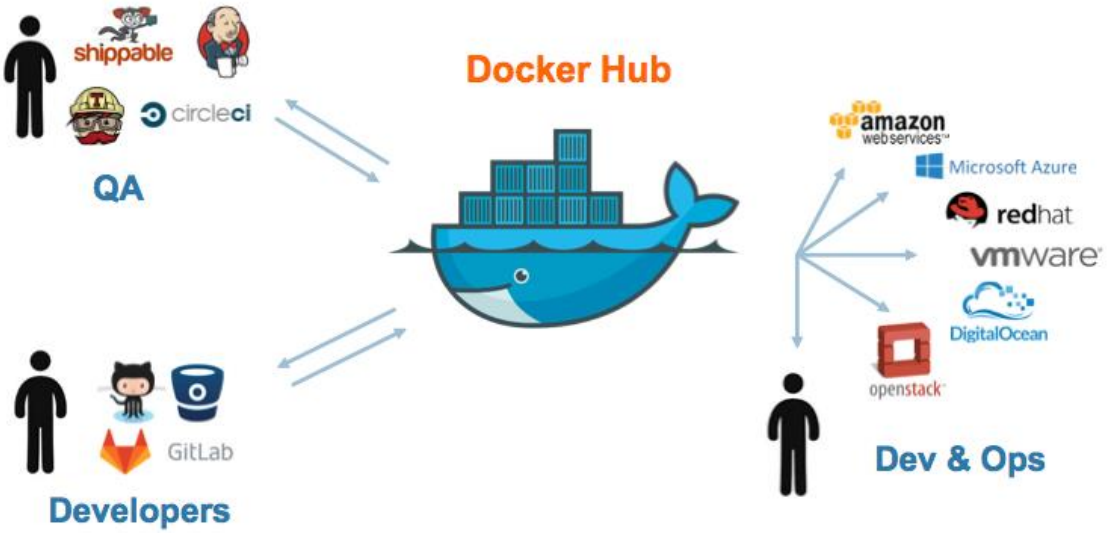


Рис. 1.4. Принцип роботи Docker Hub

Трохи пізніше були розроблені інструментальні засоби для *Docker*: ***Swarm*** – менеджер кластерів, ***Kitematic*** – графічний користувальницький інтерфейс для роботи з контейнерами і ***Machine*** – утиліта командного рядка для підтримки роботи *Docker*-хостів. З огляду на відкритість вихідних кодів *Docker Engine,* стало можливим створення великої спільноти прихильників технології *Docker* і її постійне зростання, у результаті чого користувачі надають один одному допомогу в усуненні помилок і внесення удосконалень.

Швидке зростання популярності цієї технології свідчить про те, що *Docker* дійсно стає стандартом де-факто, і це вже призвело до розробки незалежних стандартів для функціональності, часу виконання і формату контейнерів. У 2015 році найвищою точкою розвитку стало створення *Open Container Initiative*, «керуючої структури», підтримуваної *Docker, Microsoft, CoreOS* і багатьма іншими відомими організаціями, метою якої є розробка промислового стандарту. Основою для розробки служать формат і функціональні форми часу виконання *Docker*-контейнера.

Темпи зростання застосування контейнерів найбільшою мірою забезпечили розробники, які з самого початку надали інструментальні засоби для ефективного використання контейнерів. Мінімальний час від початку розробки до введення в експлуатацію *Docker*-контейнерів є надзвичайно важливою перевагою для розробників, яким необхідні швидкі ітеративні цикли розробки з можливістю негайного перегляду результатів змін, внесених до початкового коду.

Кросплатформеність та ізольованість контейнерів сприяють спрощенню співпраці з іншими розробниками та інженерами з експлуатації: розробники можуть бути цілком упевнені в тому, що їх код буде працювати в будь-яких програмних середовищах, а інженери з експлуатації можуть зосередитися на організації та налагодженню роботи контейнерів на хостах, не турбуючись про те, який код виконується всередині контейнерів. Нововведення, внесені з застосуванням технології *Docker*, істотно змінили

спосіб розробки програмного забезпечення. Без *Docker* – контейнери, найімовірніше, ще довго залишалися б малопомітним напрямком у розвитку інформаційних технологій.

Реєстрація на сайті docker.com: створення облікового запису користувача та реєстрація на сайті ***https://www.docker.com*.**

1.4. Встановлення Docker на ОС Ubuntu Linux

Одним із способів встановлення *Docker* на *OS Ubuntu* є використання репозиторіїв та пакетного менеджера apt:

1. Оновлення переліку пакетів

$ sudo apt-get update ;

2. Встановлюємо необхідні пакети, які дозволяють apt використовувати пакети по HTTPS

$ sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-properties-common;

3. Додаємо в свою систему ключ GPG офіційного репозиторію Docker

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add – ;

4. Додавання репозиторію до системи

$ sudo add-apt-repository “deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable” ;

5. Встановлення *docker engine*

$ sudo apt-get update

$ sudo apt-get install docker-ce docker-ce-cli containerd.io

6. У випадку коректного виконання попередніх кроків можливо перевірити версію Docker:

$ docker –v

Виведення в терміналі повинно бути подібне до цього:

```
[node1] (local) root@192.168.0.8 ~
$ docker -v
Docker version 20.10.0, build 7287ab3
```

Рис. 1.5. Перевірка версії *Docker Engine* для *OS Ubuntu*

Офіційну документацію та інструкцію по встановленню Docker можна переглянути за посиланням: ***https://docs.docker.com/engine/install/ubuntu/***.

1.5. Встановлення Docker на ОС Windows

Для встановлення *Docker engine* на *OS Windows* необхідно перейти за посиланням ***(https://hub.docker.com/editions/community/docker-ce-desktop-windows/)*** та завантажити файл з розширенням *.exe (рис. 1.6)

Рис. 1.6. Завантаження *Docker Engine* для *OS Windows*

Після завантаження відкрийте файл з розширенням *.exe, вкажіть папку для встановлення та інші необхідні параметри.

Після успішного завершення роботи інсталятора, відкрийте панель додатків (рис. 1.7) та запустіть додаток *Docker Engine.*

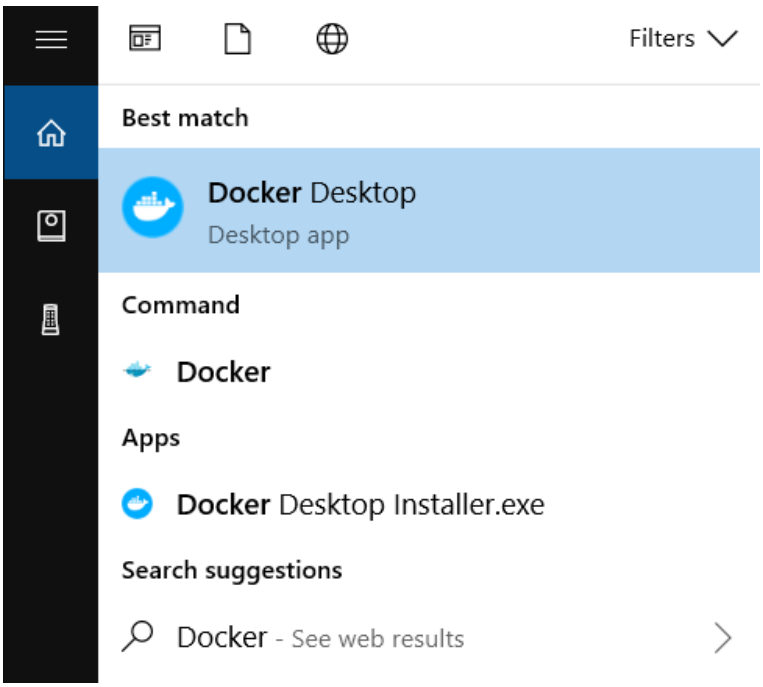


Рис. 1.7. Панель додатків *Windows*

Після запуску додатку з’явиться характерний значок (Рис. 1.8).

Рис. 1.8. Значок працюючого *Docker Engine*

1.6. Створення образу контейнера

Образ* (*image*)** – шаблон, який використовується для створення контейнерів. Являє собою зліпок файлової системи, в якому розташований код програми та його оточення (наприклад, ос *Ubuntu* з встановленим сервером *MySQL*). Перелік доступних образів можливо переглянути на сайті *Docker Hub* (https://hub.docker.com/***) (рис. 1.9, рис. 1.10).

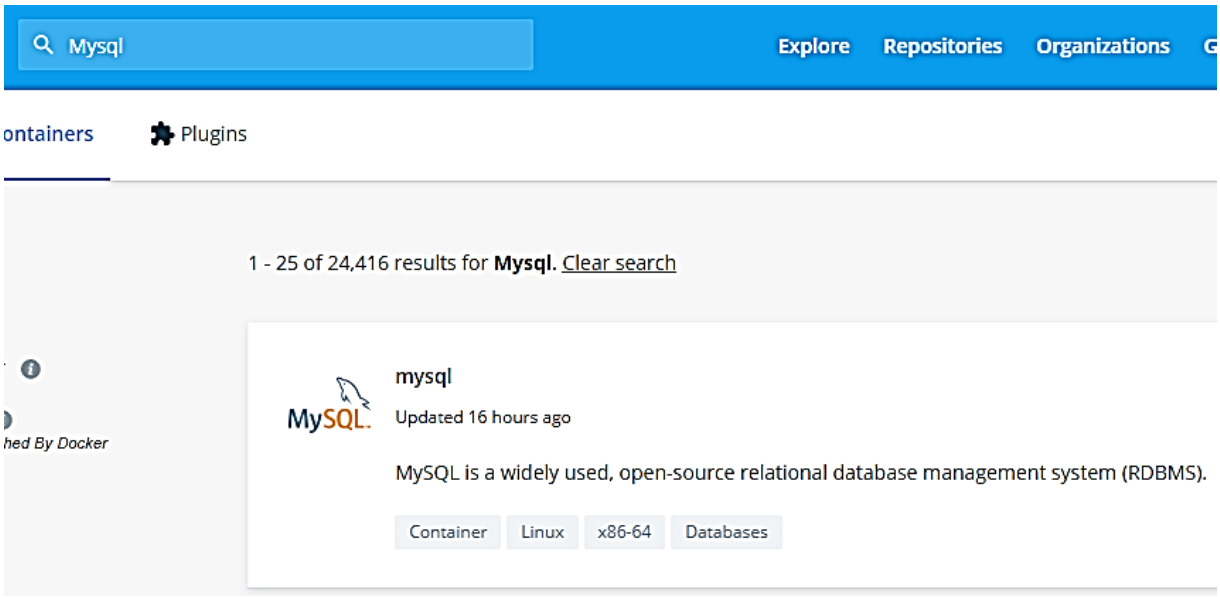


Рис. 1.9. Результати пошуку образу *MySQL* на сайті *Docker Hub*

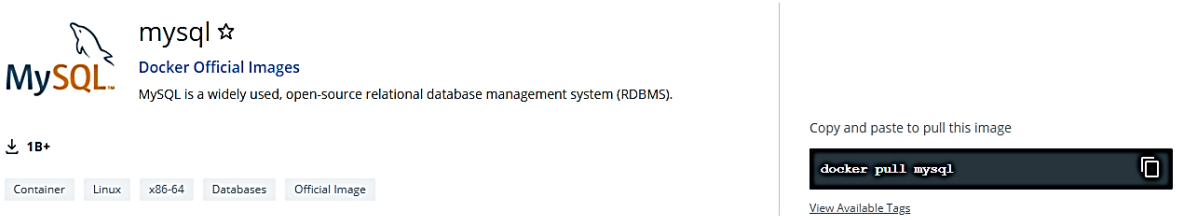


Рис. 1.10. Результати пошуку образу *MySQL* на сайті *Docker Hub*

Для того щоб завантажити цей образ необхідно ввести команду (рис. 1.11):

$ docker pull mysql

```
$ docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql
a076a628af6f: Pull complete
f6c208f3f991: Pull complete
88a9455a9165: Pull complete
406c9b8427c6: Pull complete
7c88599c0b25: Pull complete
25b5c6debdaf: Pull complete
43a5816f1617: Pull complete
69dd1fbf9190: Pull complete
5346a60dcee8: Pull complete
ef28da371fc9: Pull complete
fd04d935b852: Pull complete
050c49742ea2: Pull complete
Digest: sha256:0fd2898dc1c946b34dceaccc3b80d38b1049285c1dab70df7480de62265d6213
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest
```

Рис. 1.11. Результати завантаження образу *MySQL*

За допомогою команди ***$docker images*** можливо переглянути всі образи які зберігаються на машині (рис. 1.12).

```
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
mysql        latest    d4c3cafb11d5   30 hours ago    545MB
```

Рис. 1.12. Перегляд образів що зберігаються на машині

1.7. Розгортання першого контейнера

Для запуску контейнера необхідно виконати команду ***$docker run –h hostname-of-container –i name-of-image –t*** в результаті цього буде створено контейнер на базі *OS Debian*. Флаг ***–t*** означає що після створення контейнера буде здійснено вхід до нього. Вийти з середовища контейнера можливо за допомогою команди ***$exit***. Контейнер існує до тих пір поки запущений процес. В цьому випадку поки ви не вийшли з контейнера то він запущений. Переглянути список запущених контейнерів на хості можливо за допомогою команди ***$docker ps***, але кожен раз коли ви виконуєте команду ***$docker run*** то створюється новий контейнер для того щоб переглянути список усіх контенерів, що існують на машині, необхідно ввести команду ***$docker ps –a*** (рис. 1.13).

```
[node1] (local) root@192.168.0.8 ~
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
[node1] (local) root@192.168.0.8 ~
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS                          PORTS     NAMES
ea9196316ec1   debian    "bash"                   10 seconds ago       Exited (0) 6 seconds ago                  elegant_menin
bc4afe1049f7   debian    "bash"                   About a minute ago   Exited (0) About a minute ago             condescending
399847a56cc7   debian    "-h i-love-k-21"         About a minute ago   Created                                   musing_kepler
60c27b07e5ca   debian    "bash"                   18 minutes ago       Exited (0) 10 minutes ago                 clever_mclean
edcc52945001   debian    "bash"                   21 minutes ago       Exited (127) 20 minutes ago               amazing_albat
fbd00b353696   mysql     "docker-entrypoint.s…"   25 minutes ago       Exited (1) 25 minutes ago                 fervent_buck
[node1] (local) root@192.168.0.8 ~
```

Рис. 1.13. Перегляд контейнерів що зберігаються на машині

У кожного контейнера є свій ідентифікатор – ***container ID***, це унікальне значення. Дізнатися інформацію про контейнер можливо з використанням команди ***$docker inspect container-ID.***

```
$ docker inspect 60c27b07e5ca
[
    {
        "Id": "60c27b07e5ca6de29d79255b6dfd674b5346f896ffe84e553632984710b68465",
        "Created": "2021-01-13T16:00:37.202658266Z",
        "Path": "bash",
        "Args": [],
        "State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
```

Рис. 1.14. Перегляд інформації про контейнер

Запустити контейнер можливо за допомогою команди ***$docker start container-id,*** а зупинити, відповідно ***$docker stop container-id*** (рис. 1.15).

```
$ docker start 60c27b07e5ca
60c27b07e5ca
[node1] (local) root@192.168.0.8 ~
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS         PORTS     NAMES
60c27b07e5ca   debian    "bash"    27 minutes ago   Up 3 seconds             clever_mclean
[node1] (local) root@192.168.0.8 ~
$ docker stop 60c27b07e5ca
60c27b07e5ca
[node1] (local) root@192.168.0.8 ~
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
[node1] (local) root@192.168.0.8 ~
```

Рис. 1.15. Запуск та зупинка контейнера

Видалити контейнер можливо за допомогою команди ***$docker rm container-id***. Видалити існуючі образи можливо за допомогою команди ***$docker rmi name-of-image,*** якщо image використовується якимось контейнером то спочатку потрібно зупинити контейнер командою ***$docker stop container-id.***

1.8. Демонстрація курсів *Play with docker*

Це повністю безкоштовний курс, який надає можливість опанувати базові навички роботи з Docker. Для його проходження необхідно мати обілковий запис, вказаний вкінці розділу 1.3 та виконати наступні дії:

– перехід на ***https://www.docker.com/101-tutorial;***
– перехід за посиланням (рис. 1.16);

Рис. 1.16. Початок курсу *Play with Docker*

– після відкриття посилання, зображеного рис. 1.16 в робочому полі середовища для навчання потрібно ввести команду: ***docker run -dp 80:80 docker/getting-started:pwd;***

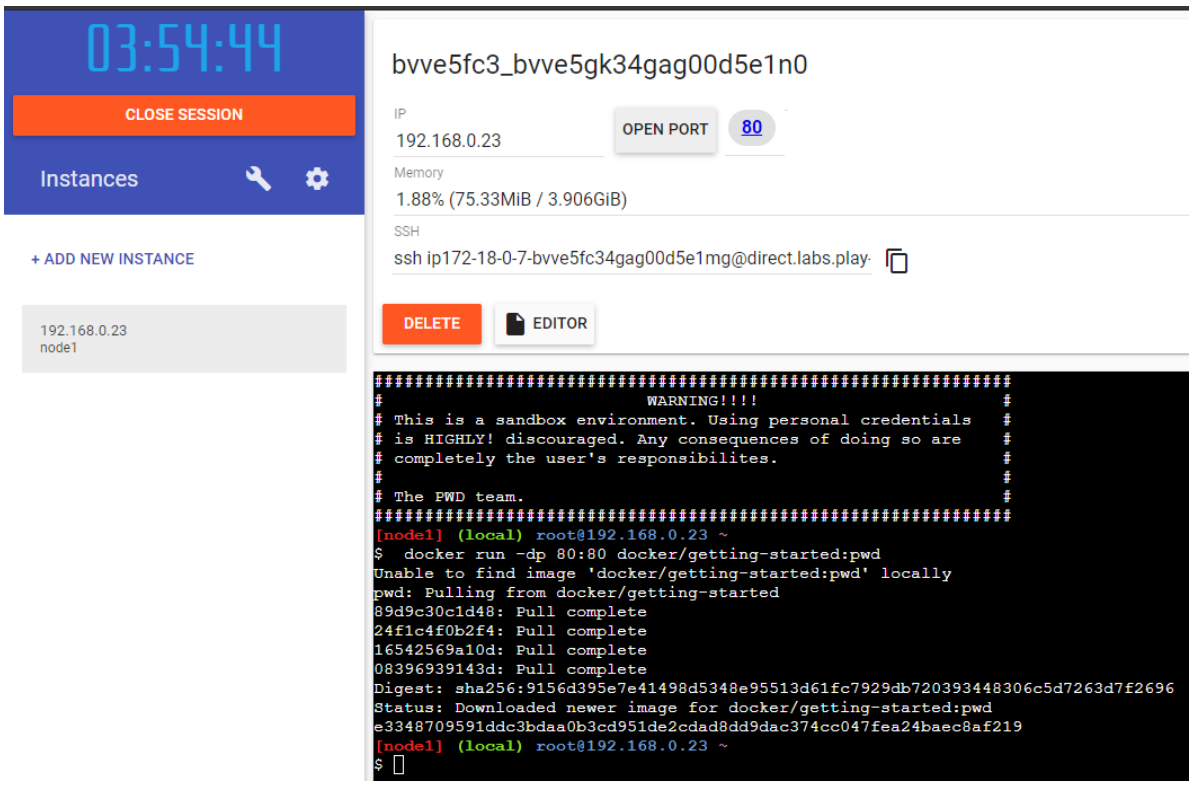


Рис. 1.17. Середовище для навчання

– натиснути на кнопку **Open Port** ввести цифру **80** і натиснути **OK** (рис. 1.18).

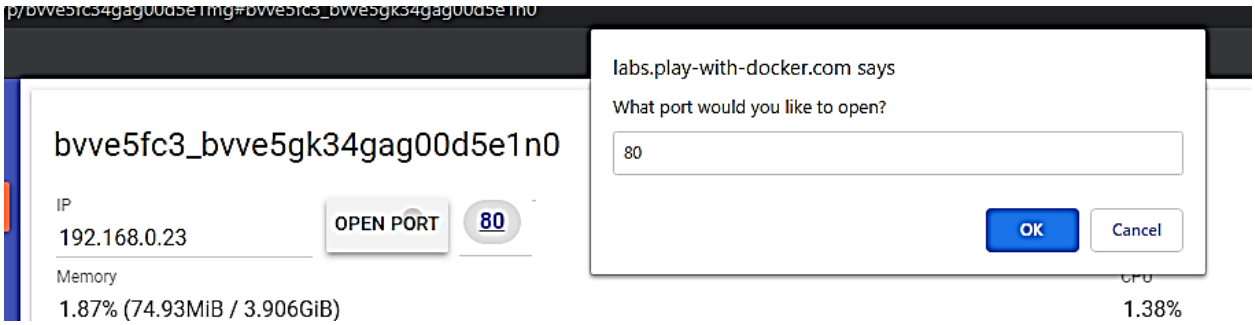


Рис. 1.18 Відкриття вікна із лабораторними роботами

– у випадку успішного виконання попередніх кроків ви потрапите на сторінку, яка містить завдання, що допоможуть вам навчитися користуватися ***Docker*** (рис. 1.19).

–

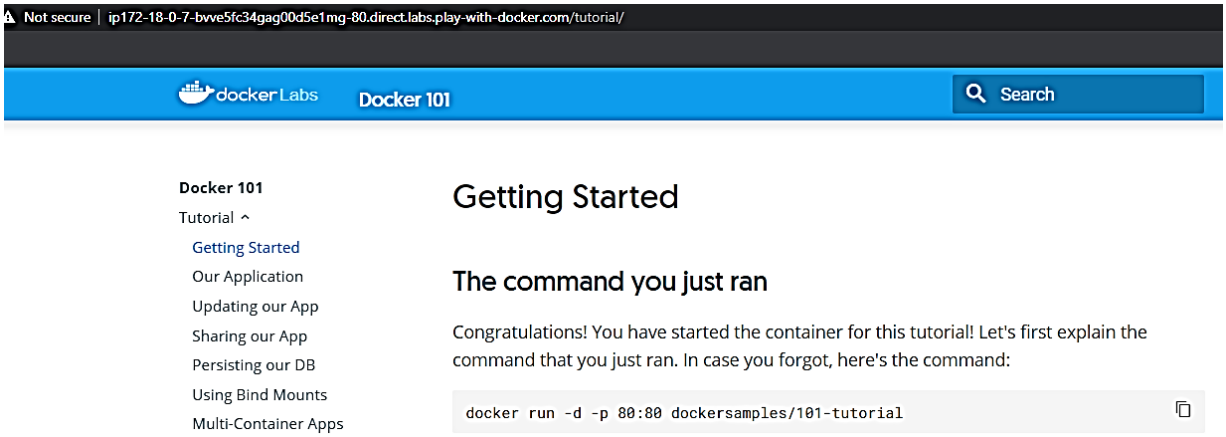


Рис. 1.19. Вікно із лабораторними роботами

1.9. Огляд Docker Hub

Docker Hub являє собою хмарний сервіс для організації спільної роботи, автоматизації робочого процесу, створення, поширення і запуску адаптованих для *Docker* застосунків. По суті *Docker Hub* надає набір сервісів, таких як поширення образу контейнера, управління змінами, організація взаємодії між користувачами і розробниками, супровід життєвого циклу, інтеграція зі сторонніми службами. Модель монетизації сервісу *Docker Hub* аналогічна *GitHub* – робота над публічними проектами безкоштовна і лише при необхідності використання приватних репозиторіїв стягується оплата.

Основні компоненти *Docker Hub*:

– інтегрована консоль для управління користувачами, групами, контейнерами, репозиторіями і робочими процесами;

– реєстр, що надає понад 14 тисяч застосунків, які можна використовувати як цеглини для створення власних додатків;

– інструменти для спільної роботи, що дозволяють користувачам обмінюватися своїми додатками через публічні та приватні репозиторії і запрошувати інших людей взяти участь в розробці;

– сервіс *Webhooks*, що дозволяє користувачеві забезпечити управління з сторонніх систем і автоматизувати виконання типових робіт через *RESTful API*;

– *Docker Hub API*, що включає сервіс автентифікації і засоби інтеграції з зовнішніми службами.

Команди для роботи з *Docker Hub:*
$docker login **–** авторизуватися на *Docker Hub;*
$docker serach **–** пошук образів на *Docker Hub;*
$docker pull **–** завантажити образ з *Docker Hub;*
$docker push **–** завантажити образ на *Docker Hub.*

Особливості *Docker Hub:*

– репозиторії образів – це дозволяє знаходити та завантажувати образи контейнерів;

– дозволяє створювати робочі групи і видавати репозиторії як приватні, які доступні для використання тільки у певній організації;

– дозволяє інтеграцію з репозиторіями вихідного коду, такими як *GitHub* і *BitBucket*.

Огляд та пошук образу *docker*-контейнера на *Docker Hub* виглядає таким чином: (рис. 1.20, рис. 1.21, рис. 1.22)

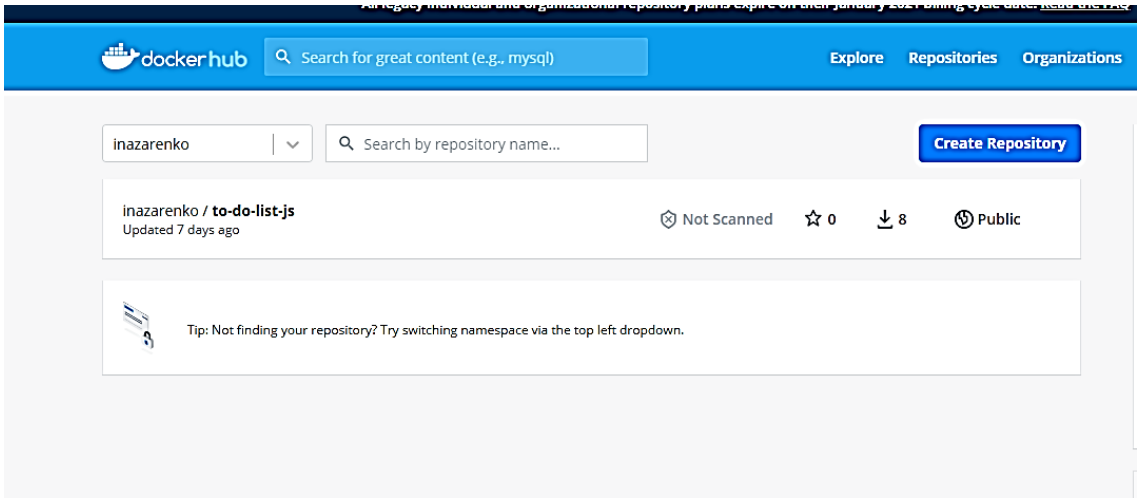


Рис. 1.20 Початкова сторінка *DockerHub*

На початковій сторінці доступні всі репозиторії створені користувачем (рис. 1.20).

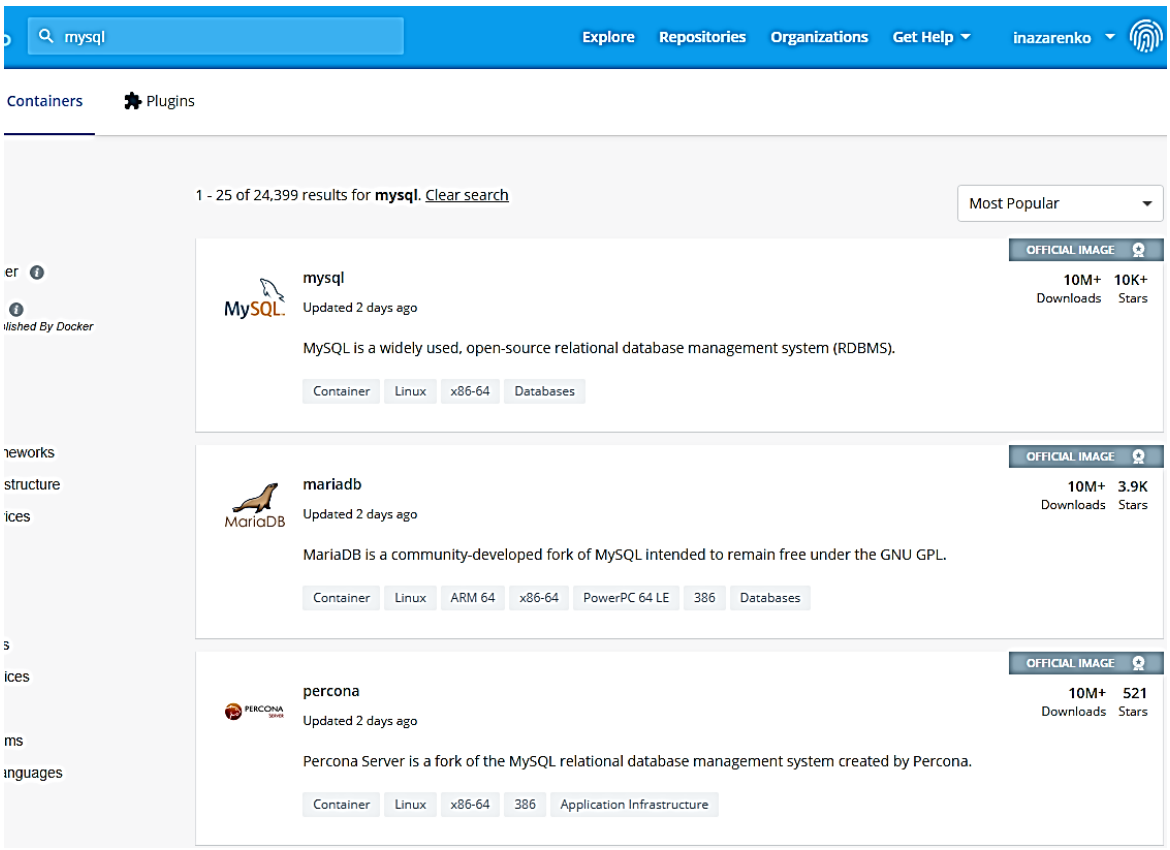


Рис. 1.21. Результати пошуку образу *MySQL* на сайті *Docker Hub*

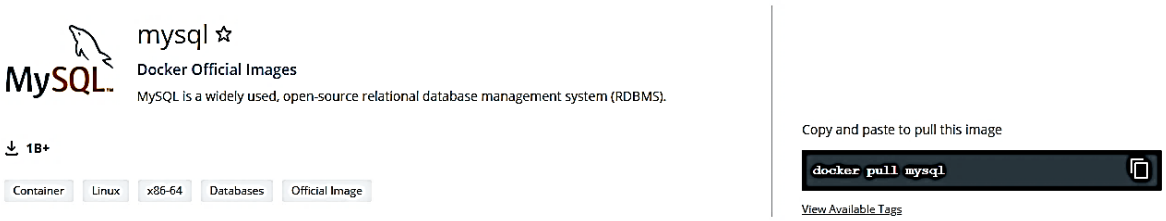


Рис. 1.22. Результати пошуку образу *MySQL* на сайті *Docker Hub*

1.9.1. Основні функції Docker Hub

Найпростіше і очевидне рішення, яке забезпечує загальний доступ до розроблених образів – використання *Docker Hub*. *Docker Hub* – реєстр, що працює в режимі онлайн і підтримується компанією *Docker Inc*. Цей реєстр надає вільні репозиторії для відкритих образів, але користувачі можуть оплатити закриті, захищені репозиторії. *Docker Hub* – це не єдиний варіант при виборі сховища приватних закритих репозиторіїв у хмарному середовищі. Оновним конкурентом є реєстр *quay.io*, що пропонує трохи більше розширені функціональні можливості, в порівнянні з *Docker Hub*, за цілком прийнятними цінами.

Доступ до реєстру *Docker Hub* можна отримати як з командного рядка, так і через сайт. Пошук існуючих образів виконується за допомогою

спеціальної команди пошуку *Docker* або безпосередньо на сайті ***https://hub.docker.com/*** .

Для зберігання образів застосовується ієрархічна система. При цьому використовуються такі терміни:

– **реєстр (*registry*)** – сервіс, який відповідає за зберігання і поширення образів. За замовчуванням використовується реєстр *Docker Hub*;

– **репозиторій (*repository*)** – набір взаємопов'язаних образів (зазвичай представляють різні версії однієї програми або сервісу);

– **тег (*tag*)** – алфавітно-цифровий ідентифікатор, який присвоюється образам, директоріям сховища (наприклад, 14.04 або *stable*).

Таким чином, команда ***docker pull amouat / revealjs: latest*** завантажить образ з тегом *latest* в репозиторій *amouat / revealjs* з реєстру *Docker Hub*. Для вивантаження в реєстр нашого image необхідно зареєструватися (створити обліковий запис) в реєстрі *Docker Hub* (в онлайн-режимі на сайті або за допомогою команди *docker login*). Після цього досить зв'язати образ з відповідним репозиторієм і виконати команду ***docker push*** для вивантаження поміченого образу в *Docker Hub*. Але необхідно додати в *Dockerfile* інструкцію ***MAINTAINER***, яка просто визначає інформацію, що дозволяє зв'язатися з автором даного *image*:

MAINTAINER Cadet <cadet@gmail.com>

При поширенні образів дуже важливо використовувати точні і змістовні імена і теги. Імена і теги присвоюються при створенні образу або встановлюються спеціальною командою ***$docker tag***

$ docker build -t my-app – визначення імені *image my-app.*

$ docker tag my-app user/my-app:production – *Image my-app* переіменовується на my-*repository/my-app*, який посилається на ім'я користувача *user* в репозиторії *Docker Hub* з тегом *production.*

За замовчуванням при створенні *image* присвоюється тег *latest*, але не слід занадто часто застосовувати тег *latest*. *Docker* використовує цей тег за замовчуванням, якщо явно не заданий будь-який інший тег, але *latest* не містить корисної інформації. У багатьох репозиторіях цей тег служить позначенням найсвіжішої стабільної версії образу, але це зовсім не обов'язково. Образи з тегом *latest*, як і всі інші образи, які не будуть оновлюватися автоматично при вивантаженні нової версії до реєстру, залишають необхідність явного виконання команди *docker pull* для отримання оновлених версій. Якщо в команді *docker run* або *docker pull* вказано ім'я образу без тега, то *Docker* використовує образ з тегом *latest*, а при відсутності такого способу повідомить про помилку. Тег *latest* занадто

часто вводить в оману користувачів, тому рекомендується повністю відмовитися від його застосування, особливо у відкритих масових репозиторіях.

$docker push user/my-app:production – вивантаження *image* з використанням щойно створеного псевдоніма. Якщо зазначений репозиторій не існував раніше, то він створюється, і виконується вивантаження образу з урахуванням заданого тега.

Після цього image ***my-app*** загальнодоступний, та будь-який користувач зможе завантажити його з командою ***docker pull***. При переході на сайт Docker Hub можливо найти свій репозиторій по URL відповідно до схеми ***https://hub.docker.com/repository/docker/user/my-app .***

Якщо потрібно оновити репозиторій, то досить повторити виконання команд установки тега і вивантаження образу для необхідного *image*. При використанні існуючого тега попередній *image* буде перезаписаний.

1.10. Мережеві налаштування та томи у Docker

Мережева підсистема *Docker* підключається, використовуючи драйвери. За замовчуванням існує кілька драйверів які забезпечують основні функції мережі:

bridge: міст – це мережевий драйвер за замовчуванням. Мережевий міст використовується, коли ваші додатки запускаються в автономних контейнерах, які повинні взаємодіяти між собою (наочний приклад *Nginx* + *MySQL*);

host: хост – це мережевий драйвер для автономних контейнерів (віддалена мережева ізоляція між контейнером і *Docker* хостом). Драйвер доступний тільки для *docker-swarm* з підтримкою *Docker* 17.06 і вище;

overlay/overlay2: оверлей (накладена мережа) – це мережевий драйвер для з'єднання кількох демонів *Docker* між собою і які дозволяють *docker-swarm* службам взаємодіяти одна з одною. Ви також можете використовувати оверлейні мережі для сприяння контактам між *docker-swarm* і автономним контейнером або між двома окремими контейнерами на різних *Docker* демонах. Ця стратегія усуває необхідність виконання маршрутизації на рівні ОС між цими контейнерами;

macvlan: маквлан – це мережевий драйвер, який дозволяє призначати *MAC*-адресу контейнеру, роблячи його відображуваним як фізичний пристрій у вашій мережі. *Docker* демон направляє трафік на контейнери за їх *MAC*-адресами. Використання *macvlan* драйвера іноді є кращим вибором при роботі з застарілими додатками, які очікують, що вони будуть безпосередньо підключені до фізичної мережі;

none – це мережевий драйвер, який вміє відключати всю мережу для контейнерів. Зазвичай використовується в поєднанні з призначеним для користувача мережевим драйвером;

network plugins: дозволяє встановити і використовувати сторонні мережеві плагіни з Docker контейнерами. Ці плагіни доступні в Docker Store або у сторонніх постачальників послуг.

За замовчуванням докер створює одну мережу (типу *Bridge*), яка співвідноситься з інтефейсом *docker0*, всі контейнери, що запускаються через *docker run*, автоматично додаються в цю мережу. Дану мережу не можна видалити або змінити. Найбільш використовуваними опціями для даної команди є:

– *subnet* – дозволяє задати необхідну мережу;

– *ip-range* – дозволяє задати діапазон динамічних адрес, які видаються контейнерам при підключенні.

Де і що краще використовувати?

Міст (*bridge*) найкраще використовувати для зв'язку декількох контейнерів на одному і тому ж *Docke*r-хості. Можна використовувати *docker-compose* і обрати дану мережу для такої зв'язки.

Хост (*host*) найкраще використовувати, коли мережевий стек не повинен бути ізольований від хоста *Docker*, але потрібно, щоб інші аспекти контейнера були ізольовані.

Оверлейну мережу (*overlay* /*overlay2*) або накладення мереж найкраще використовувати, коли потрібні контейнери, що працюють на різних *Docker*-хостах для зв'язку, або коли кілька додатків працюють разом, використовуючи *docker-swarm*.

Маквлан (*macvlan*) мережі найкраще використовувати, коли потрібен перехід з *VM* / на виділені хости, щоб контейнери виглядали як фізичні хости у певній мережі, кожен з унікальною *MAC*-адресою. Сторонні мережеві плагіни дозволяють інтегрувати *Docker* зі спеціалізованими мережевими стеками.

Docker volume

У контейнерах *Docker* організувати роботу з тимчасовими даними можна двома способами. За замовчуванням файли, створені додатком, що працює в контейнері, зберігаються в шарі контейнера, що підтримує запис. Для того щоб цей механізм працював, нічого спеціально налаштовувати не потрібно. Виходить дешево і сердито. Додатком досить просто зберегти дані і продовжити займатися своїми справами. ***Однак після того як контейнер перестане існувати, зникнуть і дані, збережені таким ось нехитрим способом.***

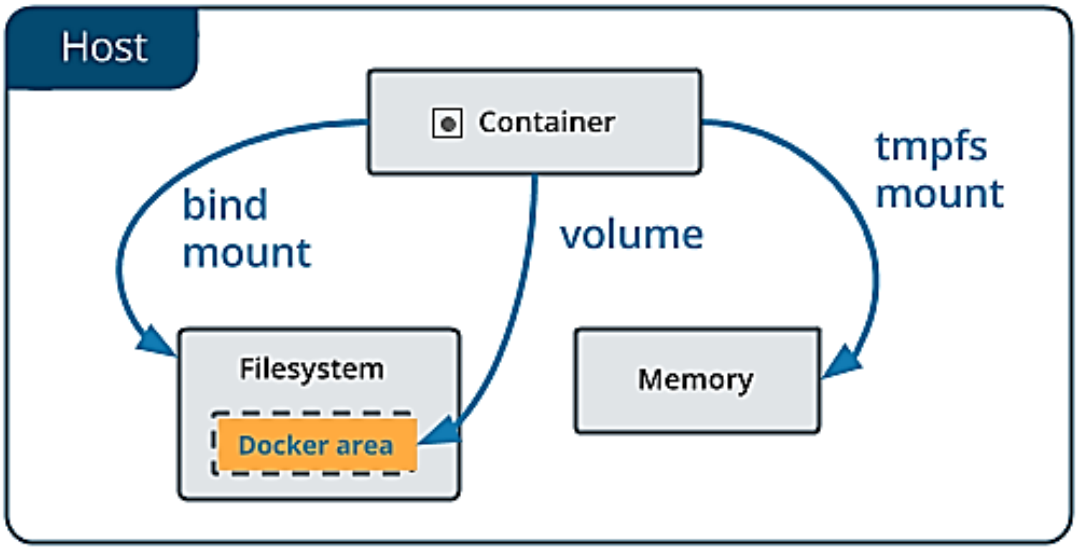


Рис. 1.23. Типи зберігання інформації в *Docker*

Для зберігання тимчасових файлів у *Docker* можна скористатися ще одним рішенням, відповідним для тих випадків, коли потрібно більш високий рівень продуктивності, в порівнянні з тим, який можна досягти при використанні стандартного механізму тимчасового зберігання даних. Якщо не потрібно, щоб дані зберігалися довше, ніж існує контейнер, можете підключити до контейнера ***tmpfs*** – тимчасове сховище інформації, яке використовує оперативну пам'ять хоста. Це дозволить прискорити виконання операцій по запису і читання даних.

Часто буває так, що дані потрібно зберігати і після того, як контейнер припинить існувати. Для цього знадобляться механізми постійного зберігання даних.

Існують два способи, що дозволяють зробити термін життя даних довше терміну життя контейнера. Один із способів полягає у використанні технології ***bind mount***. При такому підході до контейнера можна примонтувати, наприклад, реально існуючу папку. Працювати з даними, що зберігаються в такій папці, зможуть і процеси, що знаходяться за межами *Docker*.

Мінуси використання технології ***bind mount*** полягають в тому, що її використання ускладнює резервне копіювання даних, міграцію даних, спільне використання даних декількома контейнерами. Набагато краще для постійного зберігання даних використовувати ***томи (Volumes) Docker***.

Том *(Volume)* – це файлова система, яка розташована на хост-машині за межами контейнерів. Створенням і управлінням томами займається *Docker*. Ось основні властивості томів *Docker*:

– вони являють собою засоби для постійного зберігання інформації;

– вони самостійні і відокремлені від контейнерів;

– ними можуть спільно користуватися різні контейнери;

– вони дозволяють організувати ефективне читання і запис даних;

– тома можна розміщувати на ресурсах віддаленого хмарного провайдера;

– їх можна шифрувати;

– їм можна давати імена;

– вони зручні для тестування.

Для створення мережі у Docker необхідно ввести команду ***$docker network,*** вона надасть можливість маніпулювати внутрішніми мережами *docker*.

$docker network create network – створить мережу з ім'ям ***network.***

Якщо необхідно створити мережу з певним драйвером то необхідно додати до команди параметр –***driver*** *або* ***-d***. Приклад, створення мережі типу overlay:

$docker network create -d overlay my-multihost-network.

За допомогою параметрів можливо здійснити розширені налаштування:

– *subnet* – задати підмережу;

– *gateway* – задати маршрут за замовчуванням;

– *ip-range* – вказати діапазон адрес що будуть отримувати контейнери, які знаходяться у мережі;

– *alias* – дозволяє задати ім'я контейнера, за яким можна буде звертатися до нього з мережі (аналог *DNS*).

Таким чином, створення мережі з маскою 172.10.0.0/16 і діапазоном динамічних адрес 172.10.240.0/20 виглядає так:

$docker network create --subnet 172.10.0.0/16 --ip-range 172.10.240.0/20 my-net.

$docker network connect --alias host1 --ip 172.10.241.2 my-net container1 – дана команда додасть контейнер з ім'ям *container1* в мережу *my-net* і присвоїть йому адресу 172.10.241.2.

Контейнер у мережу необхідно додавати тільки один раз, після чого його можна зупиняти і запускати заново, підключення до мережі буде відбуватися автоматично з присвоєнням адреси, зазначеної в *--ip*. У разі, якщо ця адреса вже зайнята, запуск контейнера буде невдалий. Тому слід виділяти адреси за межами діапазону, зазначеного в *--ip-range*. Опція *--alias* дозволяє задати ім'я контейнера, за яким можна буде звертатися до нього з мережі (аналог *DNS*). Крім того, контейнер можна додати в мережу, додавши опції *--network, --network-alias, --ip* в команду *docker run.*

$docker network disconnect [OPTIONS] NETWORK CONTAINER – відключає контейнер від мережі.

$docker network rm name-of-network – видалення мережі.

$docker network ls – виводить список всіх мереж.

$docker network prune – видаляє всі мережі, до яких не підключений жоден контейнер.

$docker network inspect NETWORK – виводить детальну інформацію про мережі, включаючи активні контейнери в мережі.

Базові команди для роботи з Volumes (рис. 1.24):

$docker volume create --name NAME – ***c***творення Volumes;

$docker volume ls – перегляд існуючих Volumes;

$docker volume inspect NAME – перегляд інформації про існуючі Volumes;

```
[node1] (local) root@192.168.0.8 ~
$ docker volume create --name web-page
web-page
[node1] (local) root@192.168.0.8 ~
$ docker volume ls
DRIVER    VOLUME NAME
local     web-page
[node1] (local) root@192.168.0.8 ~
$ docker volume inspect web-page
[
    {
        "CreatedAt": "2021-01-15T09:05:16Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/web-page/_data",
        "Name": "web-page",
        "Options": {},
        "Scope": "local"
    }
]
```

Рис. 1.24. Виконання базових команд для роботи з *Docker Volumes*

Як видно на рис. 1.24 Volumes створений за шляхом ***/var/lib/docker/volumes/web-page/_data.***

Ви маєте можливість створювати та додавати файли, які в майбутньому можливо додати в середину контейнера. Тобто, коли ви створюєте *Volumes* то ви створюєте і папку на хостовій машині, у випадку якщо ви монтуєте *Volumes* до контейнера то весь його зміст буде доступний всередині контейнера.

Запуск контейнера і монтування до нього *Volume* відбувається командою: ***$ docker run -d -v name-of-VOLUME:/usr/share/nginx/html nginx***

/usr/share/nginx/html **–** шлях всередині контейнера куди буде примонтований *Volume*, ***Nginx* –** ім’я *image*.

1.11. Поняття Docker-Compose

Docker Compose – це інструментальний засіб, що входить до складу *Docker*. Він призначений для вирішення завдань, пов'язаних з розгортанням проектів. Вивчаючи основи *Docker*, ви могли зіткнутися зі створенням найпростіших додатків, що працюють автономно, незалежних, наприклад, від зовнішніх джерел даних або від деяких сервісів. На практиці ж подібні додатки – рідкість. Реальні проекти зазвичай включають в себе цілий набір спільно працюють додатків. Як дізнатися, чи потрібно вам, при розгортанні якогось проекту, скористатися *Docker Compose*? Насправді – дуже просто. Якщо для забезпечення функціонування цього проекту використовується кілька сервісів, то *Docker Compose* може вам стати в нагоді. Наприклад, в ситуації, коли створюють вебсайт, з яким, для виконання аутентифікації користувачів, потрібно підключитися до бази даних. Подібний проект може складатися з двох сервісів – того, що забезпечує роботу сайту, і того, який відповідає за підтримку бази даних. Технологія *Docker Compose*, якщо описувати її спрощено, дозволяє, за допомогою однієї команди, запускати безліч сервісів.

Різниця між Docker та Docker Compose. *Docker* застосовується для управління окремими контейнерами (сервісами), з яких складається програма. *Docker Compose* використовується для одночасного управління декількома контейнерами, що входять до складу програми. Цей інструмент пропонує ті ж можливості, що і *Docker*, але дозволяє працювати з більш складними додатками (рис. 1.25).

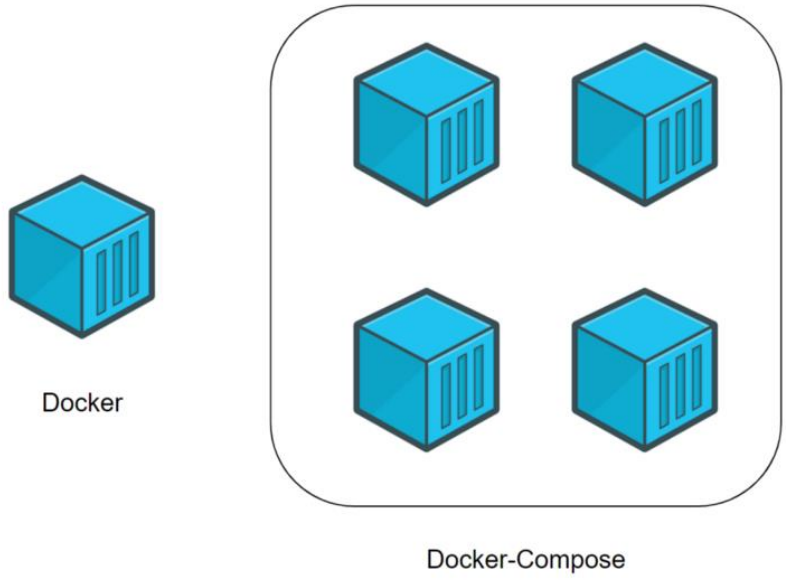


Рис. 1.25. Різниця між *Docker* та *Docker Compose*

Docker Compose – це, в умілих руках, дуже потужний інструмент, що дозволяє дуже швидко розгортати додатки, що відрізняються складною архітектурою. Зараз ми розглянемо приклад практичного використання *Docker Compose*, розбір якого дозволить вам оцінити ті переваги, які дасть вам використання *Docker Compose*. Уявіть собі, що ви є розробником якогось вебпроєкту. У цей проєкт входить два вебсайти. Перший – дозволяє людям, які займаються бізнесом, створювати, всього в кілька кліків мишкою, інтернет-магазини. Другий – націлений на підтримку клієнтів. Ці два сайти взаємодіють з однією і тією ж базою даних. Ваш проєкт стає все популярнішим, і виявляється, що потужності сервера, на якому він працює, вже недостатньо. В результаті ви вирішуєте перевести весь проект на іншу машину. На жаль, щось на зразок *Docker Compose* ви не використали. Тому вам доведеться переносити і перенастроювати сервіси по одному, сподіваючись на те, що ви, в процесі цієї роботи, нічого не забудете. Якщо ж ви використовуєте *Docker Compose*, то перенесення вашого проєкту на новий сервер – це питання, які вирішуються виконанням декількох команд. Для того щоб завершити перенесення проєкту на нове місце, вам потрібно лише виконати деякі налаштування і завантажити на новий сервер резервну копію бази даних.

Для спрощення роботи можливо використовувати ще одну невелику функцію автоматизації. Інструмент *Docker Compose (http://docs.docker.com/compose/)* призначений для швидкого налаштування і запуску різних варіантів середовищ розробки *Docker*. Важливо відзначити, що *Compose* використовує файли *YAML* для зберігання конфігурації груп контейнерів, дозволяючи розробникам позбутися від рутинної роботи і прибирає можливість створення помилок ручної роботи. На даний момент гранична простота додатку не дозволяє належним чином оцінити переваги автоматизації, але додатки зазвичай дуже швидко стають великими і складними. *Compose* звільнить розробників від необхідності супроводжувати всі допоміжні скрипти, призначені для організації роботи, включаючи запуск, встановлення з'єднань, оновлення і зупинку контейнерів. Якщо *Docker* встановлений через *Toolbox*, то *Compose* вже доступний на вашій системі. В іншому випадку виконайте інструкції по установці *Compose* з вебсайту *Docker (http://docs.docker.com/compose/install/).*

В каталозі створимо файл *docker-compose.yml* наступного змісту:

version: ***"3.3"***

services:

nginx:

image: nginx
ports:
- "5000: 5000"
volumes:
“- ./app:/app”

Ключ ***version*** вказує на версію файла, існує велика кількість версій, актуальні на сьогоднішній день 3.0+ .

Після ключа ***services*** оголошується ім’я створюваного контейнера. В одному *YAML*-файлі можна визначити кілька контейнерів (сервіси на діалекті *Compose*).

Кожне визначення контейнера повинно містити або ключ ***build***, або ключ ***image***. Для ключів image задається тег або ідентифікатор способу, з якого створюється контейнер, по аналогії з аргументом ***image*** в команді ***docker run***.

Ключ ***ports*** повністю аналогічний аргументу *-p* в команді *docker run* і служить для оголошення відкритих портів. Тут зв'язується порт 80 в контейнері з портом 80 на хості. Лапки не обов'язкові, але рекомендується їх використання, щоб уникнути вірогідної плутанини при синтаксичному розборі YAML-виразів, наприклад, 56:56 як числа в шістнадцятковій системі рахування.

Ключ ***volumes*** повністю аналогічний аргументу *-v* в команді *docker run* і служить для визначення томів. Тут визначено монтування підкаталогу *app* з поточного каталогу на каталог */app* всередині контейнера точно так, щоб можна було вносити зміни в код прямо з хоста.

У *YAML*-файлах для *Compose* можна визначати багато інших ключів, які в своїй більшості повністю відповідають аргументам команди *docker run*. Тепер при запуску команди ***docker-compose up*** буде отримано в точності такий же результат, який спостерігався після виконання попередньої команди *docker run*.

При роботі з *Compose* найбільш часто використовуються наступні команди. Призначення більшості з них легко визначити по імені, майже всі вони мають аналоги серед команд механізму *Docker*:

– ***docker-compose up*** – запуск всіх контейнерів, визначених у *Compose*-файлі. Висновок журнальних записів об'єднується в один потік. У більшості випадків використовується аргумент **-d** для запуску **Compose** в фоновому режимі;

– ***docker-compose build*** – перестворення всіх образів, створених з файлів *Dockerfile*. Команда ***up*** буде створювати образи, тільки якщо вони не існували раніше, тому команду ***build*** слід використовувати при необхідності поновлення образів;

– ***docker-compose ps*** – виведення інформації про стан контейнерів, керованих Compose;

– ***docker-compose run*** – одноразовий запуск контейнера з виконанням однієї команди (не в якості сервісу). Також запускаються всі контейнери, з якими повинні бути встановлені з'єднання, якщо не заданий аргумент ***--no-deps***. (команди, що передаються через ***run***, замінюють певні команди в файлі конфігурації сервісу. Крім того, за замовчуванням команда ***run*** не створює портів, визначених в файлі конфігурації сервісу);

– ***docker-compose logs*** – висновок журнальних записів з кольоровою підсвіткою, об'єднаний для всіх контейнерів, керованих *Compose*;

– ***docker-compose stop*** – зупинка контейнерів без їх видалення;

– ***docker-compose rm*** – видалення зупинених контейнерів. Слід пам'ятати про те, що аргумент ***-v*** дозволяє видалити всі томи, керовані механізмом *Docker*.

Звичайний порядок роботи починається з виконання команди ***docker-compose up -d*** для запуску програми. Команди ***docker-compose logs*** і ***ps*** можуть використовуватися для перевірки стану програми і як допоміжний засіб при налагодженні. Після внесення змін до початкового коду потрібно виконати ***docker-compose build***, потім ***docker-compose up -d.*** При цьому буде створено новий образ і замінений у контейнері. Важливо, що ***Compose*** зберігає всі колишні томи зі старих контейнерів, таким чином, бази даних і кеші залишаються незмінними при переході до нових версій контейнерів (це може привести до безладдя, тому будьте обережні при заміні контейнерів).

Якщо створення нового образу не потрібно, але внесені зміни в *Compose YAML*-файл, то виконайте команду *docker-compose up -d*, щоб замінити контейнер на точно такий же, але з новими налаштуваннями. Якщо потрібно примусово зупинити весь механізм *Compose* і заново створити всі контейнери, то скористайтеся прапором *--force-recreate*. Після завершення сеансу роботи з додатком виконайте команду *dockercompose stop* для його зупинки. Той же самий комплект контейнерів буде повторно запущений при виконанні команди *docker-compose start* або *up*, якщо не був змінений вихідний код. Для остаточного видалення набору контейнерів, використовуйте команду *docker-compose rm*. Детальний опис всіх команд Compose дивіться на сторінці довідкового керівництва сайту *Docker https://docs.docker.com/compose/reference/*.

1.12. Оркестрація контейнерів

Зазвичай додаток, який побудований з використанням мікросервісної архітектури, складається не з 1-го і навіть не із 5-ти сервісів, які розміщені

у контейнерах, їх кількість може рахуватися десятками. Для керування такою кількістю сервісів не зручно використовувати звичайні інструменти. Важливим стає питання забезпечення відмовостійкості та масштабованості додатку.

Для забезпечення балансування навантаження, масштабованості і підвищення відмовостійкості можуть використовуватися допоміжні засоби – ***оркестратори***.

Оркестрація – автоматичне розташування, координація, керування складними системами і службами.

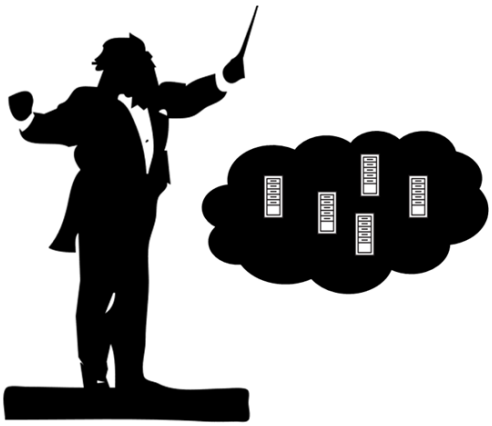


Рис. 1.26. Оркестрація

Оркестрація дозволяє створювати інформаційні системи з безлічі контейнерів, кожен з яких відповідає тільки за одну певну задачу, а спілкування здійснюється через мережеві порти і загальні каталоги. При необхідності кожен такий контейнер можна замінити іншим, що дозволяє, наприклад, швидко перейти на іншу версію бази даних при необхідності. Існують різні платформи для оркестрації контейнерів. Вони дозволяють реалізувати зручні та ефективні засоби розгортання контейнерних систем, побудови єдиної централізованої консолі для застосування політик управління. Найбільш відомі такі системи: *Kubernetes, Docker Swarm і Apache Mesos*. Це не єдині системи, є ще *Nomad, Fleet, Aurora, Microsoft Azure Container Service*, проте вони менш популярні.

Kubernetes – *OpenSource*-система для управління контейнерними кластерами. З'явилася в результаті напрацювань *Google* при використанні механізму для ізоляції процесів у віртуальному середовищі (*Borg*). У 2014 році *Google* відкрила код *Kubernetes* і стала поширювати систему під ліцензією *Apache 2.0.*

Рис. 1.27. Логотип Kubernetes

Kubernetes вибудовує ефективну систему розподілу контейнерів по вузлах кластера в залежності від поточного навантаження і наявних потреб в роботі сервісів. Ця платформа оркестрації здатна обслуговувати величезну кількість хостів, запускати на них численні контейнери *Docker*, відстежувати їх стан, контролювати спільну роботу, проводити балансування навантаження і багато іншого. *Kubernetes* – дуже складна система і дозволяє робити практично все, що стосується оркестрації контейнерів. У неї відмінний *dashboard*, гнучкі політики безпеки, вона підтримує розподілені/мережеві файлові системи, володіє широкими можливостями по визначенню своїх метаданих для сервісів, що зручно у великих інфраструктурах. Але не зважаючи на велику кількість переваг використання *Kubernetes* потребує великих знать у сфері контейнеризації і не тільки. Для початку ідеально підійде *Docker Swarm*

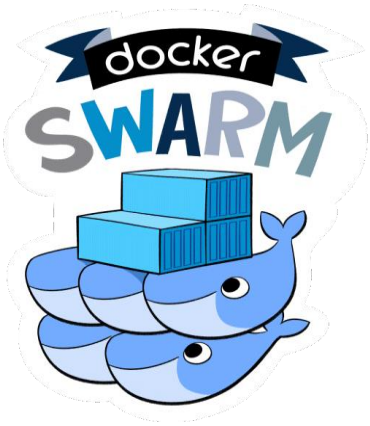


Рис. 1.28. Логотип *Docker Swarm*

Docker Swarm – друга за популярністю система оркестрації і це не випадково. Адже компанія *Docker* була першою, хто запропонував ефективну і зручну для корпоративного використання систему в 2013 році. *Docker* значно спростила розгортання повноцінних віртуальних систем і оркестрації в цілому. Інструмент контейнерної кластеризації *Docker Swarm* з'явився трохи пізніше і став частиною платформи *Docker*. Він дозволяє об'єднувати *Docker*-хости в загальний віртуальний хост.

У режимі *swarm* всі ноди діляться на два типи: *manager* і *worker*. При цьому повноцінний кластер може обходитися без робочих нод взагалі, тобто

менеджери за замовчуванням є також і робочими. Серед менеджерів завжди присутній один, який на даний момент є лідером кластеру. Всі керуючі команди, які виконуються на інших менеджерах автоматично перенаправляються на нього (рис.1.29).

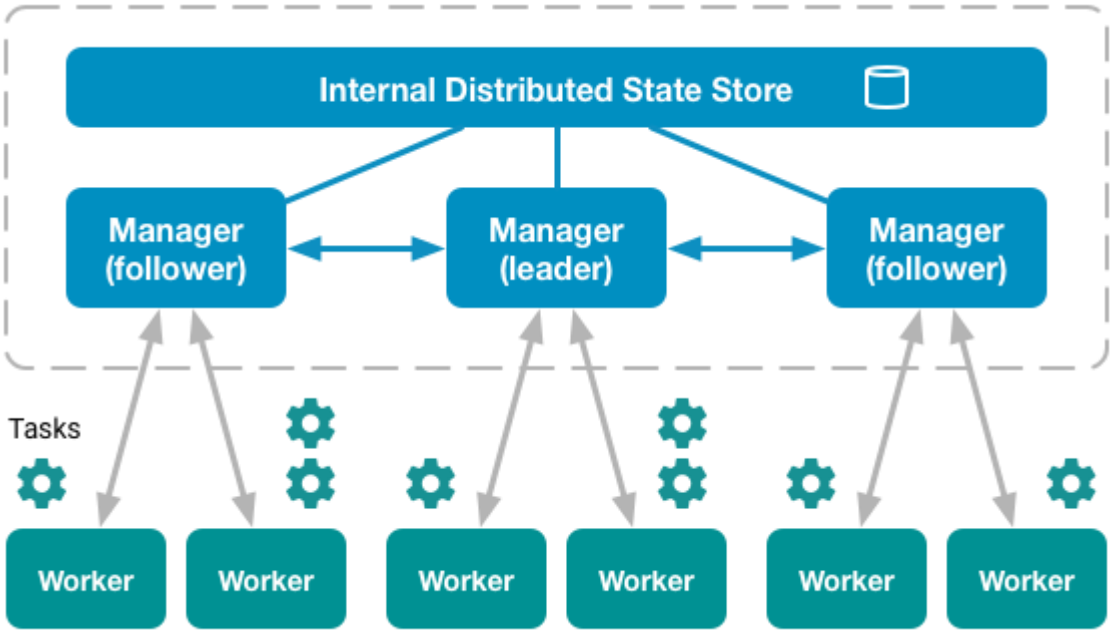


Рис. 1.29 *Swarm Cluster*

Для включення swarm режиму необхідно обрати хост який буде початковим лідером та виконати на ньому одну команду:

$docker swarm init

Працюючий кластер ***Docker Swarm*** готовий для запуску на ньому будь-якої кількості сервісів (контейнерів). Правда, у випадку якщо у кластері існує лише одна нода то стан кластера буде неконсистентним (консистентний стан досягається при кількості менеджерів не менше 3). І звичайно ні про яке масштабування і відмовостійкість в цьому випадку мови також бути не може. Для цього до кластеру потрібно підключити ще хоча б дві керуючі ноди.

Додавати і видаляти ноди можна в будь-який момент життя кластера, це ніяким серйозним чином не впливає на його працездатність.

Створення сервісу не відрізняється від створення контейнеру:

$docker service create --name nginx --publish 8080:80 --replicas 2 nginx:stable-alpine

Відмінності, як правило, полягають в різному наборі опцій. Наприклад, у сервісу немає опції *--volume*, але є опція *--mount* – ці опції дозволяють підключати до контейнерів локальні ресурси нод, але роблять це по-різному.

Зазвичай, щоб оновити одиночний контейнер, доводиться зупиняти поточний і запускати новий. Це призводить хоч і до незначного, але існуючого часу простою вашого сервісу (якщо ви не потурбувалися про те, щоб обробляти такі ситуації за допомогою інших інструментів). При використанні сервісу з кількістю реплік не менше двох – простою сервісу в більшості випадків не відбувається. Це досягається за рахунок того, що *Docker* оновлює контейнери сервісу по черзі. Тобто в один і той же момент часу завжди є хоча б один працюючий контейнер, який може обслужити запит користувача.

Для розподілу запитів між наявними нодами *Docker* використовується схема, яка має назву ***ingress load balacing*** (рис. 1.30).

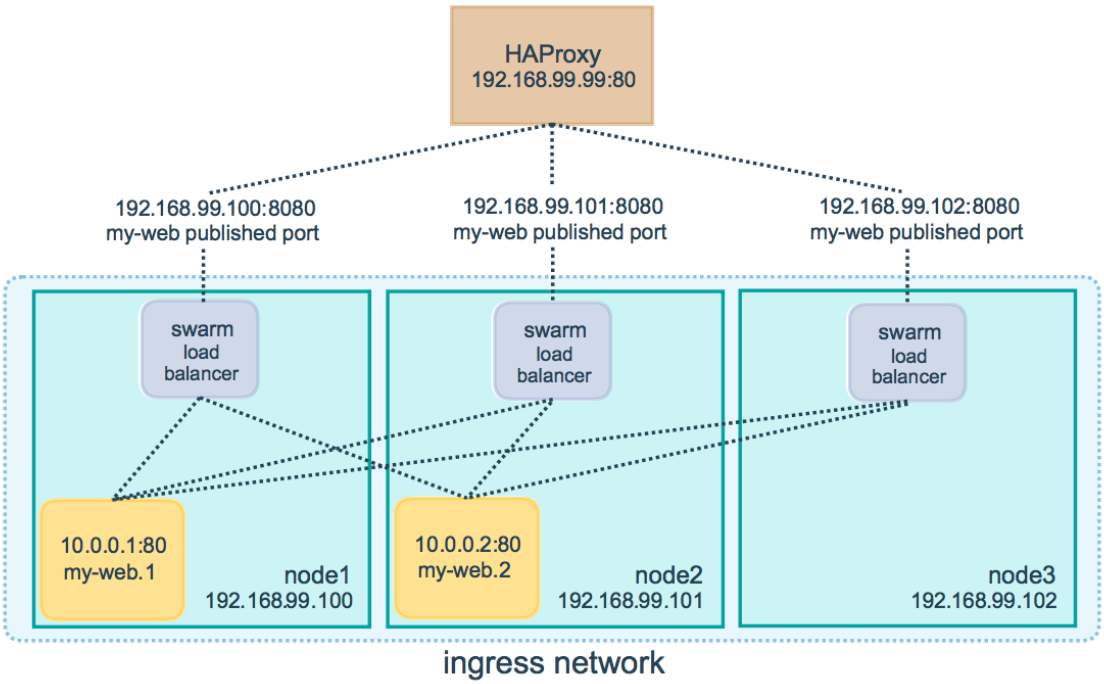


Рис. 1.30. Принцип роботи *ingress load balacing*

Вона полягає у тому, що на яку б з нод не прийшов запит користувача, він спочатку пройде через внутрішній механізм балансування, а потім буде перенаправлений на ту ноду, яка в даний момент може опрацювати такий запит. Тобто, будь-яка нода здатна обробити запит до будь-якого з сервісів кластера.

Масштабування сервісу *Docker* досягається за рахунок вказівки необхідної кількості реплік. У той момент, коли вам необхідно збільшити (або зменшити) кількість нод, які обслуговують запити від клієнта, ви просто оновлюєте параметри обслуговування із зазначенням потрібного значення опції ***--replicas***: ***$docker service update --replicas 3 nginx .***

В цьому випадку треба не забувати попередньо переконатися, що кількість доступних нод не менше, ніж кількість реплік, які ви хочете використовувати. Хоча нічого страшного не станеться навіть якщо нод менше, ніж реплік – просто деякі ноди запустять у себе більше одного контейнера, одного і того ж сервісу (в іншому випадку *Docker* буде намагатися запускати репліки одного сервісу на різних нодах).

Контрольні запитання

1. Що таке віртуалізація?
2. Поясніть поняття віртуальної ЕОМ.
3. Поясніть поняття контейнера.
4. В чому різниця між віртуальною ЕОМ та контейнером?
5. Що таке Docker?
6. Охарактеризуйте можливості та функціонал програмного забезпечення Docker.
7. Як створити образ контейнера.
8. Поясніть покроково порядок розгортання контейнера.
9. Чим корисні курси Play with Docker?
10. Перерахуйте основний функціонал Docker Hub.
11. Поясніть, що таке Docker-Compose.
12. Що таке тег?
13. Які основні драйвери в Docker забезпечують мережеві функції?
14. Які типи зберігання інформації в Docker ви знаєте?
15. Що таке Том в Docer?
16. Назвіть основні способи взаємодії та управління програмним забезпеченням Docker.
17. Яка різниця між Docker та Docker Compose?
18. Що таке YAML?
19. Що таке оркестрація?
20. Перелічіть основні програмні засоби, які використовуються для оркестрації контейнерами.

2. ІНФРАСТРУКТУРА ЯК КОД. ІНСТРУМЕНТИ УПРАВЛІННЯ КОНФІГУРАЦІЄЮ

2.1. Інфраструктура як код

Інфраструктура як код або ***IaC*** – це спосіб постачання та керування обчислювальними та мережевими ресурсами методом їх опису у вигляді програмного коду, на відміну від налаштування необхідного обладнання власноруч чи з допомогою інтерактивних інструментів. IT інфраструктура, керована таким чином, що охоплює як фізичні сервери, так і віртуальні машини, а також пов'язані з ними ресурси. У підході використовуються як виконувані скрипти, так і декларативні визначення, шаблони, які можуть перебувати в системі контролю версій. Термін найчастіше використовується для просування декларативного опису IT інфраструктури (рис. 2.1).

```
{
  "Resources" : {
    "myCFEc2" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "KeyName" : "MyOregonEC2Key",
        "ImageId" : "ami-d0f506b0",
        "InstanceType" : "t2.micro",
        "IamInstanceProfile" : "S3-Admin-Access",
        "NetworkInterfaces": [{
                "AssociatePublicIpAddress" : "true",
                "DeviceIndex" : "0",
                "GroupSet" : ["sg-f667ed91"],
                "SubnetId" : "subnet-9fb0a9e8"
        }]
      }
    }
  }
}
```

Рис. 2.1. Приклад декларативного опису IT інфраструктури

IaC подається, в першу чергу, як рішення для платформ хмарних обчислень, які, в свою чергу іноді позиціюються як IaaS рішення. IaC підтримує IaaS, але їх не слід плутати.

Історія IaC є симбіозом з двох речей: простота створення віртуальної інфраструктури і нової епохи побудови web-сайтів. У 2006 був запущений Amazon Web Services Elastic Compute Cloud і Ruby on Rails версії 1.0, які дозволили підняти питання масштабування, з яким раніше стикалися тільки величезні компанії. З новими інструментами для вирішення цієї проблеми і з'явився підхід IaC. Ідея опису інфраструктури через код, з подальшою можливістю застосування практик з розробки програмного забезпечення з'явилася в головах розробників і системних адміністраторів. Ідея обходитися з інфраструктурою як з кодом дозволила розробникам блискавично розгортати додатки.

2.2. Сучасні інструменти управління конфігурацією

Управління конфігурацією (англ. software configuration management, SCM) в програмній інженерії – комплекс методів, спрямованих на систематичний облік змін, що вносяться розробниками в програмний продукт в процесі його розробки та супроводу, збереження цілісності системи після змін, запобігання небажаних і непередбачуваних ефектів, формалізація процесу внесення змін.

В ряді джерел можна побачити абревіатуру SCCM – Software Configuration and Change Management. При тому, що в розумінні SWEBOK і відповідних стандартів SCM і SCCM тотожні, термін SCCM іноді використовується для того, щоб підкреслити принципову значимість управління змінами як складової частини конфігураційного управління.

SCM-діяльність тісно пов'язана з роботами по забезпеченню якості програмного забезпечення (Software Quality Assurance – SQA).

Роботи з конфігураційного керування програмного забезпечення включають:

- управління і планування SCM-процесів;
- ідентифікацію програмних конфігурацій;
- контроль конфігурацій;
- облік статусів конфігурацій;
- аудит, а також управління випуском (release management);
- доставкою (delivery) ПЗ до кінцевого користувача.

Керування конфігурацією – це ідентифікація компонентів системи, визначення функціональних, фізичних характеристик системи, апаратного і програмного забезпечення для контролю виконання, внесення змін і трасування конфігурації. Процес керування визначено як один з допоміжних процесів ЖЦ (ISO/IEC 12207), що виконується технічним і адміністративним менеджментом проєкту. При цьому складаються звіти про зміни, внесені у конфігурацію, і ступінь їхньої реалізації, а також проводиться перевірка відповідності внесених змін заданим вимогам.

Конфігурація системи – це перелік функцій, програмного і технічного забезпечення системи, можливі їх комбінації залежно від наявності устаткування, загальносистемних засобів і вимог до продукту.

Конфігурація ПЗ складається з набору функціональних і технічних характеристик ПЗ, заданих у технічній документації і реалізованих у готовому продукті. Це сполучення різних елементів продукту з заданими процедурами збирання компонентів і настроювання на середовище. Вхідними елементами конфігурації є графік розробки, проектна документація, вихідний виконуваний код, бібліотека компонентів, інструкції з установки і розгортання системи.

2.2.1. ANSIBLE

Рис. 2.2. Логотип програмного забезпечення Ansible

Ansible – програмне забезпечення, що надає засоби для управління конфігурацією, оркестровки, централізованої установки застосунків і паралельного виконання типових завдань на групі систем. Код Ansible написаний мовою Python і розповсюджується під ліцензією GPLv3.

З особливостей Ansible можна відзначити просту і легко читану мову управління конфігурацією, підтримку розпаралелювання робіт, відсутність необхідності установки на віддалені системи спеціальних програм-агентів (всі операції ініціюються централізовано по SSH), можливість роботи без прав root. Система Ansible не так ускладнена, як, Puppet і Chef, але при цьому надає досить широкі можливості та високу гнучкість управління.

Ansible дозволяє автоматизувати мережеву, серверну, хмарну інфраструктури з мінімальними затратами.

Ansible – інструмент, що має безагентну архітектру, що дозволяє робити будь-які зміни на клієнті за допомогою SSH-підключення. Інструмент легковісний, а також з малим порогом входу до його використання.

Переваги:

- безагентна архітектура;
- простий у використанні;
- модульний;
- відкритий код.

Ansible функціонує за push-типом застосування IaC, тобто він “штовхає” конфігурацію та налаштування на сервер, на керуємих серверах не встановлено ніякого агенту, що означає скорочення часу на налаштування.

Мінімальні вимоги для роботи:

- Linux система (Red Hat, Debian, CentOS, Ubuntu, OS X);
- Python 2.6 або 3.5;
- відкритий SSH 22 порт на Linux, на Windows WinRM порт 5986;
- username + password на серверах, які будуть налаштовуватись або SSH pub.key.

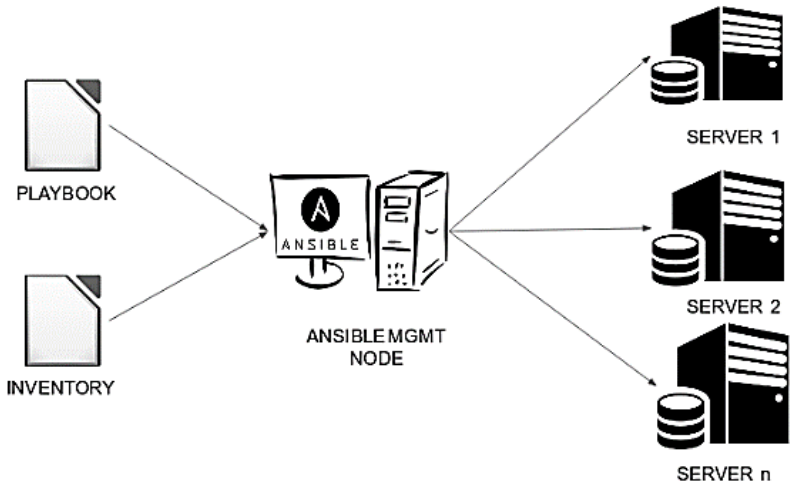


Рис. 2.3. Принцип застосування IaC за допомогою Ansible

Основні поняття в Ansible

***Керуюча нода* (*Control node*)** – будь-яка ЕОМ, на якій встановлено Ansible. Ви можете запускати команди Ansible та playbooks, викликаючи команду ansible або ansible-playbook з будь-якого керуючого вузла. Ви можете використовувати будь-який комп'ютер, на якому встановлено Python, як керуючий вузол – ноутбуки, спільні робочі столи та сервери можуть запускати Ansible. Однак, ви не можете використовувати ЕОМ з операційною системою Windows як керуючий вузол. Ви можете мати кілька вузлів управління.

***Керуємі ноди* (*Managed nodes*) –** мережеві пристрої (сервери), якими ви керуєте за допомогою Ansible. Керовані вузли також іноді називають "хостами". Ansible не встановлюється на керованих вузлах.

***Інвентар* (*Inventory*) –** список керованих вузлів. Файл інвентаризації також іноді називають "файлом хосту". У вашому інвентарі може бути вказана така інформація, як IP-адреса для кожного керованого вузла. Інвентар може також організовувати керовані вузли, створюючи та вкладаючи групи для спрощення масштабування.

***Колекції* (*Collections*)** – це формат розповсюдження вмісту Ansible, який може включати playbooks, ролі, модулі та плагіни. Ви можете встановлювати та використовувати колекції через Ansible Galaxy.

***Модулі* (*Modules*)** – одиниці коду, які виконуються Ansible. Кожен модуль має особливе використання, від адміністрування користувачів певного типу бази даних до управління інтерфейсами VLAN на певному типі мережевих пристроїв. Ви можете викликати один модуль із завданням або викликати кілька різних модулів у playbook. Починаючи з Ansible 2.10, модулі групуються в колекції.

***Завдання* (Tasks)** – одиниці дій в Ansible. Ви можете виконати одне завдання один раз за допомогою спеціальної команди.

***Списки завдань* (Playbooks)** – впорядковані списки завдань, збережені, щоб ви могли виконувати ці завдання в такому порядку

неодноразово. Списки завдань можуть містити змінні, а також завдання. Списки завдань збережені в YAML і їх легко читати, писати, ділитися та розуміти.

2.2.2. Chef

Рис. 2.4. Логотип програмного забезпечення Chef

Chef Infra – це потужна платформа автоматизації, яка перетворює інфраструктуру на код. Незалежно від того, працюєте ви в хмарі, локально або в гібридному середовищі, Chef Infra автоматизує налаштування, розгортання та управління інфраструктурою у вашій мережі, незалежно від її розміру.

Діаграма показує як розробляти, тестувати та доставляти код Chef Infra.

Рис. 2.5. Процес розробки, тестування та доставлення коду Chef Infra

Використання робочої станції Chef. Chef Workstation дозволяє створювати списки налаштувань (**cookbooks**) та керувати своєю інфраструктурою. Chef Workstation працює на комп'ютері, яким ви користуєтеся щодня, будь то Linux, macOS або Windows.

Робоча станція "шеф-кухаря" постачається з інструментами тестування Cookstyle, ChefSpec, Chef InSpec та Test Kitchen. За допомогою них ви можете переконатись, що ваш код Chef Infra виконує те, що ви задумали, перед тим, як розгортати його в середовищах, що використовуються іншими, наприклад, на виробництві.

Коли ви пишете свій код, ви використовуєте ресурси для опису вашої інфраструктури. Ресурс відповідає деякій частині інфраструктури, наприклад файлу, шаблону або пакету. Кожен ресурс декларує, в якому стані повинна бути частина системи, але не як туди потрапити. Шеф-кухар Інфра вирішує ці труднощі за вас. Chef Infra надає безліч ресурсів, готових

до використання. Ви також можете використовувати ресурси, що надходять у cookbooks-громади, або написати власні ресурси, специфічні для вашої інфраструктури.

Recept Chef Infra – це файл, який групує пов'язані ресурси, такі як усе, що потрібно для налаштування вебсервера, сервера баз даних або балансу навантаження. "Кулінарна книга" Chef Infra надає структуру вашим "рецептам" і загалом допомагає вам бути впорядкованими.

Робоча станція Chef включає інші інструменти командного рядка для взаємодії з Chef Infra.

Завантаження коду на Chef Infra Server. Завершивши розробку та тестування коду на локальній робочій станції, ви можете завантажити його на сервер Chef Infra. Сервер Chef Infra виступає в ролі концентратора для даних конфігурації. Тут зберігаються списки конфігурацій, політики, що застосовуються до систем вашої інфраструктури, та метадані, що описують кожну систему. Команда управління дозволяє вам спілкуватися з Chef Infra Server із вашої робочої станції.

Налаштування вузлів за допомогою Chef Infra Client. Chef Infra побудований таким чином, що більша частина обчислювальних зусиль відбувається на вузлах, а не на сервері Chef Infra. Вузол представляє будь-яку систему, якою ви керуєте, і, як правило, це віртуальна машина, екземпляр контейнера або фізичний сервер. По суті, це будь-який обчислювальний ресурс у вашій інфраструктурі, яким керує Chef Infra. На всіх вузлах встановлений Chef Infra Client, а Chef Infra Client доступний для багатьох платформ, включаючи Linux, macOS, Windows, AIX та Solaris.

Періодично клієнт Chef Infra звертається до сервера Chef Infra, щоб отримати найновіші конфігураційні списки. Якщо поточний стан вузла не відповідає тому, що вказаний в списках конфігурації, то Chef Infra Client виконує інструкції та переналаштовує відповідний елемент інфраструктури відповідно до конфігураційного файлу. Цей ітеративний процес забезпечує зближення мережі в цілому до стану, передбаченого бізнес-політикою

Chef Habitat пропонує новий підхід до розгортання програм, який називається автоматизацією додатків. Автоматизація додатків означає, що автоматизація упаковується разом із додатком і подорожує разом із ним, незалежно від того, де ця програма розгорнута. Блоком розгортання стає додаток та пов'язана з ним автоматизація. Середовище виконання, будь то контейнер, голий метал або PaaS, жодним чином не впливає на програму.

Chef Habitat складається з формату упаковки (packading format) та керівника (supervisor). Формат визначає пакети Chef Habitat, які є ізольованими, незмінними та перевіряються. Chef Habitat знає, як взяти пакети та запустити їх. Він знає про взаємовідносини між пакетами, його стратегію оновлення та політику безпеки.

Chef InSpec – це тестова система з відкритим кодом, яка читається людиною та машиною, щоб визначити відповідність вимогам безпеки та політики. Коли відповідність виражається як код, ви можете інтегрувати

його в конвеєр розгортання і автоматично перевірити на відповідність політикам безпеки.

Код Chef InSpec може працювати на декількох платформах. Ви можете виконувати той самий набір тестів локально, за допомогою віддалених команд, що використовують SSH або WinRM, або за допомогою зовнішніх механізмів, таких як Docker API.

За допомогою Chef InSpec ви можете зробити більше, ніж забезпечити відповідність ваших фізичних серверів. Наприклад, ви можете оцінити дані в базі даних або перевірити конфігурацію віртуальних ресурсів, використовуючи їх API.

Автоматизація Chef **(**Chef Automate) надає повний набір корпоративних можливостей для видимості та відповідності вузлів. Chef Automate інтегрується з продуктами з відкритим кодом Chef Infra Client, Chef InSpec та Chef Habitat. Chef Automate постачає комплексні послуги підтримки цілодобово та без вихідних для всієї платформи, включаючи компоненти з відкритим кодом.

Chef Automate надає вам повний стек безперервної відповідності та безпеки, а також видимість ваших програм та інфраструктури

Ноди (Nodes) Chef Automate надають сховище даних, яке приймає вхідні дані від Chef Server, Chef Habitat та Chef Automate про робочий процес та відповідності. Він надає уявлення про оперативні та робочі процеси. Існує мова запитів, вона доступна через інтерфейс користувача та інформаційні панелі.

Відповідність **(**Compliance**)** Chef Automate створює звіти, які визначають проблеми дотримання вимог, ризики безпеки та застаріле програмне забезпечення. Ви можете написати власні правила відповідності в програмі Chef InSpec, або можна швидко розпочати роботу, використовуючи вбудовані профілі, які є заздалегідь визначеними наборами правил для різних систем безпеки, таких як Тести центру інтернет-безпеки (CIS), включені як частина Chef Automate.

Висока доступність (High availibility) Chef Automate включає високодоступний сервер Chef Infra з відмовостійкістю, негайними результатами пошуку та точними даними про вашу інфраструктуру в реальному часі. Chef Automate також надає графічну консоль управління для сервера Chef Infra.

2.2.3. Puppet

Рис. 2.6. Логотип програмного забезпечення Puppet

Puppet – це інструмент, який допомагає керувати та автоматизувати конфігурацію серверів.

Використовуючи Puppet, ви визначаєте бажаний стан систем у вашій інфраструктурі, якою ви хочете керувати. Ви робите це, написавши код інфраструктури предметно-орієнтованою мовою програмування – Puppet Code, який ви можете використовувати з широким спектром пристроїв та операційних систем. Puppet код є декларативним, що означає, що ви описуєте бажаний стан своїх систем, а не кроки, необхідні для того, щоб туди потрапити. Потім Puppet автоматизує процес приведення цих систем у такий стан і їх утримання там. Puppet робить це за Puppet master та Puppet agent. ***Puppet master*** – це сервер, на якому зберігається код, який визначає бажаний стан налаштовуваних систем. Агент Puppet переводить ваш код у команди, а потім виконує його у вказаних вами системах, що називається Puppet run.

На діаграмі нижче показано, як працює архітектура майстра-агента Puppet run.

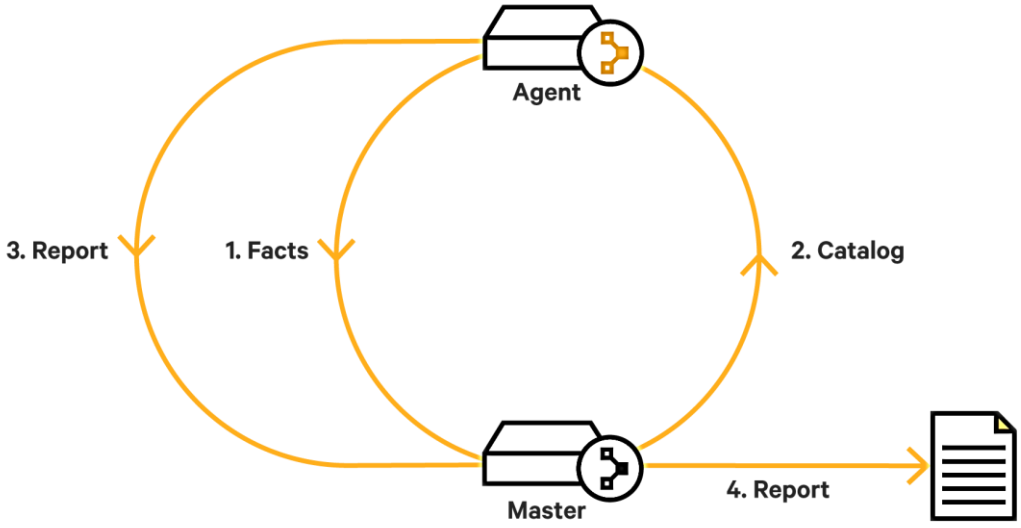


Рис. 2.7. Робота архітектури Puppet-run

Схема роботи Puppet – клієнт-серверна, хоча підтримується і варіант роботи без сервера з обмеженою функціональністю.

Використовується pull-модель роботи: за замовчуванням раз в півгодини клієнти звертаються до сервера за конфігурацією і застосовують її. Якщо ви працювали з Ansible, то там використовується інша, push-модель: адміністратор ініціює процес застосування конфігурації, самі по собі клієнти нічого не застосовуватимуть.

При мережевій взаємодії використовується двостороннє TLS-шифрування: у сервера і клієнта є свої закриті ключі і відповідні їм сертифікати. Зазвичай сервер випускає сертифікати для клієнтів, але в принципі можливо використання і зовнішнього CA.

У термінології Puppet до паппет-сервера підключаються Ноди (nodes). Конфігурація для нод пишеться в маніфестах спеціальною мовою програмування – Puppet DSL.

Puppet DSL – декларативна мова. На ній описується бажаний стан Ноди в вигляді оголошення окремих ресурсів, наприклад:

– файл існує, і у нього певний вміст;
– пакет встановлений;
– сервіс запущено.

Ресурси можуть бути взаємопов'язані:

– є залежності, вони впливають на порядок використання ресурсів. Наприклад, “спочатку встанови пакет, потім поправ конфігураційний файл, після цього запусти сервіс”.
– є повідомлення – якщо ресурс змінився, він відправляє повідомлення підписаним на нього ресурсів. Наприклад, якщо змінюється конфігураційний файл, можна автоматично перезапускати сервіс.

Крім того, в Puppet DSL є функції і змінні, а також умовні оператори і селектори. Також підтримуються різні механізми шаблонізації – EPP і ERB.

Puppet написаний на Ruby, тому багато конструкцій і терміни взяті звідти. Ruby дозволяє розширювати Puppet, дописувати складну логіку, нові типи ресурсів, функції.

Під час роботи Puppet, маніфести для кожної конкретної Ноди на сервері компілюються в каталог. Каталог – це список ресурсів і їх взаємозв'язків після обчислення значення функцій, змінних і розкриття умовних операторів.

Синтаксис і кодстайл. Ось розділи офіційної документації, які допоможуть розібратися з синтаксисом, якщо наведених прикладів буде недостатньо:

– синтаксис різних мовних конструкцій;
– синтаксис ресурсів;
– синтаксис опису нод.

Розташування файлів на паппет-сервері. Для подальших пояснень введемо поняття «коренева директорія». Коренева директорія – це директорія, в якій знаходиться Puppet-конфігурація для конкретної Ноди.

Коренева директорія різниться в залежності від версії Puppet і використання оточень. Оточення – це незалежні набори конфігурації, які зберігаються в окремих директоріях. Зазвичай використовуються в поєднанні з Git, в такому випадку оточення створюються з гілок Git. Відповідно, кожна нода знаходиться в тому чи іншому оточенні. Це налаштовується на самій ноді, або в ENC.

У третій версії (“старий Паппет”) базовою директорією була **/etc/puppet**. Використання оточень опціональне. Якщо оточення використовуються, то вони зазвичай зберігаються в **/etc/puppet/environments**, кореневою директорією буде директорія

оточення. Якщо оточення не використовуються, кореневою директорією буде базова.

Починаючи з четвертої версії (“новий Паппет”), використання оточень стало обов'язковим, а базову директорію перенесли в **/etc/puppetlabs/code**. Відповідно, оточення зберігаються в **/etc/puppetlabs/code/environments**, коренева директорія – директорія оточення.

У кореневій директорії повинна бути піддиректорія manifests, в якій лежить один або кілька маніфестів з описом нод. Крім того, там повинна бути піддиректорія modules, в якій лежать модулі. Що таке модулі, я розповім трохи пізніше. Крім того, в старому Паппеті також може бути піддиректорія files, в якій лежать різні файли, які ми копіюємо на Ноди. У новому Паппеті ж всі файли винесені в модулі.

Файли маніфестів мають розширення **.pp.**

Опис Ноди і ресурсу на ній. На ноді server1.testdomain повинен бути створений файл /etc/issue з вмістом Debian GNU/Linux \ n \ l. Файл повинен належати користувачу і групі root, права доступу повинні бути 644.

Приклад маніфесту:

```
node 'server1.testdomain' { # блок конфігурації, що відноситься до ноди server1.testdomain
    file { '/etc/issue': # описуємо файл /etc/issue
        ensure => present, # цей файл повинен існувати
        content => 'Debian GNU/Linux \n \l', # в ньому повинен бути такий вміст
        owner => root, # користувач-власник
        group => root, # група-власник
        mode => '0644', # права на файл. Вони задані в лапках у вигляді рядка, тому що в іншому випадку число з 0 на початку буде сприйматися, як записане у восьмиричній системі і все піде не так як задумано
        }
}
```

2.2.4. SaltStack

Рис. 2.8. Логотип програмного забезпечення SaltStack

Salt або ***SaltStack*** – це засіб віддаленого виконання операцій та система управління конфігурацією. Можливості віддаленого виконання

операцій дозволяють адміністраторам виконувати команди на різних машинах паралельно з гнучкою системою управління конфігурацією. Функціонал управління конфігурацією встановлює модель клієнт-сервер для швидкого, простого та безпечного приведення компонентів інфраструктури у відповідність із заданою політикою.

Ролі машин Salt. Salt master – це ЕОМ, яка контролює інфраструктуру та диктує політику для серверів, якими вона керує. Вона працює як сховище для даних конфігурації, так і як центр управління, який ініціює віддалені команди та забезпечує стан інших підлеглих ЕОМ. Для забезпечення цієї функціональності на майстрі встановлений демон, який називається salt-master.

Незважаючи на те, що можна керувати інфраструктурою, використовуючи конфігурацію без майстра, більшість установок користуються розширеними функціями, доступними в майстрі Salt. Насправді, для більшого управління інфраструктурою, Salt має можливість делегувати певні компоненти та завдання, як правило, пов'язані з майстром, на виділені сервери. Він також може працювати в багаторівневій майстер-конфігурації, де команди можуть передаватися через нижчі майстер-машини.

Ставленики (Minions) **–** сервери, якими управляє Salt, називаються ставленниками. Демон, що називається salt-minion, встановлюється на кожній з керованих машин і налаштовується для зв'язку з майстром. Міньйон відповідає за виконання інструкцій, надісланих майстром, звітування про успіх завдань та надання даних про основного хоста.

Комунікація та зв’язок компонентів Salt. Майстри Salt та ставленики за замовчуванням спілкуються за допомогою бібліотеки обміну повідомленнями ZeroMQ. Це забезпечує надзвичайно високу продуктивність мережевого зв'язку між сторонами, дозволяючи Salt надсилати повідомлення та дані на великих швидкостях. Оскільки ZeroMQ – це бібліотека, а не незалежний сервіс, ця функціональність доступна в демонах salt-master та salt-minion.

Використовуючи ZeroMQ, Salt підтримує систему відкритих ключів для автентифікації майстрів та ставлеників. При першому завантаженні ставленик генерує пару ключів і відправляє свої облікові дані на головний сервер, з яким налаштований контакт. Потім майстер може прийняти цей ключ після перевірки особи міньйона. Потім обидві сторони можуть швидко та безпечно спілкуватися за допомогою ZeroMQ, зашифрованого за допомогою ключів.

Якщо з якихось причин неможливо встановити демон-Salt- minion на вузол, Salt також може видавати команди по SSH. Цей варіант транспортування надається для зручності, але він значно погіршує продуктивність і в деяких випадках може призвести до ускладнень з іншими командами Salt. Наполегливо рекомендується використовувати демон-Salt minion, де це можливо, для продуктивності, безпеки та простоти.

Термінологія Salt. Віддалене виконання, виконання модулів та функцій. Salt намагається розрізнити віддалене виконання функцій та функції управління конфігурацією. Можливості віддаленого виконання забезпечуються модулями виконання (**execution modules**). Модулі виконання – це набори пов'язаних функцій, які виконують роботу над міньйонами.

Хоча Salt включає функції, які дозволяють запускати довільні команди оболонки на міньйонів, ідея модулів виконання полягає у забезпеченні стислого механізму для виконання команд без необхідності "викладати" та надавати докладні інструкції щодо завершення процесу. Використання модулів дозволяє Salt абстрагувати основні відмінності між системами. Ви можете отримати подібну інформацію від ставлеників під управлінням Linux або BSD, хоча фактичні механізми збору даних будуть різними.

Salt поставляється з гідним вибором вбудованих модулів виконання, щоб забезпечити нестандартну функціональність. Адміністратори також можуть писати власні модулі або включати написані спільнотою модулі, щоб розширити бібліотеку команд, які можна виконувати на машинах міньйонів.

Управління конфігурацією: стани (states), формули (formulas), шаблони (templates). Функціонал управління конфігурацією Salt можна отримати, створивши сховища конфігураційних файлів. Файли, що містяться в цих сховищах, можуть бути декількох різних типів.

***Стани та формули*.** Частина управління конфігурацією Salt в основному реалізується за допомогою системи станів.

Система стану використовує модулі стану, які відрізняються від модулів виконання, описаних вище. На щастя, модулі стану та виконання, як правило, досить добре відображають один одного. Система стану має відповідну назву, оскільки вона дозволяє адміністратору описати стан, в який повинна бути розміщена система. Як і у випадку з модулями виконання, більшість модулів стану представляють ярлики функціональних можливостей і забезпечують легкий синтаксис для багатьох типових дій. Це допомагає підтримувати читабельність і усуває необхідність включення складної логіки в самі файли управління конфігурацією.

Salt формули – це набори викликів модулів стану, організованих з метою отримання певного результату. Це файли управління конфігурацією, що описують, як повинна виглядати система після застосування формули. За замовчуванням вони записані у форматі серіалізації даних YAML, що забезпечує дуже хороший проміжок між високою читабельністю та зручністю роботи з машиною.

Адміністратор Salt може застосовувати формули, відображаючи ставлеників до певних наборів формул. Формули також можуть застосовуватися спеціально, за потреби. Ставленики виконуватимуть

знайдені всередині модулі стану, щоб привести свою систему у відповідність із наданою політикою.

Формули Salt можна знайти за посиланням https://github.com/saltstack-formulas.

Шаблони дозволяють формули Salt та інші файли писати більш гнучко. Шаблони можуть використовувати інформацію, доступну про ставлеників, для створення індивідуальних версій файлів формул або конфігурації. За замовчуванням Salt використовує формат шаблону Jinja, який забезпечує функціональність заміщення та прості логічні конструкції для прийняття рішень.

Візуалізатори (renderer) – це компоненти, що запускають шаблон для створення дійсних файлів стану або конфігурації. Візуалізатори визначаються форматом шаблону, який становить вхідні дані, та форматом серіалізації даних, який буде створений як вихідний файл. Враховуючи описані вище за замовчуванням, візуалізатор за замовчуванням обробляє шаблони Jinja для створення файлів YAML.

Запит на допис інформації до ставлеників. Для управління величезною кількістю систем Salt вимагає певної інформації про кожну з основних систем. Описані вище шаблони можуть використовувати дані, пов'язані з кожною системою, для адаптації поведінки кожного ставленика. Існує кілька різних систем для запитів або призначення цієї інформації хостам.

Зерна (grains). Для управління величезною кількістю систем Salt вимагає певної інформації про кожну з основних систем. Описані вище шаблони можуть використовувати дані, пов'язані з кожною системою, для адаптації поведінки кожного ставленика. Існує кілька різних систем для запитів або призначення цієї інформації хостам.

Salt-зерна – це частина інформації, яку збирає і підтримує ставленик, в першу чергу стосовно основної системи прийому. Вони, як правило, збираються демоном Salt-minion і передаються назад майстру за запитом. Цю функціональність можна використовувати для багатьох різних цілей.

Наприклад, зернові дані можуть бути використані для націлювання на конкретну підмножину вузлів з пулу ставлеників для віддаленого виконання або управління конфігурацією. Якщо ви хочете побачити час безвідмовної роботи ваших серверів Ubuntu, зерна дозволяють орієнтуватися лише на ці машини.

Зерна також можна використовувати як аргументи для змін конфігурації або команд. Наприклад, ви можете використовувати зерна, щоб отримати адресу IPv4, пов'язану з інтерфейсом eth0, для зміни конфігураційного файлу або як аргумент команди.

Адміністратори також можуть призначати зерна ставленикам. Наприклад, досить часто використовують зерна для призначення «ролі» серверу. Потім це можна використовувати для націлювання на підмножину вузлів, подібну до прикладу операційної системи вище.

Стовпи (pillars). Хоча можливо призначити зерна ставленикам, переважна більшість змінних конфігурації буде призначена через систему стовпів. У Salt стовп являє собою сховище ключ-значення, яке міньйон може використовувати для отримання довільно призначених даних. Це функціонує як структура даних словника, яка може бути вкладена або багаторівнева для організаційних цілей.

Стовпи пропонують кілька важливих переваг перед системою зерна для присвоєння цінностей. Найголовніше, що дані стовпів доступні лише для ставлеників, призначених йому. Інші сервери не матимуть доступу до значень, що зберігаються всередині. Це робить його ідеальним для зберігання конфіденційних даних, характерних для вузла або підмножини вузлів. Наприклад, секретні ключі або рядки підключення до бази даних часто надаються в конфігурації стовпа.

Дані стовпа часто використовуються в контексті управління конфігурацією як спосіб введення змінних даних у шаблон конфігурації. Salt пропонує вибір форматів шаблонів для заміни змінних частин файлу конфігурації елементами, характерними для вузла, який буде його застосовувати. Зерна також часто використовуються таким чином для посилання на дані хоста.

Salt-mine – це область на головному сервері, де можна зберігати результати, команд що регулярно виконуються над ставленниками. Призначення цієї системи – збирати результати довільних команд, що виконуються на машинах ставлеників. Потім це глобальне сховище може запитуватися іншими компонентами та ставленниками по всій вашій інфраструктурі.

Salt-шахта зберігає лише останні результати для кожного запуску команди, тобто це не допоможе вам, якщо вам потрібен доступ до історичних даних. Основною метою шахти є надання актуальної інформації від ЕОМ ставлеників як гнучке доповнення до вже наявних даних про зерно. Ставленики можуть запитувати дані про своїх колег, використовуючи шахтну систему. Інтервал, в якому ставленик оновлює дані в шахті, можна налаштувати для кожного ставленика.

Додатковий функціонал. Система ***Salt-реакторів*** забезпечує механізм ініціювання дій у відповідь на генеровані події. У програмі Salt зміни, що відбуваються у всій вашій інфраструктурі, змусять демони Sal-Minion або Sal-Master-генерувати події на шині повідомлень ZeroMQ. Реакторна система спостерігає за цією шиною та порівнює події зі своїми сконфігурованими реакторами, щоб відповісти належним чином.

Основною метою реакторної системи є забезпечення гнучкої системи для створення автоматизованих ситуаційних реакцій. Наприклад, якщо ви розробили стратегію автоматичного масштабування, ваша система автоматично створить вузли для динамічного задоволення ваших потреб у ресурсах. Кожен новий вузол викликав би подію. Реактор можна було б

налаштувати для прослуховування цих подій та конфігурації нового вузла, включаючи його до існуючої інфраструктури.

Salt-ранер – це модулі, які виконуються на головному сервері замість ставлеників. Деякі ранери – це утиліти загального призначення, які використовуються для перевірки стану різних частин системи або для технічного обслуговування. Деякі з них є потужними програмами, які надають можливість організувати свою інфраструктуру в більш широкому масштабі.

Salt-повертачі використовуються для вказівки альтернативних місць, куди надсилатимуться результати запуску на ставленика. За замовчуванням ставленики повертають свої дані майстру. Повертач дозволяє адміністратору перенаправити дані повернення в інший пункт призначення. Як правило, це означає, що результати повертаються до місця призначення, вказаного повертачем, і до процесу, який ініціював команду ставленика.

Найчастіше повертачі, передають результати системі баз даних або метриці або службі реєстрації. Це забезпечує гнучкий метод отримання довільних даних до цих систем. Повертачі також можуть використовуватися для збору специфічних для солі даних, таких як кеші завдань та дані про події.

2.2.5. Порівняння інструментів управління конфігурацією

Швидкий розвиток віртуалізації разом зі збільшенням потужності серверів, які відповідають промисловим стандартам, а також доступність “хмарних” обчислень привели до значного зростання числа серверів, що потребують управління, як всередині, так і поза організацією. І якщо колись ми робили це за допомогою стійок з фізичними серверами в центрі обробки даних поверхом нижче, то тепер ми повинні управляти набагато більшою кількістю серверів, які можуть бути розподілені по всій земній кулі.

У цей момент засоби управління конфігураціями і вступають в гру. У багатьох випадках, ми управляємо групами однакових серверів, на яких запущені однакові додатки і сервіси. Вони розміщуються на системах віртуалізації всередині організації, або ж запускаються як “хмарні” і гостьові в віддалених ЦОД. У деяких випадках, ми можемо говорити про велику кількість обладнання, яке існує тільки для підтримки дуже великих додатків або про обладнання, що обслуговує міріади невеликих сервісів. У будь-якому випадку, можливість “змахнути чарівною паличкою” і змусити їх усіх виконати волю системного адміністратора не може бути знецінена. Це єдиний шлях управляти величезними і зростаючими інфраструктурами.

Puppet, Chef, Ansible і Salt були задумані щоб спростити настройку та обслуговування десятків, сотень і навіть тисяч серверів. Це не означає, що маленькі компанії не отримають вигоди від цих інструментів, так як автоматизація зазвичай робить життя простіше в інфраструктурі будь-якого розміру.

Пильно глянувши на кожен з цих чотирьох інструментів, дослідивши їх дизайн і функціональність, можна зробити висновок, що незважаючи на те, що деякі інструменти оцінені вище, ніж інші, для кожного є своє місце, в залежності від цілей впровадження.

Puppet Enterprise. Puppet вважається найбільш використовуваним з чотирьох. Він найбільш повний з точки зору можливих дій, модулів і призначених для користувача інтерфейсів, представляючи повну картину ЦОД, охоплюючи майже кожну операційну систему і надаючи утиліти для всіх основних ОС. Початкова установка відносно проста, вимагає розгортання головного сервера і клієнтських агентів на кожній керованій системі.

Інтерфейс командного рядка дозволяє завантажувати і встановлювати модулі за допомогою команди puppet. Потім потрібні зміни в конфігураційних файлах, необхідні для налаштування модуля під потрібні завдання, та клієнти, які повинні отримати інструкції, отримають їх при наступному зверненні до головного сервера, або через запит від сервера, який ініціює процес зміни негайно.

Також є модулі, за допомогою яких виконується налаштування віртуальних і розміщених в «хмарах» серверів. Всі модулі та конфігурації будуються за допомогою мови програмування ruby. Отже, для роботи з Puppet потрібні деякі знання програмування, на додаток до навичок системного адміністрування.

У Puppet Enterprise найбільш повний веб-інтерфейс з усіх, що дозволяє контролювати керовані вузли в реальному часі за допомогою попередньо створених модулів і cookbooks, розміщених на головних серверах. Вебінтерфейс хороший для управління, але не дозволяє проводити «тонке» налаштування модулів. Інструменти для побудови звітів добре розроблені, надаючи глибоку деталізацію про поведінку агентів і про внесені зміни.

Enterprise Chef. Chef схожий на Puppet з точки зору загальної концепції, в ньому також є головний сервер і агенти, встановлені на керованих вузлах. На додаток до головного сервера, установка Chef також вимагає робочої станції, для управління ним. Агенти можуть бути встановлені з робочої станції за допомогою утиліти knife, яка використовує протокол SSH для розгортання, полегшуючи тягар установки. Після цього, керовані вузли автентифіковані з головним за допомогою сертифікатів.

Конфігурація Chef тісно пов’язана з системою управління версіями Git, тому знання того, як працює Git необхідні для роботи. Також як і Puppet, Chef написаний на ruby, тому буде потрібно і знання цієї мови, як і у випадку з Puppet. Модулі можуть бути завантажені або написані «з нуля», після чого встановлені на керовані вузли, відповідно до необхідних налаштувань.

На відміну від Puppet, у Chef поки немає добре реалізованої функції push, хоча доступна бета-версія коду. Це означає, що агенти повинні бути налаштовані на періодичну синхронізацію з головним сервером, і негайне застосування змін неможливо.

Призначений для користувача вебінтерфейс функціональний, але не надає можливості модифікувати конфігурації. Він не так сповнений, як веб-інтерфейс Puppet Enterprise, поступається в побудові звітів і деяких інших функціях, але дозволяє вести облік обладнання та організацію вузлів.

Як і у Puppet, у Chef великий набір модулів і рецептів налаштувань, переважно на ruby. З цієї причини, Chef добре підходить для інфраструктур, орієнтованих на розробку.

Ansible. Ansible більше схожий на Salt, ніж на Puppet або Chef. Ansible фокусується на оптимізації та швидкості, і не вимагає установки агентів на керовані вузли – всі функції здійснюються за допомогою SSH. Ansible написаний на python, на відміну від Puppet і Chef, написаних на ruby.

Установка Ansible може бути виконана шляхом клонування Git-сховища на головний сервер. Слідом за цим, вузли, над якими потрібно керувати додаються в конфігурацію Ansible, і авторизовані ключі SSH "прив'язуються" до кожного вузла, звертаючись до користувача від імені якого буде запускатися Ansible. Як тільки це зроблено, головний сервер може з'єднуватися з вузлами по протоколу SSH і виконувати всі необхідні завдання. Для роботи з системами, не дозволяє доступ з правами суперкористувача (root) по SSH, Ansible використовує облікові дані, що дозволяють виконувати дії від імені суперкористувача за допомогою команди sudo.

Ansible може використовувати Paramiko – реалізацію протоколу SSH2 мовою python, або стандартну реалізацію SSH, але є також прискорений режим, для більш швидкої і широкомасштабної комунікації.

Ansible може бути запущений з командного рядка без використання конфігураційних файлів для простих завдань, таких як перевірка, що якийсь сервіс запущений, або для поновлення тригерів і перезавантаження. Для більш комплексних завдань, конфігураційні файли створюються за допомогою YAML і називаються «сценарії» (playbook). У них можуть бути використані шаблони для розширення функціональності.

У Ansible є набір модулів, які можуть використовуватися для управління різними системами, так само як і "хмарними" інфраструктурами, такими як Amazon EC2 і OpenStack. Додаткові модулі можуть бути написані практично будь-якою мовою програмування, за умови, що висновок буде в форматі JSON.

Вебінтерфейс доступний у вигляді AnsibleWorks AWX, але безпосередньо не пов'язаний з інтерфейсом командного рядка. Це означає, що елементи конфігурації, задані через командний рядок, не з'являться в вебінтерфейсі до тих пір, поки не буде запущений цикл синхронізації. Ви можете використовувати вбудовану утиліту для цього, але необхідно запланувати її регулярний запуск. Сам по собі вебінтерфейс досить функціональний, але не такий повний, як інтерфейс командного рядка, таким чином ви або будете перемикатися між обома, або ж просто скористатися командним рядком.

SaltStack Enterprise. Salt схожий з Ansible в тому, працює через командний рядок. Він використовує метод push для зв'язку з клієнтами. Він може бути встановлений через Git або через систему управління пакетами на головному сервері і клієнтів. Клієнт робить запит до головного сервера, і якщо той дає дозвіл, дозволяє управляти даним вузлом за допомогою агента (в термінах Salt-minion).

Salt може зв'язуватися з клієнтами за протоколом SSH, але масштабованість значно розширюється за рахунок клієнтських агентів. Також, Salt включає асинхронний файловий сервер для прискорення обслуговування агентів, дозволяючи створювати добре масштабовані системи.

Як і в разі Ansible, ви можете віддавати команди, такі як як запуск сервісів або установка пакетів агентам за допомогою інтерпретатора, або можете використовувати конфігураційні файли в форматі YAML (state), для обробки комплексних завдань. Також є централізовано розміщені набори даних (pillar) до яких мають доступ конфігураційні файли під час роботи.

Ви можете запросити інформацію про конфігурацію, таку як версія ядра або детальну інформацію про мережевий інтерфейс, безпосередньо від агентів через командний рядок. Агенти можуть також задаватися через використання елементів інвентарю, званих "зернами" (grain), що дозволяють легко передавати команди на визначені сервери. Наприклад, однією командою можна відправити запит до агентів, що розташовані на серверах з певною версією ядра.

Як і попередні продукти, Salt надає велику кількість модулів для різноманітного програмного забезпечення, операційних систем і "хмарних" сервісів. Допоміжні модулі можуть бути написані мовами python або PyDSL. Salt надає можливість керувати і Windows-вузлами, так само як і Unix, але більше розрахований на системи Unix і Linux.

Вебінтерфейс Salt-Halite – занадто новий і не повний, як користувацькі інтерфейси інших систем. З його допомогою можна переглядати системні журнали повідомлень і статус керованих вузлів, а також є можливість виконувати на них команди. Цей інструмент активно розробляється, і обіцяє значні поліпшення, але поки це голий "скелет" і містить багато помилок.

Найбільша перевага Salt – масштабованість і гнучкість. У вас може бути кілька рівнів головних серверів, які організовують пов'язану структуру, для забезпечення розподілу навантаження і збільшення відмовостійкості. Головні сервера верхнього рівня можуть управляти нижчестоячими в ієрархії і їх підлеглими вузлами. Інша перевага – однорангова система обміну повідомленнями, що дозволяє підлеглим вузлів задавати питання головним, а ті можуть отримувати відповіді від інших серверів для завершення картини. Це може бути корисним, якщо дані для завершення налаштування вузла знаходяться в базі даних реального часу.

Puppet або Chef? Ansible або Salt? Тоді як Puppet і Chef орієнтовані на розробників, Salt і Ansible більше підходять для потреб системних адміністраторів. Простий інтерфейс і зручність використання Ansible підходять мисленню сисадмінів у компаніях з великим числом Unix і Linux систем. Ansible швидко і легко запускається “з контейнера”.

Salt найнадійніший з чотирьох і теж підійде системним адміністраторам. Добре, що масштабується і володіє достатнім функціоналом, лише вебінтерфейс залишає бажати кращого.

Puppet найбільш зрілий і, ймовірно, найдоступніший з чотирьох продуктів, з точки зору зручності використання, хоча і настійно рекомендуються початкові знання ruby. Puppet не так оптимізований як Ansible або Salt, і його конфігурація часом може нагадувати “фількину грамоту”. Puppet найбільш безпечний для гетерогенного оточення, в той час як Ansible або Salt більш безпечний для великих або більш гомогенних інфраструктур.

Chef стабільне і ретельно відпрацьоване рішення, але поки не дотягує до рівня Puppet, з точки зору основних функцій, проте його можна розширити. Chef може бути важким для вивчення адміністраторами з недостатнім досвідом в програмуванні, але може підійти компаніям, які займаються розробкою ПЗ.

2.3. Робота з інструментом управління конфігурацією Ansible

Систему Ansible часто описують як засіб управління конфігураціями, і зазвичай вона згадується в тому ж контексті, що і Chef, Puppet і Salt. Коли ми говоримо про управління конфігураціями, то часто маємо на увазі якийсь описовий стан серверів, а потім фіксацію їх реального стану з використанням спеціальних засобів: необхідні пакети додатків встановлені, файли конфігурації містять очікувані значення і мають необхідні дозволи в файловій системі, необхідні служби працюють і т. д. Подібно іншим засобам управління, Ansible представляє предметно-орієнтовану мову (Domain Specific Language, DSL), яка використовується для опису станів серверів.

Ці інструменти також можна використовувати для розгортання програмного забезпечення. Під розгортанням ми часто маємо на увазі процес отримання двійкового коду з вихідного, копіювання необхідних файлів на сервер і запуск служб. Capistrano і Fabric – два приклади інструментів з відкритим кодом для розгортання додатків. Ansible теж є чудовим інструментом як для розгортання, так і для управління конфігураціями програмного забезпечення. Використання єдиної системи управління конфігураціями і розгортанням значно спрощує життя системним адміністраторам.

Деякі фахівці наголошують на необхідності узгодження розгортання, коли в процес залучено кілька віддалених серверів і операції повинні

здійснюватися в певному порядку. Наприклад, базу даних потрібно встановити до установки вебсерверів або виводити вебсервери з під управління балансувальника навантаження тільки по одному, щоб система не припиняла роботу під час оновлення. Система Ansible хороша і в цьому, оскільки спочатку створювалася для проведення маніпуляцій відразу на декількох серверах. Ansible має дивно просту модель управління порядком дій.

Нарешті, ви почуєте, як люди говорять про ініціалізації (provisioning) нових серверів. В контексті хмарних послуг, таких як Amazon EC2, під ініціалізацією мається на увазі розгортання нового екземпляра віртуальної машини. Ansible охоплює і цю область, надаючи кілька модулів підтримки хмар, включаючи EC2, Azure, Digital Ocean, Google Compute Engine, Linode і Rackspace, а також будь-які хмари, що підтримують OpenStack API.

2.3.1. Як працює Ansible

На рис. 2.9 показаний простий приклад використання Ansible. Користувач, якого ми назвемо Стейсі, застосовує Ansible для налаштування трьох вебсерверів Nginx, що діють під управлінням Ubuntu. Вона написала для Ansible сценарій webservers.yml. У термінології Ansible сценарії називаються playbook. Сценарій описує, які хости (Ansible називає їх віддаленими серверами) підлягають настройці і упорядкований список завдань, які мають бути виконані на цих хостах. У цьому прикладі хости носять імена web1, web2 і web3, і для настройки кожного з них потрібно виконати наступні завдання:

- встановити Nginx;
- згенерувати файли конфігурації для Nginx;
- скопіювати сертифікат безпеки;
- запустити Nginx.

Стейсі запускає сценарій командою ansible-playbook. У прикладі сценарій називається webservers.yml і запускається командою

$ ansible-playbook webservers.yml

Ansible встановлює паралельні SSH-з'єднання з хостами web1, web2 і web3. Виконує першу задачу зі списку на всіх хостах одночасно. У цьому прикладі перша задача – встановлення apt-пакета Nginx (оскільки Ubuntu використовує диспетчер пакетів apt). Тобто дана задача в сценарії виглядає приблизно так:

name: install nginx
apt: name=ngtnx

Виконуючи її, Ansible виконає наступні дії:

- згенерує сценарій мовою Python, який встановить пакет Nginx;
- скопіює його на хости web1, web2 і web3;
- запустить на хостах web1, web2 і web3;

– дочекається, поки сценарій буде завершено на всіх хостах.

Далі Ansible приступить до наступної задачі в списку і повторить описані ці ж чотири кроки. Важливо відзначити, що:

– кожне завдання виконується на всіх хостах одночасно;

– Ansible очікує, поки завдання буде завершено на всіх хостах, перш ніж приступити до виконання наступного;

– завдання виконуються в установленому вами порядку.

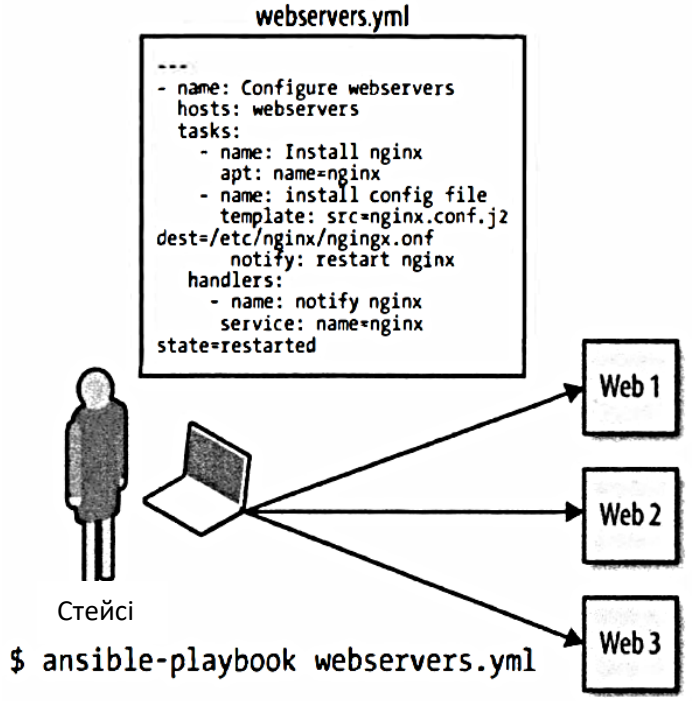


Рис. 2.9. Ansible виконує сценарій налаштування трьох вебсерверів

Які переваги дає ANSIBLE?

Існує кілька відкритих систем управління конфігураціями. Нижче перераховуються деякі особливості, котрі залучили нашу увагу до Ansible.

Простота синтаксису

Нагадаю, що завдання управління конфігураціями в Ansible визначаються у вигляді сценаріїв (playbooks). Синтаксис сценаріїв Ansible заснований на YAML, мові опису даних, яка створювалася спеціально, щоб легко сприйматися людиною. У деякому роді YAML для JSON – те ж, що Markdown для HTML.

Сценарії Ansible, як документація, що виконується. Вони на кшталт файлів README, які описують дії, необхідні для розгортання програмного забезпечення, але, на відміну від них, сценарії завжди містять актуальні інструкції, оскільки самі є виконувальним кодом.

Відсутність необхідності установки на віддалених хостах

Для управління серверами за допомогою Ansible на них повинна бути встановлена підтримка SSH і Python версії 2.5 або вище або Python 2.4 з

бібліотекою simplejson. Немає ніякої необхідності встановлювати на хостах будь-яке інше програмне забезпечення.

На керуючій машині (тій, що ви використовуєте для управління віддаленими машинами) повинен бути встановлений Python версії 2.6 або вище.

Заснований на технології примусового налаштування

Деякі системи управління конфігураціями, що використовують агентів, такі як Chef і Puppet, за замовчуванням засновані на технології добровільного налаштування. Агенти, встановлені на серверах, періодично підключаються до центральної службі і читають інформацію про конфігурацію. Управління змінами конфігурації серверів в цьому випадку виглядає так:

1. Ви: вносите зміни в сценарій управління конфігураціями;
2. Ви: передаєте зміни центральній службі;
3. Агент на сервері: періодично включається по таймеру;
4. Агент на сервері: підключається до центральної служби;
5. Агент на сервері: читає нові сценарії управління конфігураціями;
6. Агент на сервері: запускає отримані сценарії локально, оновлюючи стан сервера.

Ansible, навпаки, за замовчуванням використовує технологію примусового налаштування. Внесення змін виглядає так:

1. Ви: вносите зміни в сценарій;
2. Ви: запускаєте новий сценарій;
3. Ansible: підключається до серверів і запускає модулі, оновлюючи стан серверів.

Як тільки ви запустите команду ansible-playbook, Ansible підключиться до віддалених серверів і виконає всю роботу.

Примусове налаштування дає важливу перевагу – ви контролюєте час оновлення серверів. Вам не доводиться чекати. Прихильники добровільного налаштування стверджують, що їх підхід краще масштабується на більше число серверів і зручніше, коли нові сервери можуть з’явитися в будь-який момент. Однак відкладемо цю дискусію на потім, а поки відзначимо, що Ansible з успіхом використовувався для управління тисячами вузлів і показав відмінні результати в мережах з серверами, які динамічно додаються і видаляються.

Якщо вам дійсно подобається модель, заснована на прийомах добровільного налаштування, для вас Ansible офіційно підтримує особливий режим, званий Ansible -pull. Ви можете дізнатися більше про це з офіційної документації:

http: //docs.ansiИe.com/ansiЬle/playbooks_intro.html#ansible-pull.

Управління невеликим числом серверів

Так, Ansible можна використовувати для управління сотнями і навіть тисячами вузлів. Але керувати єдиним вузлом за допомогою Ansible також дуже легко – вам потрібно лише написати один сценарій. Ansible підтверджує принцип Алана Кея: “Просте має залишатися простим, а складне – можливим”.

Вбудовані модулі

Ansible можна використовувати для виконання довільних команд оболонки на віддалених серверах, але його дійсно потужною стороною є набір модулів. Модулі необхідні для виконання таких завдань, як встановлення пакетів програм, перезапуск служби або копіювання файлів конфігурації.

Як ми побачимо пізніше, модулі Ansible несуть декларативну функцію і використовуються для опису необхідного стану серверів. Наприклад, ви могли б викликати модуль user, щоб переконатися в існуванні облікового запису deploy в групі web:

user: name=deploy group=web

Модулі також є ідемпотентними. Якщо користувач deploy не існує, Ansible створить його. Якщо він існує, Ansible просто перейде до наступного кроку. Тобто сценарії Ansible можна запускати на сервері багато разів. Це велике удосконалення, в порівнянні з підходом на основі сценаріїв командної оболонки, тому що повторний запуск таких сценаріїв може привести до незапланованих і добре, якщо нешкідливих наслідків.

Дуже тонкий шар абстракції

Деякі системи управління конфігураціями надають рівень абстракції настільки потужний, що дозволяють використовувати одні і ті ж сценарії для управління серверами з різними операційними системами. Наприклад, замість конкретних диспетчерів пакетів, таких як yum або apt, можна використовувати абстракцію “пакет”, що підтримується системою управління конфігураціями.

Ansible працює не так – для установки пакетів у системи, які засновані на диспетчері apt, ви повинні використовувати диспетчер apt, а в системах, заснованих на диспетчері yum – диспетчер yum.

На практиці це спрощує використання Ansible, хоча на перший погляд може здатися недоліком. Ansible не вимагає вивчення нових наборів абстракцій, що нівелюють різницю між операційними системами. Це скорочує обсяг документації для вивчення перед початком написання сценаріїв.

При бажанні ви можете писати власні сценарії Ansible для виконання певних дій, в залежності від операційної системи на віддаленому сервері.

Модуль є основною одиницею повторного використання в спільноті Ansible. Оскільки область застосування модуля обмежена і залежить від певної операційної системи, це дозволяє писати якісні і надійно працюючі модулі. Проект Ansible завжди відкритий для нових модулів, пропонованих спільнотою.

Не занадто проста система Ansible?

Плануючи перехід з іншої системи управління конфігураціями на Ansible, дійсно можуть виникнути сумніви в його ефективності.

Однак, як скоро буде показано, Ansible має набагато ширшу функціональність, ніж сценарії командної оболонки. Як уже згадувалося, модулі Ansible гарантують ідемпотентність, Ansible має чудову підтримку шаблонів і змінних з різними областями видимості. Будь-хто, хто вважає, що суть Ansible полягає в роботі зі сценаріями командної оболонки, ніколи не займався підтримкою нетривіальних програм оболонки.

А як щодо масштабованості SSH? Ansible застосовує SSH-мультиплексування для оптимізації продуктивності. Деякі фахівці використовують Ansible для управління тисячами вузлів.

Що я повинен знати?

Для ефективної роботи з Ansible необхідно знати основи адміністрування операційної системи Linux. Ansible дозволяє автоматизувати процеси, але не виконує чарівним чином ті з них, з якими ви не справляєтеся.

Читачі цього посібника повинні бути знайомі, по крайній мірі, з одним з дистрибутивів Linux (Ubuntu, RHEL / CentOS, SUSE і ін.) і розуміти, як:

- підключитися до віддаленої машині через SSH;
- працювати в командному рядку Bash (канали та перенаправлення);
- встановлювати пакети додатків;
- використовувати команду sudo;
- перевіряти і встановлювати дозволи для файлів;
- запускати і зупиняти служби;
- встановлювати змінні середовища;
- писати сценарії (будь-якою мовою).

Якщо все це вам відомо, можете сміливо приступати до роботи з Ansible.

Ansible використовує формат файлів YAML і мову шаблонів Jinja2. Відповідно, вам необхідно вивчити їх, але обидві технології прості в освоєнні.

2.3.2. Встановлення ANSIBLE

На сьогоднішній день всі основні дистрибутиви Linux включають пакет Ansible. Тому, якщо ви працюєте в Linux, ви зможете встановити його, використовуючи “рідний” диспетчер пакетів. Але майте на увазі, що це може бути не сама остання версія Ansible. Якщо ви працюєте в MacOS X, рекомендовано використовувати чудовий диспетчер пакетів Homebrew.

Якщо такого пакета в вашій версії ОС немає, ви можете встановити Ansible за допомогою pip, диспетчера пакетів Python, виконавши наступну команду:

$ sudo pip install ansible

При бажанні Ansible можна встановити в локальне віртуальне оточення Python (virtualenv). Якщо ви не знайомі з віртуальними оточеннями, можете використовувати більш новий інструмент під назвою pipsi. він автоматично створить нове віртуальне оточення і встановить в нього Ansible:

$ wget https: //raw.githubusercontent.co/itsuhiko/pipsi/aster/get-pipsi.py

$ python get-pipsi.py

$ pipsi tnstall ansible

Якщо ви вирішите скористатися pipsi, додайте шлях *~/.local/bin* в змінну оточення PATH. Деякі плагіни та модулі Ansible можуть вимагати установки додаткових бібліотек Python. Якщо ви зробили установку за допомогою pipsi і хотіли б встановити docker (необхідний для модулів з бібліотеки Ansible Docker) і boto (необхідний для модулів з бібліотеки Ansible EC2), виконайте наступні команди:

$ cd - /. Local / venvs / ansible

$ source in / activate

$ pip tnstall docker-py boto

Якщо вам цікаво випробувати в роботі нову версію Ansible, завантажте її з GitHub:

$ git clone https://github.co/ansible/ansible.git -recursive

При роботі з цією версією вам кожен раз буде потрібно виконати наступні команди, щоб встановити змінні оточення, включаючи змінну PATH, щоб оболонка змогла знаходити програми ansible і ansible-playbooks.

$ cd ./ansible

$ source ./hacking/env-setup

Підготовка сервера для експериментів

Для виконання прикладів, наведених в посібнику, вам необхідно мати SSH доступ і права користувача root на сервері Linux. На щастя, сьогодні

легко отримати недорогий доступ до віртуальної машини Linux в загальнодоступних службах хмарних послуг, таких як Amazon EC2, Google Compute Engine, Microsoft Azure, Digital Ocean, Linode.

Передача інформації про сервер в Ansible

Ansible може управляти тільки відомими йому серверами. Передати інформацію про сервери в Ansible можна в файлі реєстру.

Кожному серверу повинно бути присвоєно ім’я для ідентифікації в Ansible. З цією метою можна використовувати ім'я хоста або вибрати інший псевдонім. З ім’ям також повинні визначатися додаткові параметри подключення. Дамо нашому серверу псевдонім testserver.

Створіть в каталозі playbooks файл з ім’ям hosts. Він буде служити реєстром. Якщо припустити, що у вас є Ubuntu-машина в хмарі Amazon EC2 з ім’ям хоста ec2-203-0-113-120.compute-1.amazonaws.com, вміст файла реєстру буде виглядати так (все в один рядок):

testserver ansible_host=ec2-203-0-113-120.compute-1.amazonaws.com \ansible_user=ubuntu ansible_private_key_file=/path/to/keyfile.pem

Щоб перевірити здатність Ansible підключитися до сервера, використовуємо утиліту командного рядка Ansible. Ми будемо зрідка користуватися нею, в основному для вирішення специфічних завдань.

Попросимо Ansible встановити з’єднання з сервером testserver, зазначеним в файлі реєстру hosts, і викликати модуль ping:

$ ansible testserver -i hosts -m ping

Якщо на локальному SSH-клієнті включена перевірка ключів хоста, ви побачите щось, схоже на першу спробу Ansible підключитися до сервера:

The authenttcity of host '[127.0.0.1):2222 ([127.0.0.1):2222)' \ can't be established. RSA key fingerprint ts e8:0d:7d:ef:57:07:81:98:40:31:19:53:a8:d0:76:21. Аге you shure you want to continue connecttng (yes/no)?

Просто введіть yes.

У разі успіху з’явиться наступний результат:

testserver I success >> {

“changed”: false, “ping”: “pong”

}

Ми бачимо, що команда виконалася успішно. Частина відповіді “changed”: false говорить про те, що виконання модуля не змінило стану сервера. Текст “Ping”: “pong” є характерною особливістю модуля ping.

Модуль ping не виробляє ніяких змін. Він лише перевіряє здатність Ansible почати SSH-сеанс з сервером.

Спрощення завдання за допомогою файлу ansible.cfg

Нам довелося ввести багато тексту в файл реєстру, щоб повідомити системі Ansible інформацію про тестовий сервер. На щастя, Ansible підтримує декілька способів передачі такої інформації, і ми не зобов'язані групувати її в одному місці. Зараз ми скористаємося одним з таких способів – файлом ansible.cfg для визначення деяких налаштувань за замовчуванням, щоб потім нам не довелося набирати так багато тексту.

Приклад 2.1 показує, як у файлі ansible.cfg визначаються позиціонування файлу реєстру (inventory), ім'я користувача SSH (remote_user) і приватний ключ SSH (private_key_file). Ці налаштування припускають використання Vagrant. При використанні окремого сервера необхідно встановити тільки значення remote_user і private_key_file.

У нашому прикладі конфігурації, перевірка SSH-ключів хоста відключена. Це зручно при роботі з Vagrant. В іншому випадку необхідно вносити зміни в файл ~/.ssh/known_hosts кожен раз, коли видаляється наявний або створюється новий Vagrant-сервер. Однак відключення перевірки ключів для серверів у мережі несе певні ризики.

Приклад 2.1 ansible.cfg

[defaults] inventoгy = hosts

remote_user = vagrant

private_key_file = .vagrant/machines/default/virtualbox/prtvate_key

host_key_checking = False

З налаштуваннями за замовчуванням відпадає необхідність вказувати ім'я користувача або файл з ключами SSH в файлі hosts. Запис спрощується до: *testserver ansible_ host = 127.0.0.1 ansible_ port = 2222*

Ми також можемо запустити Ansible без аргументу *-i. hostname:*

$ *Ansible testserver -m ping*

Чудово використовувати інструмент командного рядка ansible для запуску довільних команд на віддалених серверах. Довільні команди також можна виконувати за допомогою модуля ***comand***. При запуску модуля необхідно вказати аргумент -a з запускаємою командою.

Наприклад, ось як можна перевірити час роботи сервера з моменту останнього запуску:

$ ansible testserver -m command -a uptime

Результат повинен виглядати приблизно так:

testserver | success | rc = 0 >>

17:14:07 up 1:16, 1 user, load average: 0.16, 0.05, 0.04

Модуль command настільки часто використовується, що зроблений модулем стандартно за замовчуванням, тобто його ім’я можна пропустити в команді:

$ ansible testserver -a uptime

Якщо команда в аргументі -a містить прогалини, її необхідно взяти в лапки, щоб командна оболонка передала весь рядок як єдиний аргумент. Для прикладу ось як виглядає витяг кількох останніх рядків з журналу /var/log/dmesg:

$ ansible testserver -a “tail/var/log/dmesg”

Виведення, що повертається машиною Vagrant, виглядає наступним чином:

testserver I success I rc=0 >>

[5.170544] type=1400 audit(1409500641.335:9): apparmor="STATUS" operation=

"profile_replace" profile="unconfined" name="/usr/li.b/NetworkManager/nm-dhcp-

client.act on" pid=888 comm="apparmor_parser"

[5.170547] type=1400 audit(1409500641.335:10): apparmor="STATUS" operation=

"profile_replace" profile="unconfined" name="/usr/li.b/connman/scripts/dhclientscript"

pid=888 comm="apparmor_parser"

[5.222366] vboxvi.deo: Unknown symbol drm_open (err 0)

[5.222370] vboxvideo: Unknown symbol drm_poll (err 0)

[5.222372] vboxvideo: Unknown symbol drm_pci_i.nit (err 0)

[5.222375] vboxvi.deo: Unknown symbol drm_i.octl (err 0)

[5.222376] vboxvi.deo: Unknown symbol drm_vBlank_i.ni.t (err 0)

[5.222378] vboxvi.deo: Unknown symbol drm_mmap (err 0)

[5.222380] vboxvi.deo: Unknown symbol drm_pci._exi.t (err 0)

[5.222381] vboxvi.deo: Unknown symbol drm_release (err 0)

Щоб виконати команду з привілеями root, потрібно передати параметр -b. В цьому випадку Ansible виконає команду від імені користувача root. Наприклад, для доступу до /var/log/syslog потрібні привілеї root:

$ ansible testserver -b -a "tail /var/log/syslog"

Результат буде виглядати приблизно так:

testserver I success I rc=0 >>

Aug 31 15:57:49 vagrant-ubuntu-trusty-64 ntpdate[1465]: /

adjust ti.me servers 91.189

94.4 offset -0.003191 sec

Aug 31 16:17:01 vagrant-ubuntu-trusty-64 CRON[1480]: (гоот)CMD(cd/ && run-p

rts --героґt /etc/cron.hourly)

Aug 31 17:04:18 vagrant-ubuntu-trusty-64 ansble-ping: Invoked with data=None

Aug 31 17:12:33 vagrant-ubuntu-tгusty-64 ansiЬIe-ping: Invoked with data=None

Aug 31 17:14:07 vagrant-ubuntu-trusty-64 ansiЬIe-соммand: Invoked with executable

None shell=False aгgs=uptiме reмoves=None creates=None chdir=None

Aug 31 17:16:01 vagrant-ubuntu-trusty-64 ansible-соммand: Invoked with executable

None shell=False args=tail /var/log/мessages reмoves=None creates=None chdir=None

Aug 31 17:17:01 vagrant-ubuntu-trusty-64 CRON(2091]: (гоot) CMD (cd / && run-pa rts --героґt /etc/cron.hourly)

Aug 31 17:17:09 vagrant-ubuntu-trusty-64 ansiЬIe-соммand: Invoked with / executaЬIe= N one shell=False args=tail /var/log/dмesg reмoves=None creates=None chdir=None

Aug 31 17:19:01 vagrant-ubuntu-trusty-64 ansiЬIe-соммand: Invoked with / executable= None shell=False aгgs=tail /var/log/мessages reмoves=None creates=None chdir=None

Aug 31 17:22:32 vagrant-ubuntu-trusty-64 ansiЬIe-соммand: Invoked with / executable= one shell=False args=tail /var/log/syslog reмoves=None creates=None chdir=None

Як бачите, Ansible фіксує свої дії в syslog.

Утиліта ansible не обмежується модулями ping і command: ви можете використовувати будь-який модуль за бажанням. Наприклад, за допомогою такої команди можна встановити Nginx в Ubuntu:

$ ansible testserver -b -m apt -a наме = nginx

Перезапустити Nginx можна так:

$ ansible testserver -b -m service -a "name = nginx\state = restarted "

Оскільки тільки користувач root може встановити пакет Nginx і перезапустити служби, необхідно вказати аргумент -b.

2.3.3. Сценарії: початок

Працюючи з Ansible, більшу частину часу ви будете приділяти написанню сценаріїв. Сценарієм в Ansible називається файл, що описує

порядок управління конфігураціями. Розглянемо, наприклад, установку вебсервера Nginx і його налаштування для підтримки захищених з'єднань.

До кінця цього розділу у вас повинні з'явитися такі файли:

- playbooks / ansible.cfg;
- playbooks / hosts;
- playbooks / Vagrantfile;
- playbooks / web-notls.yml;
- playbooks / web-tls.yml;
- playbooks / files / nginx.key;
- playbooks / files / nginx.crt;
- playbooks / files / nginx.conf;
- playbooks / templates / index.html.j2;
- playbooks / templates / nginx. conf.j2.

Підготовка

Перш ніж запустити сценарій на машині Vagrant, необхідно відкрити порти 80 і 443. Як показано на рис. 2.10, ми налаштуємо Vagrant так, щоб запити до портів 8080 і 8443 на локальній машині перенаправлялись портам 80 і 443 на машині Vagrant. Це дозволить отримати доступ до вебсервера, запущеного на Vagrant, за адресами http://localhost:8080 і https://localhost: 8443.

Змініть вміст Vagrantfile, як показано нижче:

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do lconfigl

confi.g.v111.box = "ubuntu/trusty64"

confi.g.v111.network "forwarded_port", guest: 80, host: 8080

confi.g.v111.network "forwarded_port", guest: 443, host: 8443 end

Ці налаштування відображать порти 8080 і 8443 локальної машини в порти 80 і 443 машини Vagrant. Після збереження змін дайте команду застосувати їх:

$ vagгant гeload

В результаті на екрані повинні з`явитися такі рядки:

==> default: Forwarding ports ...

default: 80 => 8080 (adapter 1)

default: 443 => 8443 (adapter 1)

default: 22 => 2 222 (adapter 1)

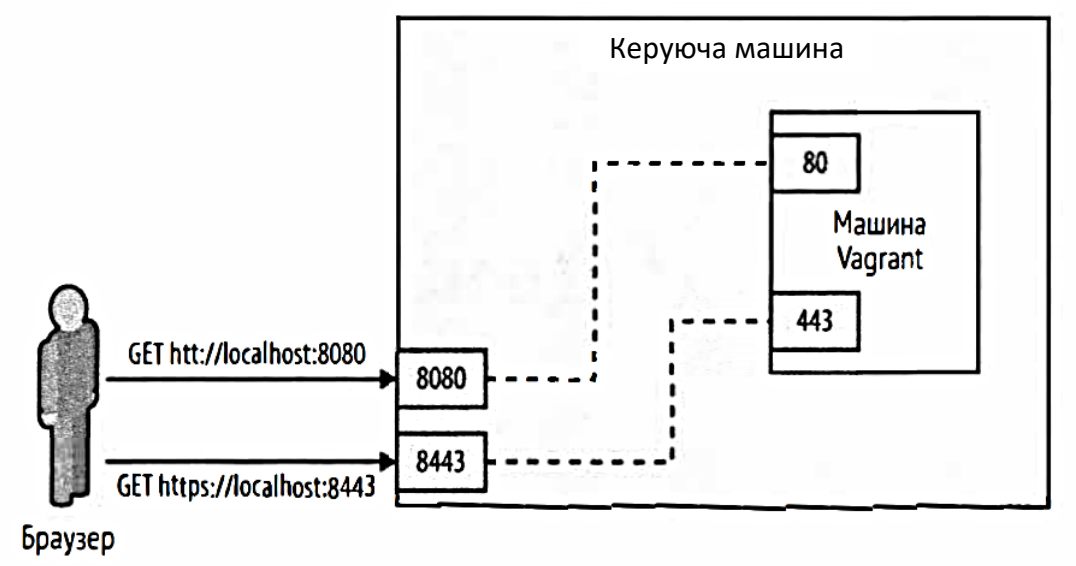


Рис. 2.10. Відкриття портів на машині Vagrant

Дуже простий сценарій

У нашому першому прикладі сценарію ми налаштуємо хост для запуску вебсервера Nginx. У цьому прикладі ми не будемо налаштовувати підтримку TLS-шифрування вебсервером. Це зробить установку простіше. Однак правильний вебсайт повинен підтримувати TLS-шифрування, і ми побачимо, як це зробити, далі в цьому розділі.

Спочатку подивимося, що вийде, якщо запустити сценарій з прикладу 2.2, а потім детально вивчимо його вміст.

Приклад 2.2. web-notls.yml

```
- name: Configure webserver with nginx
  hosts: webservers
  become: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/nginx.conf dest=/etc/nginx/sites-available/default

    - name: enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link

    - name: copy index.html
      template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
        mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

Файл конфігурації Nginx

Даному сценарію необхідні два додаткових файли. Спочатку створимо файл конфігурації Nginx.

Nginx поставляється з файлом конфігурації, готовим до використання лише для обслуговування статичних файлів. Але частіше його необхідно допрацьовувати під свої потреби. Тому ми змінимо файл конфігурації за замовчуванням в рамках даного прикладу. Як стане зрозуміло пізніше, ми повинні додати в файл конфігурації підтримку TLS. У прикладі 2.3 приводиться стандартний файл конфігурації Nginx. Збережіть його з ім'ям playbooks/files/nginx.conf.

Приклад 2.3. files/nginx.conf

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;
    root /usr/share/nginx/html;
    index index.html index.htm;

    server_name localhost;

    location / {
        try_files $uri $uri/ =404;
    }
}
```

Створення початкової сторінки

Додамо свою початкову сторінку. Використовуємо шаблони Ansible, щоб згенерувати файл. Збережіть файл з прикладу 2.4 в playbooks/templates/index.html.j2.

Приклад 2.4 playbooks/templates/index.html.j2

```
<html>
  <head>
    <title>Welcome to ansible</title>
  </head>
  <body>
    <h1>nginx, configured by Ansible</h1>
    <p>If you can see this, Ansible successfully installed nginx.</p>

    <p>{{ ansible_managed }}</p>
  </body>
</html>
```

У цьому шаблоні використовується спеціальна змінна Ansible ansible_managed. Обробляючи шаблон, Ansible замінить її інформацією про час створення файлу шаблону. На рис. 2.11 показаний скріншот веббраузера з створеною HTML-сторінкою.

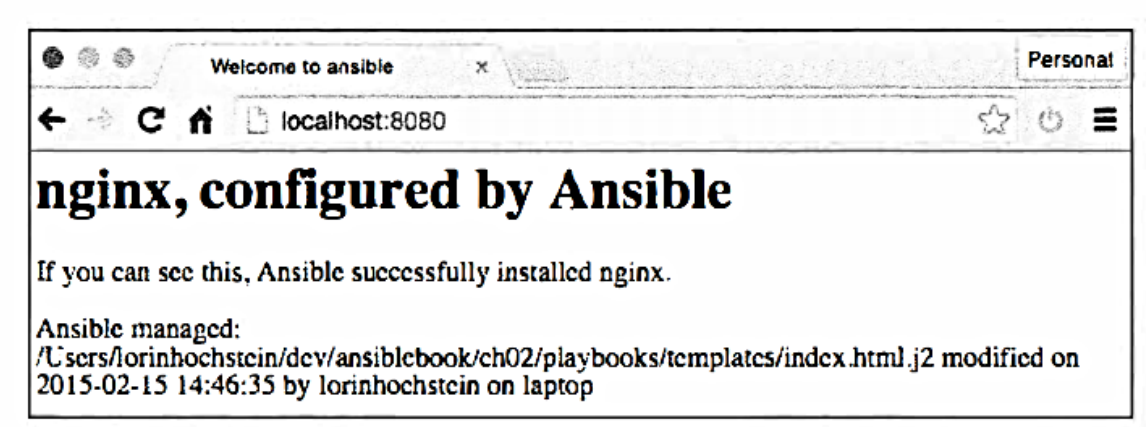


Рис. 2.11. Вид початкової сторінки, що була створена

Створення групи вебсерверів

Тепер створимо групу webservers в файлі реєстру, щоб отримати можливість послатися на неї в сценарії. Поки в цю групу увійде тільки наш тестовий сервер testserver.

Файли реєстру мають формат .ini. Детальніше цей формат ми розглянемо пізніше. Відкрийте файл playbooks/hosts в редакторі і додайте рядок [webservers] над рядком testserver, як показано в прикладі 2.5. Це означає, що testserver включений в групу webservers.

Приклад 2.5. playbooks/hosts

```
[Webservers]
testserver ansible_host = 127.0.0.1 ansible_port = 2222
```

Тепер можна спробувати виконати команду ping для групи webservers за допомогою утиліти ansible:

```
$ ansible webservers -m ping
```

Результат повинен виглядати так:

```
testserver | success >> {
"changed":false,
"ping":"pong",
}
```

Запуск сценарію

Сценарії виконуються командою ansible-playbook, наприклад:

```
$ ansible-playbook web-notls.yml
```

У прикладі 2.6 показано, як повинен виглядати результат.

Приклад 2.6. Результат запуску сценарію командою ansible-playbook

PLAY [Configure webserver with nginx] **********
GATHERING FACTS **********
ok: [testserver]
TASK: [tnstall nginx] **********
changed: [testserver]
TASK: [copy nginx config file] **********
changed: [testserver]
TASK: [enable configuration] **********
ok: [testserver]
TASK: [copy index.html] **********
changed: [testserver]
TASK: [restart nginx] **********
changed: [testserver]
PLAY RECAP **********
testserver ok=б changed=4 unreachable=0 failed=0

Якщо ви не зробили ніяких помилок, у вас повинно вийти відкрити в браузері сторінку http://localhost:8080. В результаті ви повинні побачити початкову сторінку, як показано на рис. 2.11.

Сценарії пишуть на YAML

Всі сценарії Ansible пишуться на YAML. YAML – це формат файла, схожий з JSON, але набагато простіше для сприйняття людиною. Перш ніж перейти до сценарію, розглянемо основні поняття YAML, найбільш важливі при написанні сценаріїв.

Початок файла

Файли YAML починаються з трьох дефісів, що позначають початок документа:

Однак Ansible не вважатиме за помилкою, якщо ви забудете вказати три дефіси на початку сценарію.

Коментарі

Коментарі починаються зі знаку “решітка” і тривають до кінця рядка, як в сценарії мовою командної оболонки, Python і Ruby:

 # Це коментар на мові YAML.

Рядки

Зазвичай рядки в YAML не беруться в лапки, навіть якщо вони включають прогалини. Хоча це не забороняється. Наприклад, ось рядок мовою YAML: *це приклад пропозиції.*

Аналог в JSON виглядає так:

"Це приклад пропозиції".

Іноді Ansible вимагає брати рядки в лапки. Зазвичай це рядки з фігурними дужками {{i}}, які використовуються для підстановки значень змінних. Але про це трохи пізніше.

Булеві вирази

У YAML є власний логічний тип. Він пропонує широкий вибір рядків, які можуть інтерпретуватися як “істина” і “брехня”.

Наприклад, ось булевий вираз на YAML:

True.

Аналог в JSON виглядає так:

true.

Списки

Списки в YAML схожі на масиви в JSON і Ruby або списки в Python. В YAML вони називаються послідовностями, але ми називатимемо їх списками, щоб уникнути протиріч з офіційною документацією Ansible.

Списки оформляються за допомогою дефіса:

- My Fair Lady

- Oklahoma

- The Pirates of Penzance

Аналог в JSON:

"My Fair Lady",

"Oklahoma",

"The Pirates of Penzance"

Ще раз зверніть увагу, що в YAML не потрібно брати рядки в лапки, навіть при наявності в них прогалин.

YAML також підтримує формат вбудованих списків. Він виглядає так:

[My Fair Lady, Oklahoma, The Pirates of Penzance]

Словники

Словники в YAML подібні об’єктам в JSON, словникам в Python або хеш-масивам в Ruby. Технічно в YAML вони називаються відображеннями, але ми називатимемо їх словниками, щоб уникнути протиріч з офіційною документацією Ansible.

Вони виглядають так:

address: 742 Evergreen Texace

ci.ty: Springfield

state: North Takoma

Аналог в JSON:

{

"Address": "742 Evergreen Texace",
"City": "Springfield",
"State": "North Takoma"
}

YAML також підтримує формат вбудованих словників:

{Address: 742 Evergreen Texace, city: Springfield, state: North Takoma}.

Об'єднання рядків

Під час написання сценаріїв часто виникають ситуації, коли необхідно передати модулю багато аргументів. В естетичних цілях їх можна помісити в кілька рядків у файлі. Однак при цьому необхідно, щоб Ansible сприймав їх як єдиний рядок.

У YAML для цього можна скористатися знаком "більше" (>). парсер YAML в цьому випадку замінить розриви рядків пробілами. Наприклад:

addгess:>
Departмent of Coмputer Science,
A.V. Williaмs Building,
University of Maryland
city: College Park
state: Maryland

Аналог в JSON:

{
"Addгess": "Departмent of Coмputeг Science, A. V. Williaмs Building, University of Maryland ",
"City": "College Paгk",
"State": "Maryland"
}

Структура сценарію

Розглянемо наш сценарій з точки зору YAML. У прикладі 2.6 він показується знову:

Приклад 2.6 web-notls.yml

- naмe: Configure webserver with nginx
hosts: webservers
become: Tгue
tasks:
name: install nginx
apt: name=nginx update_cache=yes
name: copy nginx config file

copy: src=files/nginx.conf dest=/etc/nginx/sites-availaЬIe/default
name: enable configuration
file: >
dest=/etc/nginx/sites-enabled/default
src=/etc/nginx/sites-available/default
state=link
name: copy index.html
template:src=templates/index.html.j2dest=/usr/share/nginx/html/index.html
mode=0644
name: restart nginx
service: name=nginx state=restarted

Операції

У будь-якому форматі YAML або JSON – сценарій є списком словників, або списком операцій.

Ось як виглядає операція з нашого прикладу:

naмe: Configure webserver with nginx
hosts: webservers
becoмe: Tгue
tasks:
naмe: install nginx
apt: naмe=nginx update_cache=yes
naмe: copy nginx config file
copy: src=files/nginx.conf dest=/etc/nginx/sites-availaЬIe/default
naмe: enaЬIe configuration
file: >
dest=/etc/nginx/sites-enaЬIed/default
src=/etc/nginx/sites-availaЬIe/default
state=Hnk
naмe: copy index.htмl
template:src=templates/index.htмl.j2 dest=/usr/share/nginx/htмl/index.htмl
мode=0644
naмe: restart nginx
service: naмe=nginx state=restarted

Кожна операція повинна містити:

- список конфігуруємих хостів;
- список завдань, які виконуються на цих хостах.

Сприймайте операцію як щось, що пов’язує хости і завдання.

Крім хостів і завдань, операції також можуть містити параметри. Ми розглянемо це питання пізніше, а зараз познайомимося з трьома основними параметрами: name, become, vars.

name

Коментар, який описує операцію. Ansible виведе його перед запуском операції.

become

Якщо має значення "істина", Ansible виконає кожну задачу, попередньо отримавши привілеї користувача root (за замовчуванням). Це може стати в нагоді для управління серверами UЬuntu, оскільки за замовчуванням ця система не дозволяє встановлювати SSH-з'єднання з привілеями root.

vars

Список змінних і значень. Ми побачимо призначення цього параметра пізніше в цьому розділі.

Завдання

Наш приклад сценарію містить одну операцію з п'ятьма завданнями. Ось перші завдання:

name: install nginx

apt: name = nginx update_cache = yes

Оскільки параметр name не є обов'язковим, завдання можна записати так:

- *apt: name = nginx update_cache = yes*

Навіть при тому, що імена завдань можна не вказувати, рекомендовано використовувати їх, оскільки вони служать хорошим нагадуванням їх цілей. Імена будуть особливо корисні для тих, хто спробує розібратися в вашому сценарії, в тому числі і вам через півроку. Як ми вже бачили, Ansible виводить ім'я завдання перед його запуском. Нарешті, можна також використовувати прапор --start-at-task <ім'я завдання>, щоб за допомогою ansible-p1aybook запустити сценарій з середини завдання. В цьому випадку необхідно послатися на завдання по імені.

Кожне завдання має містити ключ з назвою модуля і його аргументами. В даному прикладі модуль називається apt і приймає аргументи *name = nginx update_cache = yes*.

Ці аргументи повідомляють модулю apt встановити пакет nginx і оновити кеш пакетів (аналог команди apt-get update) перед установкою.

Важливо зрозуміти, що з точки зору парсера YAML, що використовується Ansible, аргументи сприймаються як рядки, а не

словники. Тобто, щоб розбити аргументи на кілька рядків, необхідно використовувати правило об'єднання рядків YAML:

name: install nginx

apt:>

name = nginx

update_cache = yes

Ansible підтримує також синтаксис, що дозволяє визначати аргументи модулів як словники YAML. Це може стати в нагоді при роботі з модулями, які мають складні аргументи.

Ansible підтримує також старий синтаксис, який використовує ключ *action* і записує ім'я модуля в значення. Наприклад, попередній приклад можна записати так:

name: install nginx

action: apt name = nginx update_cache = yes.

Модулі

Модулі – це сценарії, які поставляються з Ansible і виконують певні дії на хості. Правда, треба визнати, що це досить загальний опис, але серед модулів Ansible зустрічається безліч варіантів. В цьому розділі використовуються наступні модулі:

apt

Встановлює або видаляє пакети з використанням диспетчера пакетів apt.

copy

Копіює файл з локальної машини на хости.

file

Встановлює атрибути файла, символічного посилання або каталогу.

service

Запускає, зупиняє або перезапускає службу.

template

Створює файл на основі шаблону і копіює його на хости.

Як розповідалося раніше, Ansible виконує завдання на хості, генеруючи сценарій, виходячи з імені модуля і його аргументів, а потім копіює його на хост і запускає.

До складу Ansible входить понад 200 модулів, і їх число зростає з кожною новою версією. Також можна знайти модулі, написані сторонніми розробниками, або написати свої власні.

Є зміни? Відстеження стану хостів

Коли ви запускаєте команду ansible-playbook, вона виводить інформацію про стани кожного завдання, що виконується в рамках операції.

Поверніться до прикладу 2.6 і зверніть увагу, що стан деяких задач вказано як changed (змінено), а інших – ok. Наприклад, завдання install nginx має статус changed.

TASK: [lnstall nglnx] **********

changed: [testserver]

З іншого боку, завдання enable configuration має статус ok.

TASK: [enable configuration] **********

ok: [testserver]

Будь-яка запущена задача потенційно може змінити стан хоста. Перед тим як здійснити будь-яку дію, модулі перевіряють, чи потрібні зміни стану хоста. Якщо стан хоста відповідає значенням аргументів модуля, Ansible не робить ніяких дій і повідомляє, що статус ok.

Якщо між станом хоста і значеннями аргументів модуля є різниця, Anssible вносить зміни в стан хоста і повідомляє, що статус був змінений (changed).

Як показано в прикладі вище, salfxf insta11 nginx внесла зміни, а це значить, що до запуску сценарію пакет nginx встановлено. Завдання enable configuration не внесла змін, значить, на сервері вже був збережений файл конфігурації і він ідентичний тому, який ми копіювали. Причина у тому, що файл nginx.conf, який використовувався в сценарії, ідентичний файлу nginx.conf, який встановлюється з пакета nginx в UЬuntu.

Пізніше в цьому розділі ми побачимо, що здатність Ansible визначати зміни стану можна використовувати для виконання додаткових дій за допомогою обробників. Але навіть без оброблювачів корисно мати в своєму розпорядженні інформацію про зміну стану хостів в результаті виконання сценарію.

Підтримка TLS

Тепер розглянемо більш складний приклад. Додамо в попередній сценарій налаштування підтримки TLS вебсервером. Для цього нам знадобляться наступні нові елементи:

– змінні;
– обробники.

У прикладі 2.7 приводиться наш сценарій з включеним налаштуванням підтримки TLS.

Приклад 2.7. web-tls.yml

```
name: Configure webserver with nginx and tls
    hosts: webservers
    become: True
    vars:
```

```
        key_file: /etc/nginx/ssl/nginx.key
        cert_file: /etc/nginx/ssl/nginx.crt
        conf_file: /etc/nginx/sites-availaЬIe/default
        server_naмe: localhost
    tasks:
    naмe: Install nginx
    apt: name=nginx update_cache=yes cache_valid_tiмe=3600
    naмe: create directories for ssl certificates
    file: path=/etc/nginx/ssl state=directory
    name: copy TLS key
    copy: src=files/nginx.key dest={{ key_file }} owner=root
мode=0600
    notify: restart nginx
    name: copy TLS certificate
    copy: src=files/nginx.crt dest={{ cert_file }}
    notify: restart nginx
    name: copy nginx config file
    template: src=teмplates/nginx.conf.j2 dest={{ conf_file }}
    notify: restart nginx
    name: enable configuration
    file: dest=/etc/nginx/sites-enaЬIed/default src={{ conf_file }}
state=link
    notify: restart nginx
    name: copy index.html
    template:src=templates/index.html.j2
dest=/usr/share/nginx/htl'll/index.html
    mode=0644
handlers:
    name: restart nginx
    service: name=nginx state=restarted
```

Створення сертифіката TLS

Ми повинні вручну створити сертифікат TLS. Для промислової експлуатації сертифікат TLS необхідно придбати в центрі сертифікації або використовувати безкоштовну службу, таку як Let's Encrypt, яка підтримується в Ansible за допомогою модуля letsencrypt. Ми використовуємо “самоподпісанний” (self-signed) сертифікат, оскільки його можна створити безкоштовно.

Створіть підкаталог files в каталозі playbooks, а потім сертифікат TLS і ключ:

$ mkdir files

*$ openssl req -x509 -nodes -days 3650 -newkey rsa:2048 *

*-subj /CN=localhost *

-keyout files/nginx.key -out files/nginx.crt

Ця пара команд створить файли nginx.key і nginx.crt в каталозі files. Термін дії сертифіката обмежений 10 роками (3650 днів) з дня його створення.

Змінні

Тепер операція в нашому сценарії включає розділ vars:

vars:

key_file: /etc/nginx/ssl/nginx.key

cert_file: /etc/nginx/ssl/nginx.crt

conf_file: /etc/nginx/sites-available/default

server_name: localhost

Цей розділ визначає чотири змінні і їх значення.

У нашому прикладі кожне значення – це рядок (наприклад, /etc/nginx/ssl/nginx.key), але взагалі значенням змінної може служити будь-який вираз допустимий в YAML. На додаток до рядків і булевих виразів можна використовувати списки і словники.

Змінні можна використовувати в задачах і в файлах шаблонів. Для посилання на змінні використовуються дужки {{i}}. Ansible замінить дужки значенням змінної.

Припустимо, що в сценарії є наступне завдання:

name: copy TLS key

copy: src=files/nginx.key dest={{key_file}} owner=root mode=0600

При виконанні завдання Ansible замінить {{key_fi.le}} на /etc/ngi..nx/ssl/ngi.nx.key.

2.3.4. Створення шаблону з конфігурацією Nginx

Якщо ви займалися вебпрограмуванням, то, ймовірно, стикалися з системою шаблонів для створення розмітки HTML. Якщо ні, то поясню, що шаблон – це простий текстовий файл, в якому з використанням спеціального синтаксису, визначаються змінні, які повинні замінюватися фактичними значеннями. Якщо ви коли-небудь отримували автоматично згенерованого електронного листа від будь-якої компанії, то напевно помітили, що в листі використовується шаблон, аналогічний наведеному в прикладі 2.8.

Приклад 2.8. Шаблон електронного листа

Dear {{name}},

You have {{num_coments}} new coments on your blog: {{blog_name}}.

У випадку з Ansible це не HTM-сторінки або електронні листи, а файли конфігурації. Якщо можна уникнути редагування файлів конфігурації вручну, краще так і зробити. Це особливо корисно, якщо використовуються одні й ті ж конфігураційні дані (наприклад, IP-адреса сервера черги або облікові відомості для бази даних) в декількох файлах. Набагато розумніше помістити інформацію про конкретне оточення в одному місці, а потім створювати всі файли, що вимагають цієї інформації, на основі шаблону.

Для підтримки шаблонів, Ansible використовує механізм Jinja2. Якщо ви коли-небудь користувалися бібліотеками шаблонів, такими як Mustache, ERB або Django, тоді Jinja2 здасться вам знайомим інструментом.

У файл конфігурації Nginx необхідно додати інформацію про місце зберігання ключа і сертифіката TLS. Щоб виключити використання жорстко заданих значень, які можуть змінюватися з часом, ми скористаємося підтримкою шаблонів в Ansible.

В каталозі playbooks створіть підкаталог templates і файл templates/nginx.conf.j2, як показано в прикладі 2.9.

Приклад 2.9. templates/nginx.conf.j2

```
server {
        listen 80 default_server;
        listen [::]: 80 default_server ipv6only = on;
        listen 443 ssl;
        root / usr / share / nginx / html;
        index index.html index.htm;
        server_name {{server_name}};
        ssl_certificate {{cert_file}};
        ssl_certificate_key {{key_file}};
        location / {
                try_files $ uri $ uri / = 404;
        }
}
```

Ми використовуємо розширення файла .j2, щоб показати, що файл є шаблоном Jinja2. Однак ви можете використовувати будь-яке інше розширення. Для Ansible це неважливо.

У нашому шаблоні використовуються три змінні:

– server_name – назва хоста вебсервера (наприклад, www.example.com);

– cert_file – шлях до файлу сертифіката TLS;

– key_file – шлях до файлу приватного ключа TLS.

Ми визначимо ці змінні в сценарії.

Ansible також використовує механізм шаблонів Jinja2 для визначення змінних в сценаріях. Згадайте: ми вже зустрічали вираз {{conf_file}} в самому сценарії.

Ви можете використовувати всі можливості Jinja2 в своїх шаблонах, але ми не будемо докладно розглядати їх тут. За додатковою інформацією про шаблони Jinja2 звертайтеся до офіційної документації (http: //jinja.pocoo.org/docs/dev/templates/). Втім, вам навряд чи будуть потрібні всі просунуті можливості. Але ви майже напевно будете користуватися фільтрами.

Обробники

А тепер повернемося до нашого сценарію web-tls.yml. Ми не обговорили ще два елемента. Один з них – розділ оброблювачів handlers:

handlers:

- name: restart nginx

service: name = nginx state = restarted

І другий – ключ notify в деяких завданнях:

- name: copy TLS key

copy: src = files / nginx.key dest = {{key_file}} owner = root mode = 0600

notify: restart nginx

Обробники – це одна з умовних форм, які підтримуються в Ansible. Обробник схожий із завданням, але запускається тільки після отримання повідомлення від завдання. Завдання надсилає повідомлення, якщо виявляється зміна зі стану системи після її виконання.

Завдання повідомляє обробник з ім’ям, переданим йому у аргументі. У попередньому прикладі ім’я обробника restart nginx. Сервер Nginx потрібно перезапустити, якщо зміниться будь-який з компонентів:

– ключ TLS;
– сертифікат TLS;
– файл конфігурації;
– вміст каталогу sites-enabled.

Ми додаємо інструкцію notify в кожну задачу, щоб забезпечити перезапуск Nginx, якщо виконується одна з цих умов.

Запуск сценарію

Запуск сценарію виконується командою ansible-playbook.

$ ansible-ptaybook web-tts.yt

Виведення повинно виглядати приблизно так:

PLAY [Configure webseɾver with nginx and tls] **********

GATHERING FACTS **********
ok: [testseɾver]
TASK: [Install nglnx] **********
changed: [testserver]
TASK: [cɾeate diɾectories foɾ tls certificates] **********
changed: [testserver]
TASK: [copy TLS key] **********
changed: [testserver]
TASK: [copy TLS certificate] **********
changed: [testserver]
TASK: [copy nginx config file] **********
changed: [testserver]
TASK: [enable configuration] **********
ok: [testserver]
NOTIFIED: [restart nginx] **********
changed: [testserver]
PLAY RECAP **********
testserver: ok = B changed = б unreachable = 0 failed = 0

Відкрийте в браузері сторінку https: // localhost: 8443 (не забудьте “s” в кінці https). Якщо ви використовуєте Chrome, то, як і на рис. 2.12, отримаєте неприємне повідомлення про те, що “встановлене з’єднання не захищене”.

Рис. 2.12. Деякі браузери, такі як Chrome, не довіряють “самопідписним” сертифікатам TLS

Не турбуйтесь. Помилка очікувана, оскільки ми створили “самопідпісний” сертифікат TLS. А такі браузери, як Chrome, довіряють тільки сертифікатам, випущеним довіреним центром сертифікації.

Контрольні запитання

1. Поясніть поняття інфраструктури як коду.
2. Перелічіть інструменти управління конфігурацією.
3. Чим відрізняються інструменти управління конфігурацією один від оного?
4. Як працює Ansible?
5. Як встановити Ansible?
6. Що таке сценарій Ansible?
7. Поясніть сутність процесу керування конфігурацією.
8. З чого складається конфігурація програмного забезпечення?
9. Яку архітектуру має інструмент Ansible?
10. За яким типом застосування IaC працює Ansible?
11. Перерахуйте основні поняття в Ansible.
12. Функціонал Chef.
13. Функціонал Puppet.
14. Функціонал SaltStack.
15. Що таке playbook в Ansible?
16. Для чого використовується файл ansible.cfg?
17. Чим пов’язані сценарії та YAML в Ansible?
18. Яка структура сценарію в Ansible?
19. Чи підтримується TLS в Ansible?

3. АВТОМАТИЗОВАНЕ РОЗГОРТАННЯ МЕРЕЖЕВОЇ ІНФРАСТРУКТУРИ

3.1. Переваги інфраструктури як коду

Тепер, коли ви познайомилися зі всілякими різновидами IaC, можна задатися питанням: навіщо нам це потрібно? Навіщо вивчати цілу купу нових мов і інструментів, обтяжуючи себе ще більшою кількістю коду, який потрібно підтримувати?

Справа в тому, що код досить потужний. Зусилля, які йдуть на перетворення ручних процесів у код, винагороджуються величезним поліпшенням ваших можливостей по доставці ПО. Згідно з доповіддю про стан DevOps за 2019 рік (bit.ly/31kCUYX), організації, які застосовують такі методики, як IaC, розгортають код в 200 разів частіше і відновлюються після збоїв в 24 рази швидше, а на реалізацію нових функцій йде в 2555 разів менше часу.

Коли ваша інфраструктура визначена у вигляді коду, можна істотно поліпшити процес доставки ПЗ, використовуючи широкий діапазон методик зі світу програмування. Це дає переваги.

Самообслуговування. У більшості команд, які розгортають код вручну, мало сисадмінів (часто один), і тільки вони знають всі магічні заклинання для виконання розгортання і мають доступ до промислового середовищі. Це стає суттєвою перешкодою на шляху зростання компанії. Якщо ж ваша інфраструктура визначена у вигляді коду, весь процес розгортання можна автоматизувати, завдяки чому розробники зможуть доставляти свій код тоді, коли їм це потрібно.

Швидкість і безпека. Автоматизація значно прискорює процес розгортання, тому що комп'ютер може виконати всі його етапи куди швидше людини. При цьому підвищується безпека, так як автоматичний процес буде більш послідовним, відтвореним, стійким до помилок з людським фактором.

Документація. Замість того щоб тримати стан інфраструктури в голові одного системного адміністратора, ви можете описати його в вихідному файлі, який кожен зможе прочитати. Іншими словами, IaC грає роль документації, дозволяючи будь-якому працівнику компанії зрозуміти, як все працює, навіть якщо системний адміністратор йде у відпустку.

Управління версіями. Вихідні файли IaC можна зберігати в системі управління версіями, завдяки чому в журналі фіксацій коду буде записана вся історія вашої інфраструктури. Це дуже допомагає при налагодженні, так

як в разі виникнення проблеми завжди можна, насамперед, відкрити журнал і подивитися, що саме змінилося у вашій інфраструктурі. Слідом за цим проблему можна вирішити за рахунок простого відкату до попередньої версії коду IaC, в якій ви впевнені.

Перевірка. Якщо стан вашої інфраструктури описано в файлі, при кожному його зміненні можна влаштовувати розбір коду, запускати набір автоматичних тестів і проганяти його через засоби статичного аналізу. Досвід показує, що все це значно зменшує ймовірність дефектів.

Повторне використання. Ви можете упакувати свою інфраструктуру в універсальні модулі, і замість того, щоб виробляти розгортання кожного продукту в кожному середовищі з нуля, у вас буде можливість використовувати в якості основи відомі, задокументовані і перевірені на практиці компоненти.

Радість. Є ще одна дуже важлива причина, чому ви повинні використовувати IaC, яку часто не беруть до уваги: радість. Розгортання коду і управління інфраструктурою вручну – рутинний і виснажливий процес. Розробники і сисадміни терпіти не можуть такого роду роботу, оскільки в ній немає ніякої творчості, виклику або визнання. Ви можете ідеально розгортати код протягом місяців, і ніхто навіть не помітить, поки в один прекрасний день ви не напартачили. Це створює напружені неприємні обставини. IaC пропонує кращу альтернативу, яка дозволяє комп'ютерам і людям робити те, що вони вміють найкраще: автоматизувати і, відповідно, писати код.

Тепер ви розумієте, чому IaC важлива. Наступне питання: чи є Terraform кращим засобом IaC саме для вас? Щоб на це відповісти, ми коротко розглянемо принцип роботи Terraform, а потім порівняємо його з іншими популярними продуктами в цій галузі.

Як працює Terraform

Ось узагальнена і трохи спрощена картина того, як працює Terraform. Terraform – це інструмент з відкритим вихідним кодом від компанії HashiCorp, написаний мовою програмування Go. Код на Go компілюється в єдиний двійковий файл (якщо бути точним, по одному файлу для кожної підтримуваної операційної системи) з передбачуваною назвою terraform.

Цей файл дозволяє розгорнути інфраструктуру прямо з вашого ноутбука або складального сервера (або будь-якого іншого комп’ютера), і для всього цього не потрібно ніякої додаткової інфраструктури. Все завдяки тому, що всередині виконуваний файл terraform робить від вашого імені API-виклики до одного / кількох провайдерів, таких як AWS, Azure, Google Cloud, DigitalOcean, OpenStack і т. д. Це означає, що Terraform використовує

інфраструктуру, яку провайдери надають для своїх API-серверів, а також їх механізми аутентифікації, які ви вже застосовуєте (наприклад, ваші API-ключі для AWS).

Але звідки Terraform знає, які API-виклики потрібно робити? Для цього вам необхідно створити текстові файли з конфігурацією, в яких описується, яку інфраструктуру ви хочете створити. У концепції «інфраструктура як код» ці файли грають роль коду. Ось приклад конфігурації Terraform:

```
resource "aws_instance" "example" {
ami    = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
}
resource "google_dns_record_set" "a" {
name = "demo.google-example.com"
managed_zone = "example-zone"
ttl      = 300
rrdatas        = [aws_instance.example.public_ip]
 }
```

Навіть якщо ви ніколи раніше не бачили код Terraform, не повинно бути особливих проблем з тим, щоб його зрозуміти. Цей фрагмент змушує Terraform виконати API-виклики до двох провайдерів: до AWS, щоб розгорнути там сервер, і до Google Cloud, щоб створити DNS-запис, який вказує на IP-адресу сервера з AWS. Terraform дозволяє використовувати єдиний простий синтаксис для розгортання взаємопов’язаних ресурсів в декількох різних хмарах.

Ви можете описати всю свою інфраструктуру (сервери, бази даних, балансувальник навантаження, топологію мережі і т. д.) в конфігураційних файлах Terraform і зберегти їх в системі управління версіями. Потім цю інфраструктуру можна буде розгорнути за допомогою певних команд, таких як terraformapply. Утиліта terraform проаналізує ваш код, перетворює його в послідовність API-викликів до хмарних провайдерів, які в ньому задані, і виконає ці API-виклики від вашого імені максимально ефективним чином (рис. 3.1).

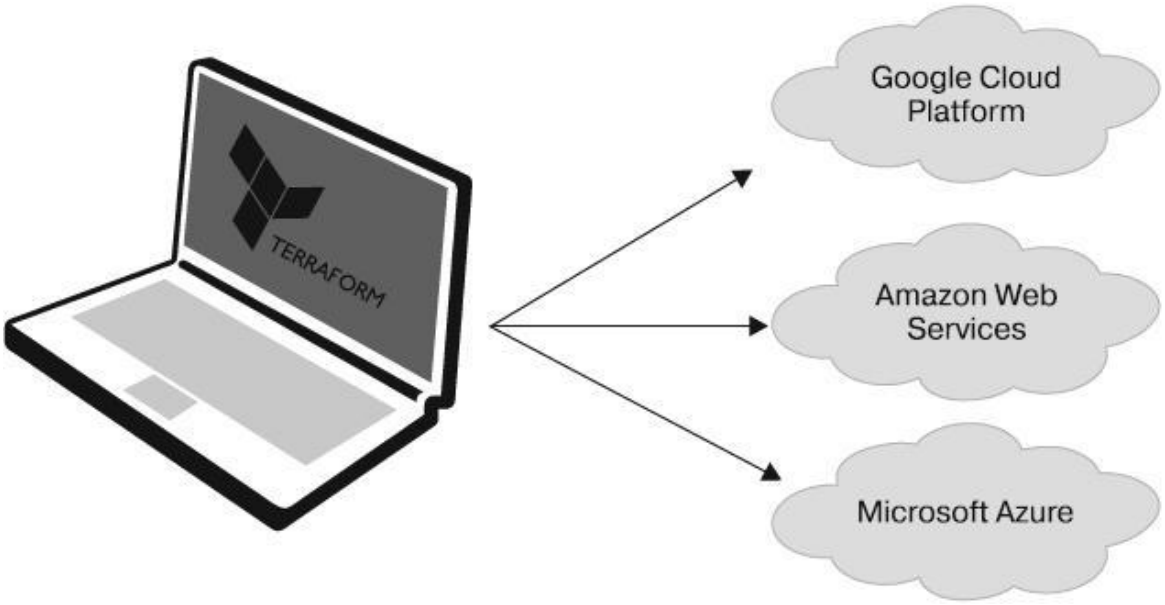


Рис. 3.1. Terraform – це утиліта, яка перетворює вміст ваших конфігураційних файлів в API-виклики до хмарних провайдерів

Якщо хтось у вашій команді хоче змінити інфраструктуру, то замість того, щоб робити це вручну прямо на серверах, вони редагують конфігураційні файли Terraform, перевіряють їх за допомогою автоматичних тестів і розбору коду, фіксують оновлений код у системі управління версіями і потім виконують команду terraformapply, щоб зробити необхідні для розгортання змін API-виклики.

Прозора переносимість між хмарними провайдерами

Оскільки Terraform підтримує безліч різних хмарних провайдерів, часто виникає питання: чи забезпечує цей інструмент прозору переносимість між ними? Наприклад, якщо ви використовували Terraform для опису багатьох серверів, баз даних, балансувальників навантаження та іншої інфраструктури в AWS, чи можете ви кілька разів клацнувши кнопкою миші розгорнути все це в іншій хмарі, такій як Azure або Google Cloud?

На практиці це питання виявляється не зовсім коректним. Ви не можете розгорнути “ідентичну інфраструктуру” в різних хмарах, оскільки інфраструктура, яка надається хмарними провайдерами, різниться! Сервери, балансувальники навантаження і бази даних, пропоновані AWS, Azure і Google Cloud, сильно розрізняються з точки зору можливостей, конфігурації, управління, безпеки, масштабованості, доступності, спостережливості і т. д. Не існує простого і прозорого способу подолати ці

відмінності, особливо з огляду на те, що деякі функції одного хмарного провайдера часто відсутні у всіх інших.

Підхід, який використовується в Terraform, дозволяє писати код для кожного провайдера окремо, користуючись його унікальними можливостями; при цьому всередині для всіх провайдерів застосовується таж мова, інструментарій та методики IaC.

Ініціалізація ресурсів плюс управління конфігурацією. Приклад: Terraform і Ansible. Terraform використовується для розгортання всієї внутрішньої інфраструктури, включаючи топологію мережі (тобто віртуальні приватні хмари (virtual private cloud, або VPC), підмережі, таблиці маршрутизації), сховища даних (MySQL, Redis), балансувальник навантаження і сервери. Ansible бере на себе розгортання ваших додатків поверх цих серверів.

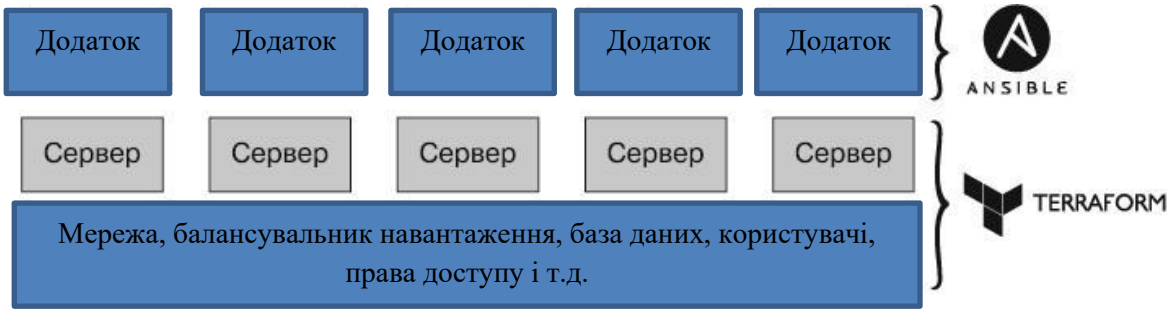


Рис. 3.2. Спільне використання Terraform і Ansible

Цей підхід дозволяє швидко приступити до роботи, оскільки вам не потрібна ніяка додаткова інфраструктура (Terraform і Ansible – суто клієнтські програми) і обидва інструменти можна інтегрувати безліччю різних способів (наприклад, Terraform призначає серверам спеціальні теги, які Ansible використовує для пошуку і зміни цих серверів). Основний недолік полягає в тому, що застосування Ansible зазвичай має на увазі використання великої кількості процедурного коду і змінюваних серверів, тому розширення кодової бази, інфраструктури і вашої команди може ускладнити обслуговування.

Ініціалізація ресурсів плюс шаблонізація серверів

Приклад: Terraform і Packer. Packer використовується для упаковки ваших додатків у вигляді образів ВМ. Потім Terraform розгортає: а) сервери за допомогою цих образів; б) всю решту інфраструктури, включаючи топологію мережі (тобто VPC, підмережі, таблиці маршрутизації), сховища даних (як MySQL, Redis) і балансувальники навантаження. Це проілюстровано на рис. 3.3.

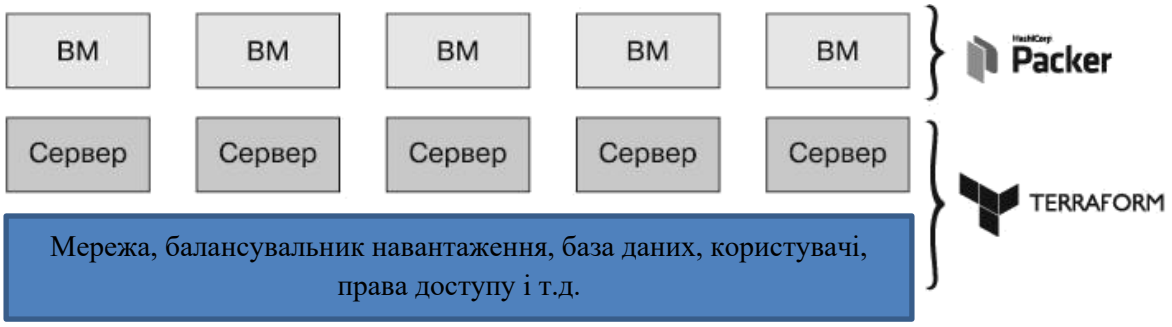


Рис. 3.3. Спільне застосування Terraform і Packer

Цей підхід теж дозволяє швидко приступити до роботи, так як вам не потрібна ніяка додаткова інфраструктура (Terraform і Packer є суто клієнтськими додатками). Пізніше в цій книзі ви зможете вдосталь попрактикуватися в розгортанні образів ВМ за допомогою Terraform. Крім того, ви отримуєте незмінну інфраструктуру, що спростить її обслуговування. Однак у цій комбінації є два суттєвих недоліки. По-перше, на збірку і розгортання образів ВМ може йти багато часу, що сповільнить випуск оновлень. По-друге, як ви побачите в наступних розділах, Terraform підтримує обмежений набір стратегій розгортання (наприклад, сам по собі цей інструмент не дозволяє реалізувати “синьо-зелені” оновлення), тому вам доведеться або написати багато складних скриптів, або звернутися до засобів оркестрації, як це буде показано далі.

Ініціалізація ресурсів, шаблонізація серверів, оркестрація

Приклад: Terraform, Packer, Docker і Kubernetes. Packer використовується для створення образів ВМ з встановленими Docker і Kubernetes. Потім Terraform розгортає: а) сервери за допомогою цих образів; б) всю решту інфраструктури, включаючи топологію мережі (VPC, підмережі, таблиці маршрутизації), сховища даних (MySQL, Redis) і балансувальник навантаження. Коли сервери завантажаться, вони сформують кластер Kubernetes, яким ви будете запускати і адмініструвати свої додатки у вигляді контейнерів Docker (рис. 3.4).

Перевага цього підходу в тому, що образи Docker збираються досить швидко, тому їх можна запускати і тестувати на локальному комп’ютері.

Kubernetes, включаючи різні стратегії розгортання, автозбереження, автомасштабирование і т. д. Недоліки пов’язані з підвищенням складності – як з точки зору інфраструктури (розгортання кластерів Kubernetes є складним і дорогим, хоча більшість основних хмарних провайдерів тепер надають керовані сервіси Kubernetes, на які можна покласти частину цієї роботи), так і в сенсі додаткових шарів абстракції (Kubernetes, Docker, Packer), які необхідно вивчати, обслуговувати і налагоджувати.

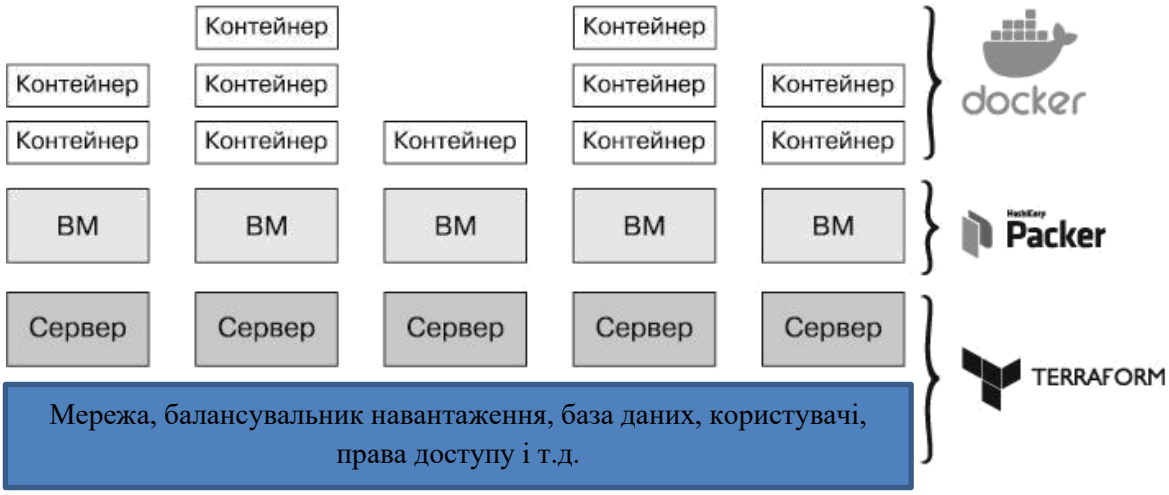


Рис. 3.4. Спільне використання Terraform, Packer, Docker і Kubernetes

Kubernetes, включаючи різні стратегії розгортання, автозбереження, автомасштабирование і т. д. Недоліки пов'язані з підвищенням складності – як з точки зору інфраструктури (розгортання кластерів Kubernetes є складним і дорогим, хоча більшість основних хмарних провайдерів тепер надають керовані сервіси Kubernetes, на які можна покласти частину цієї роботи), так і в сенсі додаткових шарів абстракції (Kubernetes, Docker, Packer), які необхідно вивчати, обслуговувати і налагоджувати.

3.2. Приступаємо до роботи з Terraform

У цьому розділі ви навчитеся основам застосування Terraform. Цей інструмент простий у вивченні, тому ви швидко пройдете шлях від виконання ваших перших команд до розгортання кластера серверів з балансувальником навантаження, який розподіляє між ними трафік. Така інфраструктура буде гарною відправною точкою для запуску масштабованих високодоступних вебсервісів.

Terraform вміє форматувати інфраструктуру як в публічних хмарах, на зразок Amazon Web Services (AWS), Azure, Google Cloud і DigitalOcean, так і в приватних хмарних платформах і системах віртуалізації на кшталт OpenStack і VMWare. Практично у всіх прикладах коду в книзі буде використовуватися AWS. Це хороший вибір для вивчення Terraform з наступних причин.

AWS, поза всякими сумнівами, є найпопулярнішим провайдером хмарної інфраструктури. Його частка на ринку хмарних рішень становить 45%, що більше, ніж у трьох найближчих конкурентів (Microsoft, Google і IBM), разом узятих (http://bit.ly/2kWCuCm).

AWS надає найширший спектр надійних і масштабованих сервісів з хмарним розміщенням, включаючи Amazon Elastic Compute Cloud (Amazon EC2), який можна використовувати для розгортання віртуальних серверів, Auto Scaling Groups (ASGs), що спрощує керування кластером віртуальних серверів, і Elastic Load Balancers (ELBs), за допомогою якого можна розподіляти трафік між віртуальними серверами кластера.

У перший рік використання AWS пропонує щедрий безкоштовний тариф (https://aws.amazon.com/free/), який дозволяє виконати всі ці приклади без грошових витрат. Якщо ви вже вичерпали свої безкоштовні кредити, робота з прикладами з цієї книги буде коштувати не дорожче декількох доларів.

Непотрібно хвилюватися, якщо ви раніше не використовували AWS або Terraform. Цей підручник підійде для новачків в обох технологіях. Він проведе вас через такі етапи, як:

підготовка вашого профілю в AWS;
установка Terraform;
розгортання одного сервера;
розгортання одного вебсервера;
розгортання конфигурируемого вебсервера;
розгортання кластера вебсерверів;
розгортання балансувальника навантаження;
видалення непотрібних ресурсів.

Підготовка вашого профілю в AWS

Якщо у вас немає облікового запису в AWS, зайдіть на сторінку aws.amazon.com і зареєструйтеся. Відразу після реєстрації ви входите в систему в якості кореневого користувача. Цей обліковий запис дозволяє робити що завгодно, тому з точки зору безпеки її краще не використовувати регулярно. Вона вам буде потрібна тільки для створення інших, призначених для користувача, облікових записів з обмеженими правами, на одну з яких ви повинні негайно перейти.

Щоб створити більш обмежений обліковий запис, слід використовувати сервіс Identity and Access Management (IAM). IAM – це те місце, де відбувається управління обліковими записами користувачів і їх правами. Щоб створити нового користувача IAM, перейдіть в консоль IAM (https://amzn.to/33fM2jf), клацніть на посиланні Users (Користувачі) і потім натисніть кнопку Create New Users (створити нових користувачів). Введіть ім’я користувача і переконайтеся, що прапорець Generate an access key for each user (згенерувати ключ доступу для кожного користувача) встановлено, як показано на рис. 3.5. Майте на увазі, що AWS вносить косметичні зміни

в свою вебконсоль, тому на момент читання цієї книги сторінки IAM можуть трохи відрізнятися.

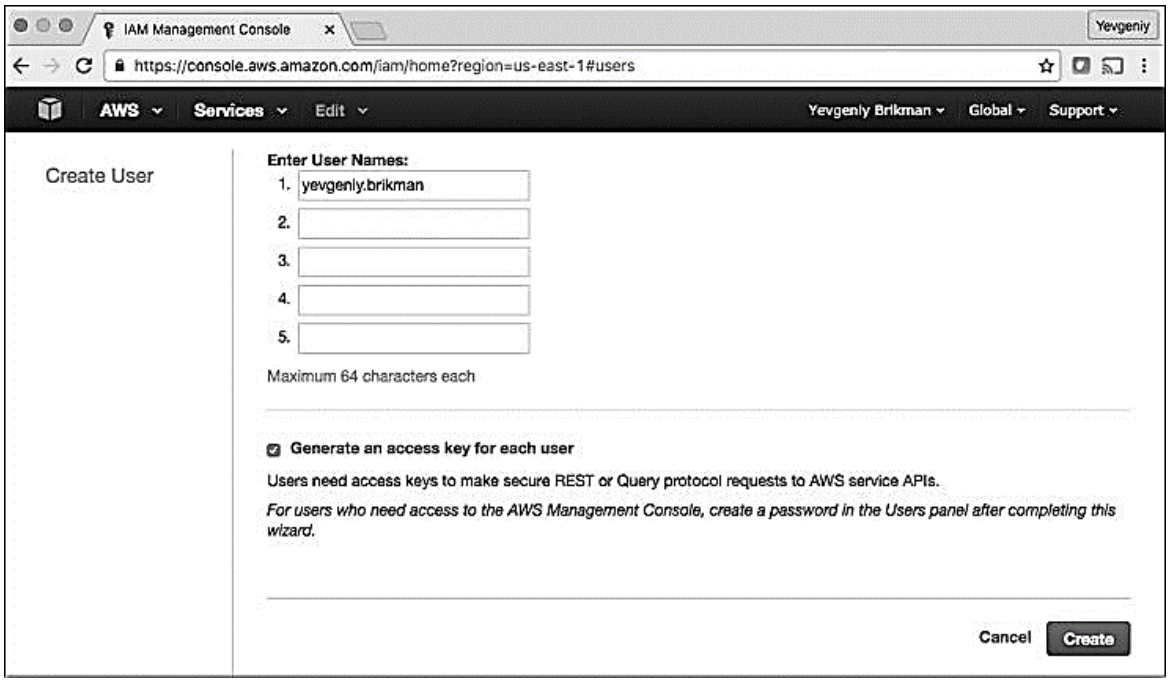


Рис. 3.5. Створення нового користувача IAM

Натисніть кнопку Create (Створити). AWS покаже вам облікові дані доступу цього користувача, які, як видно на рис. 3.6, складаються з ID ключа доступу (Access Key ID) і секретного ключа доступу (Secret Access Key). Їх слід негайно зберегти, оскільки їх більше ніколи не покажуть, а вони в цьому керівництві ще знадобляться. Пам'ятайте, що ці дані дають доступ до вашого облікового запису в AWS, тому зберігайте їх у безпечному місці (наприклад, в диспетчері паролів, такому як 1Password, LastPass або OS X Keychain) і ніколи нікому не давайте.

Зберігши свої облікові дані, натисніть кнопку Close (Закрити). Таким чином ви перейдете до списку користувачів IAM. Клацніть на імені користувача, якого тільки що створили, і виберіть вкладку Permissions (Права доступу). Нові користувачі IAM за замовчуванням позбавлені всяких прав і, отже, не можуть нічого робити в рамках облікового запису AWS.

Щоб видати користувачеві IAM якісь права, ви повинні зв'язати його обліковий запис з однією або декількома політиками IAM. Політика IAM – це документ у форматі JSON, який визначає, що користувачеві дозволено, а що – ні. Ви можете створювати свої власні політики або обійтися вже готовими, які називаються керованими політиками.

Для виконання прикладів з цієї книги вам потрібно призначити своєму користувачеві IAM наступні керовані політики (рис. 3.7):

AmazonEC2FullAccess;

AmazonS3FullAccess;
AmazonDynamoDBFullAccess;
AmazonRDSFullAccess;
CloudWatchFullAccess;
IAMFullAccess.

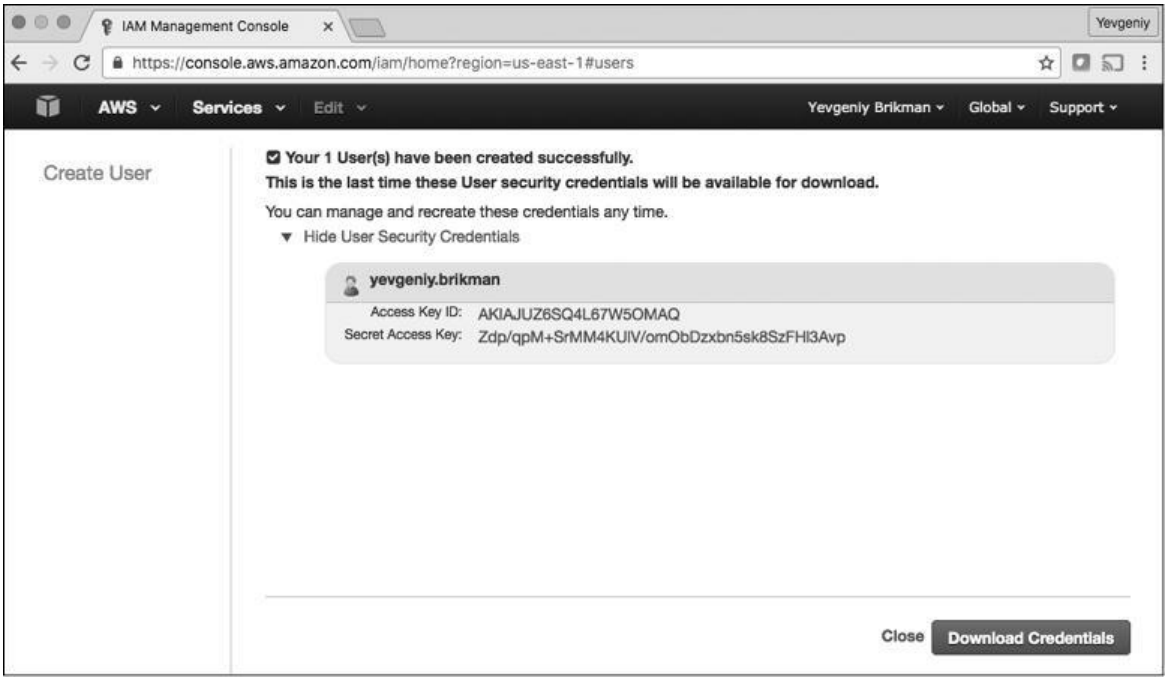


Рис. 3.6. Зберігайте свої облікові дані AWS в надійному місці. Нікому їх не показуйте (не хвилюйтеся, ті, що на знімку екрана – несправжні)

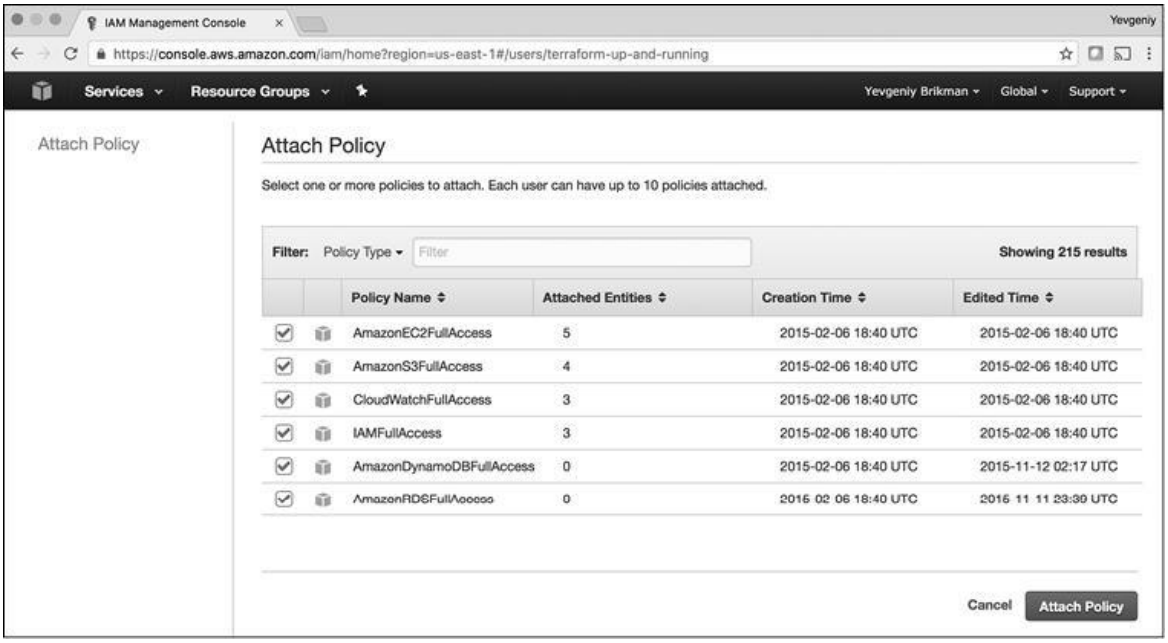


Рис. 3.7. Призначення декількох керованих політик IAM вашому новому користувачеві IAM

Зауваження про віртуальні приватні хмари за замовчуванням

Якщо ви використовуєте існуючий обліковий запис AWS, у нього вже має бути хмара VPC за замовчуванням. VPC (Virtual Private Cloud – віртуальна приватне хмара) – це ізольована область вашого профілю AWS з власними віртуальної мережею і простором IP-адрес. Практично будь-який ресурс AWS розгортається в VPC. Якщо не вказати VPC явно, ресурс буде розгорнуто в VPC за замовчуванням, який є частиною всіх нових облікових записів AWS. VPC за замовчуванням застосовується у всіх прикладах в цій книзі, тому, якщо з якоїсь причини ви його видалили, перейдіть на інший регіон (у кожного регіону своя хмара VPC за замовчуванням) або створіть нову в вебконсолі AWS (https://amzn.to/31lVUWW). В іншому випадку вам доведеться оновити майже всі приклади, додавши в них параметр vpc_id або subnet_id, який вказує на призначене для користувача VPC.

3.2.1. Встановлення Terraform

Ви можете завантажити Terraform на домашній сторінці проєкту за адресою https://www.terraform.io. Клацніть на посиланні для завантаження, виберіть відповідний пакет для своєї операційної системи, збережіть ZIP-архів і розпакуйте його в папку, в яку хочете встановити Terraform. Архів містить єдиний двійковий файл під назвою terraform, який слід додати в змінну середовища PATH. Як варіант, спробуйте пошукати Terraform в диспетчері пакетів своєї ОС; наприклад, в OS X можна виконати brewinstallterraform.

Щоб переконатися, що все працює, запустіть команду terraform. Ви повинні побачити інструкції по застосуванню:

$ terraform
Usage: terraform [-version] [-help] <command> [args]
Common commands:
apply Builds or changes infrastructure
console Interactive console for Terraform interpolations
destroy Destroy Terraform-managed infrastructure
env Workspace management
fmt Rewrites config files to canonical format
(...)

Щоб система Terraform могла вносити зміни до вашого профілю AWS, потрібно прописати в змінних середовища AWS_ACCESS_KEY_ID.

AWS_SECRET_ACCESS_KEY облікові дані для користувача IAM, якого ви створили раніше. Наприклад, ось як це можна зробити в терміналі Unix / Linux / macOS:

$ Export AWS_ACCESS_KEY_ID = (your access key id)

$ Export AWS_SECRET_ACCESS_KEY = (your secret access key)

Майте на увазі, що ці змінні середовища діють тільки в поточній командній оболонці, тому після перезавантаження комп'ютера або відкриття нового вікна терміналу доведеться знову їх експортувати.

Способи аутентифікації

Крім змінних середовища, Terraform підтримує ті ж механізми аутентифікації, що і всі утиліти командного рядка і SDK для AWS. Таким чином, ви зможете використовувати файл $ HOME / .aws / credentials, який автоматично генерується, якщо запустити AWS CLI з командою configure, або ролі IAM, які можна призначити майже будь-якому ресурсу в AWS. Детальніше про це у статті A Comprehensive Guide to Authenticating to AWS on the Command Line за адресою http://bit.ly/2M11muR.

3.2.2. Розгортання одного сервера

Код Terraform пишеться мовою конфігурації HashiCorp (HashiCorp Configuration Language, або HCL) і зберігається в файлах з розширенням .tf28. Це декларативна мова, тому ваше завдання описати потрібну вам інфраструктуру, а Terraform розбереться з тим, як її створити. Terraform вміє створювати інфраструктуру на різноманітних платформах (або провайдерів в термінології проекту), включаючи AWS, Azure, Google Cloud, DigitalOcean і багато інших.

Код Terraform можна писати в, практично, будь-якому текстовому редакторі. Якщо пошукати, можна знайти підсвічування синтаксису Terraform для більшості редакторів (зверніть увагу, що вам, можливо, потрібно шукати по слову HCL, а не Terraform), включаючи vim, emacs, Sublime Text, Atom, Visual Studio Code і IntelliJ (у останнього навіть є підтримка рефакторинга, пошуку входжень і переходу до оголошення).

Насамперед, при використанні Terraform звичайно треба вибрати провайдера (одного або декілька), з яким ви хочете працювати. Створіть порожню папку і помістіть в неї файл з ім'ям main.tf і наступним змістом:

```
provider "aws" {
region = "us-east-2"
}
```

Це говорить Terraform про те, що в якості провайдера ви збираєтеся використовувати AWS і хочете розгортати свою інфраструктуру в регіоні us-east-2. Обчислювальні центри AWS розкидані по всьому світу і згруповані по регіонах. Регіони AWS – це окремі географічні області, такі як us-east-2 (Огайо), eu-west-1 (Ірландія) і ap-southeast-2 (Сідней). Усередині кожної області знаходиться кілька ізольованих обчислювальних центрів, відомих як зони доступності: наприклад, us-east-2a, us-east-2b і т. д.

Кожен тип провайдерів підтримує створення різного виду ресурсів, таких як сервери, бази даних і балансувальник навантаження. Узагальнений синтаксис створення ресурсу в Terraform виглядає так:

resource "<PROVIDER> _ <TYPE>" "<NAME>" {[CONFIG ...]
}

PROVIDER – ім’я провайдера (припустимо, aws), TYPE – тип ресурсу, створюваного в цьому провайдері (на кшталт instance), NAME – ідентифікатор, за допомогою якого ви хочете посилатися на ресурс в коді Terraform (скажімо, my_instance), а CONFIG містить один або кілька аргументів, передбачених спеціально для цього ресурсу.

Наприклад, щоб розгорнути в AWS один (віртуальний) сервер (Екземпляр EC2), вкажіть у файлі main.tf ресурс aws_instance:

```
resource "aws_instance" "example" {
ami = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
}
```

Ресурс aws_instance підтримує багато різних аргументів, але поки що вам потрібні тільки два з них, які є обов’язковими:

ami. Образ Amazon Machine Image (AMI), який буде запущений на сервері EC2. В AWS Marketplace (https://aws.amazon.com/marketplace/) можна знайти платні та безкоштовні образи. Ви також можете створити власний примірник AMI, застосовуючи такі інструменти, як Packer (обговорення образів і шаблонізації серверів шукайте в підрозділі «Засоби шаблонізації серверів» на с. 31). Код, представлений вище, призначає параметру ami ідентифікатор AMI Ubuntu 18.04 в us-east-2. Цей образ можна використовувати безкоштовно.

instance_type. Тип сервера EC2, який потрібно запустити. У кожного типу є свій обсяг ресурсів процесора, пам’яті, дискового простору мережі. Всі доступні варіанти перераховані на відповідній сторінці за адресою https://amzn.to/2H49EOH. У наведеному вище прикладі використовується тип t2.micro, який має один віртуальний процесор з 1 Гбайт пам’яті і входить в безкоштовний тариф AWS.

3.2.3. Використовуйте документацію

Terraform підтримує десятки провайдерів, у кожного з яких є десятки ресурсів, і для кожного ресурсу передбачені десятки аргументів. Запам'ятати все це неможливо. При написанні коду Terraform слід регулярно звірятися з документацією, щоб подивитися, які ресурси вам доступні і як їх використовувати. Наприклад, ось документація для ресурсу aws_instance за адресою http://bit.ly/33dmi7g. Незважаючи на свій багаторічний досвід використання Terraform, я все ще звертаюся до цієї документації по кілька разів на день!

Відкрийте термінал, перейдіть в папку, в якій ви створили файл main.tf, і виконайте команду terraforminit:

terraform init Initializing the backend ...

Initializing provider plugins ...

- Checking for available provider plugins ...

- Downloading plugin for provider "aws" (terraform-providers / aws) 2.10.0...

The following providers do not have any version constraints in

configuration, so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking changes, it is recommended to add version = "..." constraints to the corresponding provider blocks in configuration, with the constraint strings suggested below.

provider.aws: version = "~> 2.10"

Terraform has been successfully initialized!

Виконуваний файл terraform підтримує основні команди Terraform, але він не містить ніякого коду для провайдерів (на кшталт AWS, Azure або GCP). Тому, починаючи роботу з цим інструментом, ви повинні виконати terraforminit, щоб він міг просканувати ваш код, визначити, з якими провайдерами ви працюєте, і завантажити для них відповідні модулі. За замовчуванням код провайдерів завантажується в папку .terraform, яка є робочою папкою Terraform (варто додати її в .gitignore). У наступних розділах ви познайомитеся з іншими сценаріями використання команди init і папки .terraform. А поки що просто знайте, що init необхідно виконувати кожен раз, коли ви починаєте писати новий код Terraform, і це можна робити багаторазово (це ідемпотентна команда).

Тепер, завантаживши код провайдера, виконайте команду terraformplan: $ terraform plan

(...)

Terraform will perform the following actions:

aws_instance.example will be created

+ Resource "aws_instance" "example" {

+ Ami = "ami-0c55b159cbfafe1f0"

+ Arn = (known after apply)

+ Associate_public_ip_address = (known after apply)

+ Availability_zone = (known after apply)

+ Cpu_core_count = (known after apply)

+ Cpu_threads_per_core = (known after apply)

+ Get_password_data = false

+ Host_id = (known after apply)

+ Id = (known after apply)

+ Instance_state = (known after apply)

+ Instance_type = "t2.micro"

+ Ipv6_address_count = (known after apply)

+ Key_name = (known after apply)

(...)

}

Plan: 1 to add, 0 to change, 0 to destroy.

Команда plan дозволяє побачити, що зробить Terraform, без внесення будь-яких змін. Це хороша можливість ще раз перевірити свій код перед випуском його в зовнішній світ. За своїм результатом команда plan схожа на утиліту diff, яка є частиною Unix, Linux і git: знак плюс позначає все, що буде створено; знак мінус (–) – що буде видалено, а то, що позначено тильдою (~), буде змінено. У попередньому висновку ви можете бачити, що Terraform планує створити лише один сервер EC2 і нічого іншого – саме те, що нам потрібно.

Щоб ініціювати створення сервера, потрібно виконати команду terraformapply:

$ Terraform apply

(...)

Terraform will perform the following actions:

aws_instance.example will be created + resource "aws_instance" "example" {

+ Ami = "ami-0c55b159cbfafe1f0"

+ Arn = (known after apply)

+ Associate_public_ip_address = (known after apply)

+ Availability_zone = (known after apply)

+ Cpu_core_count = (known after apply)

+ Cpu_threads_per_core = (known after apply)
+ Get_password_data = false

+ Host_id = (known after apply)
+ Id = (known after apply)
+ Instance_state = (known after apply)
+ Instance_type = "t2.micro"
+ Ipv6_address_count = (known after apply)
+ Ipv6_addresses = (known after apply)
+ Key_name = (known after apply)
(...)
}
Plan: 1 to add, 0 to change, 0 to destroy.
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
Enter a value:

Ви можете помітити, що команда apply відображає таке ж виведення, як plan, і зппитує вас, чи дійсно ви хочете перейти до здійснення цього плану. plan – окрема команда, яка в основному підходить для швидкої перевірки та розбору коду, але в більшості випадків ви будете відразу виконувати команду apply, перевіряючи її висновок.

Введіть yes і натисніть клавішу Enter, щоб розгорнути сервер EC2: Do you want to perform these actions?

Terraform will perform the actions described above. Only 'yes' will be accepted to approve.

Enter a value: yes
aws_instance.example: Creating ...
aws_instance.example: Still creating ... [10s elapsed]
aws_instance.example: Still creating ... [20s elapsed]
aws_instance.example: Still creating ... [30s elapsed]
aws_instance.example: Creation complete after 38s [id = i-07e2a3e006d785906]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Вітаємо, ви розгорнули сервер EC2 в своєму обліковому записі AWS, використовуючи Terraform! Щоб в цьому переконатися, перейдіть в консоль EC2 (https://amzn.to/2GOFxdI); ви повинні побачити сторінку, схожу на ту, що зображена на рис. 3.8.

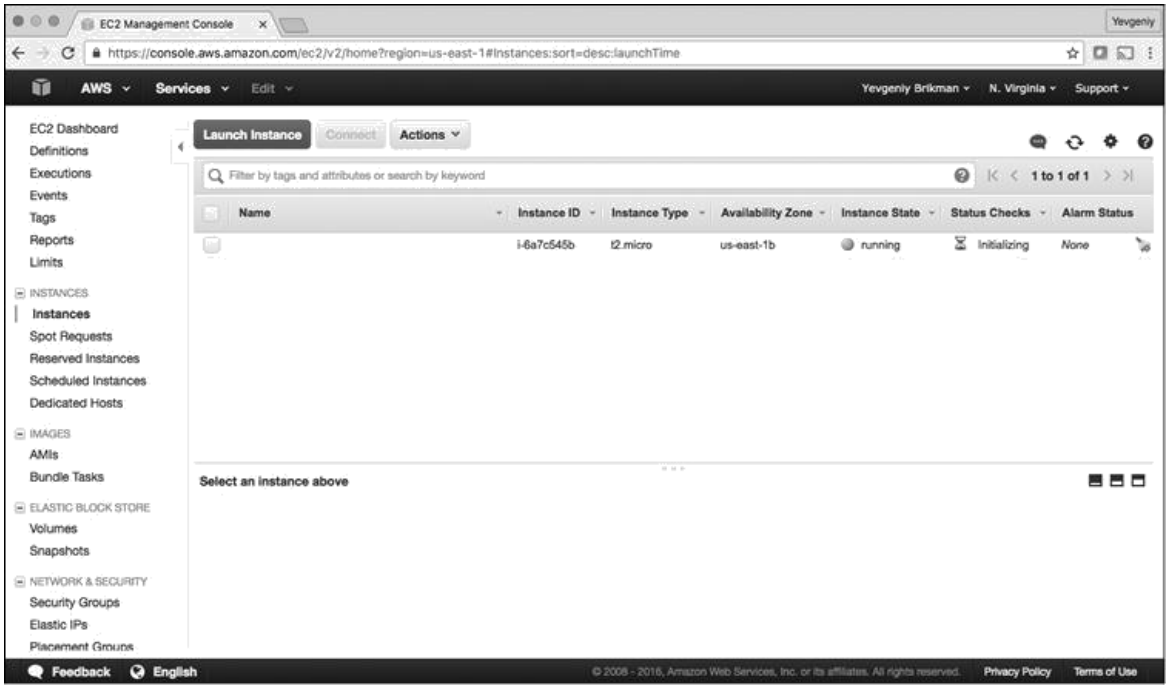


Рис. 3.8. Один сервер EC2

Ми дійсно бачимо наш сервер! Однак потрібно визнати, що це не самий вражаючий приклад. Зробимо його трохи більш цікавим. Для початку зверніть увагу на те, що у нашого сервера EC2 немає імені. Щоб його вказати, додайте до ресурсу aws_instance параметр tags:

resource "aws_instance" "example" {
ami = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
tags = {
Name = "terraform-example"
}
}

Щоб побачити наслідки цих змін, ще раз виконайте команду terraformapply:

$ Terraform apply
aws_instance.example: Refreshing state ...
(...)
Terraform will perform the following actions:
aws_instance.example will be updated in-place
~ Resource "aws_instance" "example" {

ami = "ami-0c55b159cbfafe1f0" availability_zone = "us-east-2b" instance_state = "running" (...)

tags = {

"Name" = "terraform-example"
}
(...)

}
Plan: 0 to add, 1 to change, 0 to destroy.
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
Enter a value:

Система Terraform відстежує всі ресурси, які були створені для цього набору конфігураційних файлів. Тому їй відомо, що сервер EC2 вже існує (зверніть увагу на рядок Refreshingstate ... при виконанні команди apply), і вона може вивести різницю між тим, що розгорнуто на даний момент, і тим, що описано в вашому коді. У попередньому порівнянні видно, що Terraform хоче створити один тег під назвою Name. Це саме те, що вам потрібно, тому введіть yes і натисніть клавішу Enter.

Оновивши свою консоль EC2, ви побачите приблизно таке вікно, як на рис. 3.9.

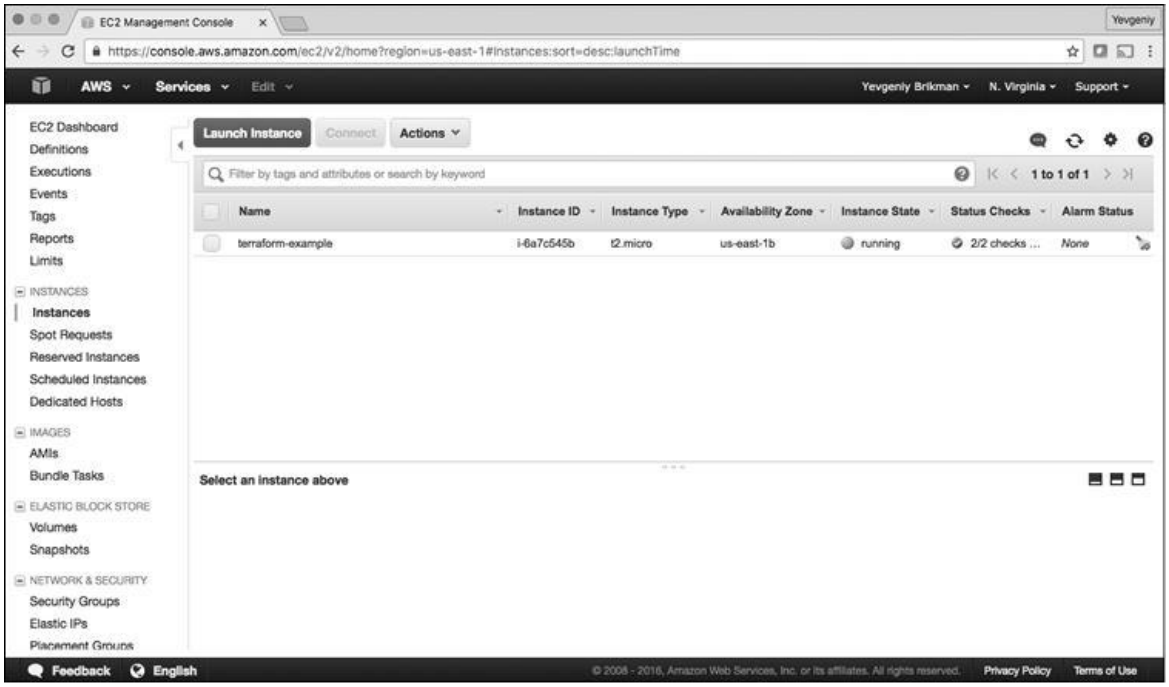


Рис. 3.9. У сервера EC2 тепер є тег з ім’ям

Отже, у вас тепер є робочий код Terraform, варто зберегти його в системі управління версіями. Це дозволить вам ділитися ним з іншими членами команди, відслідковувати історію всіх змін інфраструктури і використовувати журнал фіксацій для налагодження. Нижче показаний приклад того, як створити локальний Git-репозиторій і застосовувати його для зберігання конфігураційного файла Terraform:

git init
git add main.tf
git commit -m "Initial commit"

Потрібно також створити файл під назвою .gitignore. Він змушує Git ігнорувати певні типи файлів, які ви б не хотіли зберегти завдяки випадку:

.terraform

** .tfstate*

** .tfstate.backup*

Файл .gitignore, показаний вище, змушує Git ігнорувати папку .terraform, яку Terraform використовує в якості тимчасової робочої папки, а також файли виду * .tfstate, в яких Terraform зберігає стан (в розділі 3 ви дізнаєтеся, чому файли стану не можна записувати в репозиторій). Файл .gitignore теж потрібно зафіксувати:

git add .gitignore

git commit -m "Add a .gitignore file"

Щоб поділитися цим кодом зі своїми колегами, вам необхідно створити загальний Git-репозиторій, до якого ви всі зможете звертатися. Для цього можна використовувати GitHub. Перейдіть на сайт https://github.com, зареєструйтесь (якщо ви цього ще не зробили), створіть новий репозиторій і вкажіть його в якості віддаленої точки входу під назвою origin для свого локального Git-сховища:

git remote add origin git@github.com:

<YOUR_USERNAME> / <YOUR_REPO_NAME> .git

Тепер, коли ви хочете поділитися своїми змінами з колегами, можете завантажити зміни в репозиторій origin:

git push origin master

Якщо ви хочете подивитися, які зміни внесли ваші колеги, можете завантажити зміни зі сховищ origin:

git pull origin master

При подальшому читанні цього посібника і в цілому при застосуванні Terraform не забувайте регулярно фіксувати (gitcommit) і завантажувати (gitpush) свої зміни. Це дозволить працювати над кодом спільно з членами вашої команди. До того ж всі зміни вашої інфраструктури записуватимуться в журнал фіксацій, що стане дуже до речі в разі пошкодження.

3.2.4. Розгортання одного вебсервера

Наступним кроком буде розгортання на цьому інстансі вебсервера.

Ми спробуємо розгорнути найпростішу вебархітектуру: один веб-сервер, здатний відповідати на HTTP-запити (рис. 3.10).

У реальних умовах для створення вебсервера використовувався б вебфреймворк на зразок Ruby on Rails або Django. Щоб не ускладнювати цей приклад, ми запустимо найпростіший вебсервер, який завжди повертає текст Hello, World:

#! / Bin / bash

echo "Hello, World"> index.html

nohup busybox httpd -f -p 8080 &

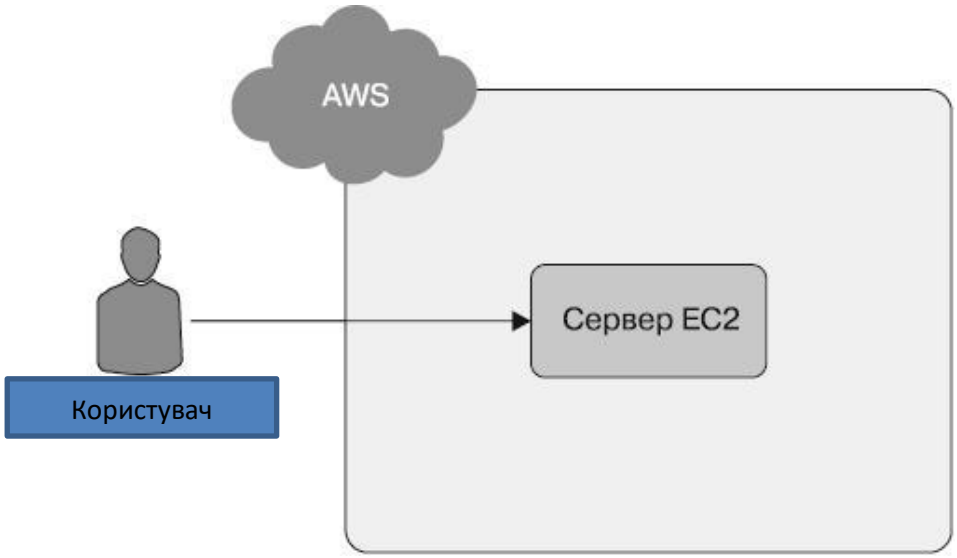


Рис. 3.10. Архітектура: один вебсервер, запущений в AWS, який відповідає на HTTP-запити

Це bash-скрипт, який записує текст Hello, World в файл index.html, який потім лунає на порті 8080 за допомогою вебсервера, запущеного з використанням інструменту під назвою busybox (https://busybox.net/) (в Ubuntu встановлений по замовчуванню). Я вказав команду busybox між nohup і &, щоб вебсервер постійно працював у фоновому режимі, навіть після завершення bash-скрипта.

Номери портів

В цьому прикладі замість стандартного HTTP-порту 80 використовується 8080. Справа в тому, що всі порти менше 1024 вимагають адміністраторських привілеїв. Це потенційний ризик безпеки, оскільки будь-який зловмисник, якому вдасться зламати ваш сервер, теж отримає привілеї адміністратора.

У зв'язку з цим вебсервер рекомендується запускати від імені звичайного користувача з обмеженими правами доступу. Це означає, що вам слід прослуховувати порти з більш високими номерами, але, як ви побачите пізніше в цьому розділі, ви можете налаштувати балансувальник навантаження так, щоб він працював на порті 80 і перенаправляв трафік на більш високі порти ваших серверів.

Як запустити цей скрипт на сервері EC2? Оскільки в цьому прикладі в якості вебсервера використовується однорядковий скрипт на основі busybox, ви можете обійтися стандартним чином Ubuntu 18.04 і запустити скрипт Hello, World в рамках конфігурації користувальницьких даних вашого сервера EC2. Ви можете передати параметру user_data в коді

Terraform те, що буде виконано при завантаженні сервера: або скрипт командної оболонки, або директиву cloud-init. Варіант зі скриптом показаний нижче:

```
resource "aws_instance" "example" {
        ami         = "ami-0c55b159cbfafe1f0"
        instance_type       = "t2.micro"
        user_data = <<-EOF
                #!/bin/bash
                echo "Hello, World" > index.html
                nohup busybox httpd -f -p 8080 &
                EOF
        tags = {
          Name = "terraform-example"
        }
      }
```

<< - EOF і EOF – елементи синтаксису heredoc, який дозволяє створювати багаторядкові літерали без використання безлічі символів переходу на новий рядок.

Перед запуском цього вебсервера потрібно зробити ще дещо. AWS за замовчуванням закриває для сервера EC2 весь вхідний і вихідний трафік.

Щоб ваш сервер міг приймати запити на порті 8080, необхідно створити групу безпеки:

```
resource "aws_security_group"
  "instance" { name = "terraform-
  example-instance" ingress {
    from_port   = 8080
    to_port       = 8080
    protocol      = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Цей код створює новий ресурс під назвою aws_security_group (зверніть увагу, що всі ресурси в AWS починаються з aws_) і робить так, щоб ця група дозволяла приймати на порті 8080 TCP-запити з блоку CIDR 0.0.0.0/0. Блок CIDR – це короткий запис діапазону IP-адрес. Наприклад, блок CIDR 10.0.0.0/24 представляє всі IP-адреси між 10.0.0.0 і 10.0.0.255. Блок CIDR 0.0.0.0/0 охоплює діапазон всіх можливих IP-адрес, тому дана група безпеки дозволяє приймати на порті 8080 запити з будь-якого IP30.

Створення групи безпеки, як такої, буде недостатньо. Потрібно зробити так, щоб сервер EC2 її використав. Для цього ви повинні передати її ідентифікатор аргументу vpc_security_group_ids ресурсу aws_instance. Але спочатку необхідно познайомитися з виразами Terraform.

Terraform виразом є все, що повертає значення. Ви вже бачили найпростіший тип виразів – *літерали*, такі ж як рядки (Наприклад, "ami-0c55b159cbfafe1f0") і числа (скажімо, 5). Terraform підтримує багато інших різновидів виразів, які будуть зустрічатися на сторінках цього підручника.

Особливо корисним типом виразів є посилання, яке дозволяє звертатися до значень з інших ділянок коду. Щоб вказати ID групи безпеки, потрібно послатися на атрибут ресурсу за допомогою такого синтаксису:

<PROVIDER> _ <TYPE>. <NAME>. <ATTRIBUTE>.

PROVIDER – це ім'я провайдера (наприклад, aws), TYPE – це тип ресурсу (на зразок security_group), NAME – ім'я цього ресурсу (в нашому випадку група безпеки називається "instance"), а ATTRIBUTE – це або один з аргументів ресурсу (скажімо, name), або один з атрибутів, які він експортував (список доступних атрибутів можна знайти в документації кожного ресурсу). Група безпеки експортує атрибут під назвою id, тому до нього можна звернутися за допомогою такого виразу:

aws_security_group.instance.id

Ви можете використовувати ідентифікатор цієї групи безпеки в аргументі vpc_security_group_ids ресурсу aws_instance:

resource "aws_instance" "example" {
ami = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
vpc_security_group_ids = [aws_security_group.instance.id] user_data = <<-EOF
#!/bin/bash
echo "Hello, World" > index.html
nohup busybox httpd -f -p 8080 &
EOF
tags = {
Name = "terraform-example"
}
}

Посилаючись в одному ресурсі на інший, ви створюєте неявну залежність. Terraform аналізує такі залежності, робить з них граф, застосовує його для автоматичного визначення порядку, в якому повинні створюватися ресурси. Наприклад, якби цей код розгортався з нуля, система Terraform знала б про те, що групу безпеки потрібно створити раніше, ніж сервер EC2, оскільки останній використовує ID цієї групи. Ви можете навіть вивести граф залежностей за допомогою команди graph:

$ terraform graph digraph {
compound = "true" newrank = "true" subgraph "root" {
"[root] aws_instance.example"
[label = "aws_instance.example", shape = "box"] "[root] aws_security_group.instance"

[label = "aws_security_group.instance", shape = "box"] "[root] provider.aws"

[label = "provider.aws", shape = "diamond"] "[root] aws_instance.example" ->

"[root] aws_security_group.instance" "[root] aws_security_group.instance" ->

"[root] provider.aws"

"[root] meta.count-boundary (EachMode fixup)" -> "[root] aws_instance.example"

"[root] provider.aws (close)" ->

"[root] aws_instance.example"

"[root] root" ->

"[root] meta.count-boundary (EachMode fixup)"

"[root] root" ->

"[root] provider.aws (close)"

}

}

Результат команди виконаний мовою опису графів під назвою DOT. Сам граф можна відобразити, як це зроблено на рис. 3.11, з використанням додатку Graphviz або його вебверсії GraphvizOnline (bit.ly/2mPbxmg).

При проходженні по дереву залежностей, Terraform намагається якомога сильніше розпаралелити створення ресурсів, що призводить до досить ефективного застосування змін. У цьому принадність декларативної мови: ви просто описуєте те, що вам потрібно, а Terraform визначає найбільш ефективний спосіб реалізації.

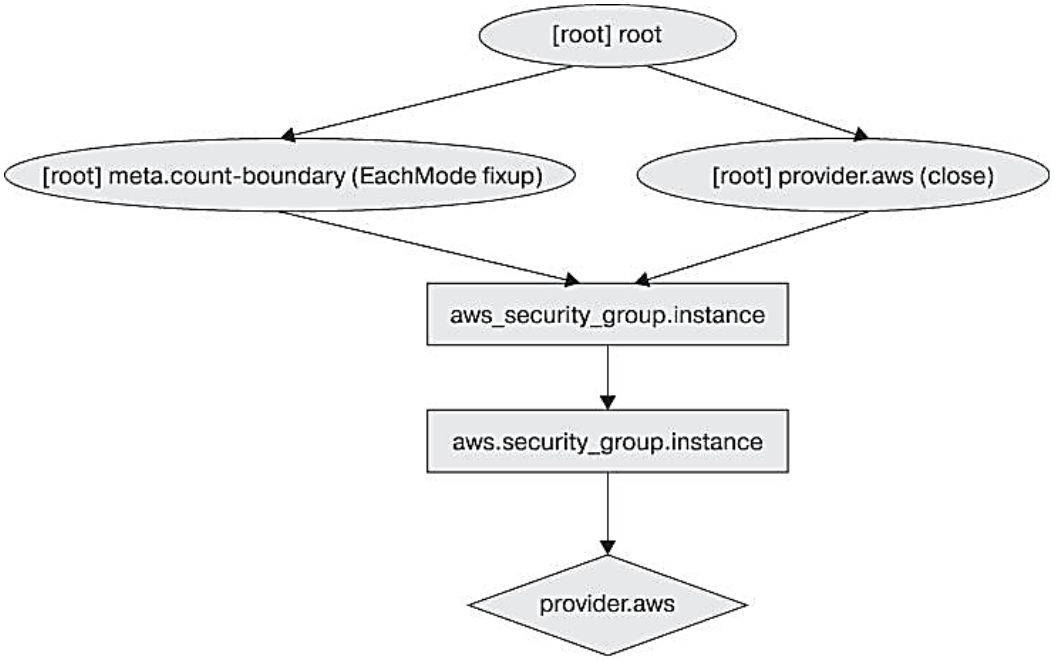


Рис. 3.11. Граф залежностей для сервера EC2 і його групи безпеки

Виконавши команду apply, ви побачите, що Terraform хоче створити групу безпеки і замінити наявний сервер EC2 іншим – з новими даними користувачів:

```
$ terraform apply
(...)
Terraform will perform the following actions:
 # aws_instance.example must be replaced
-/+ resource "aws_instance" "example" {
 ami = "ami-0c55b159cbfafe1f0"
 ~ availability_zone = "us-east-2c" -> (known after apply)
 ~ instance_state = "running" -> (known after apply)
 instance_type = "t2.micro"
 (...)
 + user_data = "c765373..." # forces replacement
 ~ volume_tags = {} -> (known after apply)
 ~ vpc_security_group_ids = [
 - "sg-871fa9ec",
 ] -> (known after apply)
 (...)
 }
# aws_security_group.instance will be created
+ resource "aws_security_group" "instance" {
 + arn = (known after apply)
 + description = "Managed by Terraform"
 + egress = (known after apply)
 + id = (known after apply)
 + ingress = [
 + {
 + cidr_blocks = [
 + "0.0.0.0/0",
 ]
 + description = ""
 + from_port = 8080
 + ipv6_cidr_blocks = []
 + prefix_list_ids = []
 + protocol = "tcp"
 + security_groups = []
 + self = false
 + to_port = 8080
 }, ]
 + name = "terraform-example-instance"
 + owner_id = (known after apply)
 + revoke_rules_on_delete = false
 + vpc_id = (known after apply)
 }
Plan: 2 to add, 0 to change, 1 to destroy.
```

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
Enter a value:

Символи – / + у виведенні плану означають “замінити”. Щоб зрозуміти, чим продиктована та чи інша зміна, пошукайте в виведенні плану словосполучення forces replacement. Зміна багатьох аргументів ресурсу aws_instance призводить до заміни. Це означає, що наявний сервер EC2 буде видалений, а його місце займе новий сервер. Варто відзначити, що, незважаючи на заміну вебсервера, жоден з його користувачів не помітить перебоїв в роботі. Також, за допомогою Terraform можна виконувати розгортання з нульовим часом простою.

Схоже, з планом все в порядку, тому введіть yes, і ви побачите, як розгортається новий сервер EC2 (рис. 3.12).

Якщо клацнути на новому сервері, внизу сторінки, на панелі з описом, можна побачити його публічну IP-адресу. Зачекайте, щоб він завантажився, і потім зробіть HTTP-запит за цією адресою на порті 8080, використовуючи браузер або утиліту на кшталт curl:

curl http://<EC2_INSTANCE_PUBLIC_IP>:8080
Hello, World

Ура! Тепер у вас є робочий вебсервер, запущений в AWS!

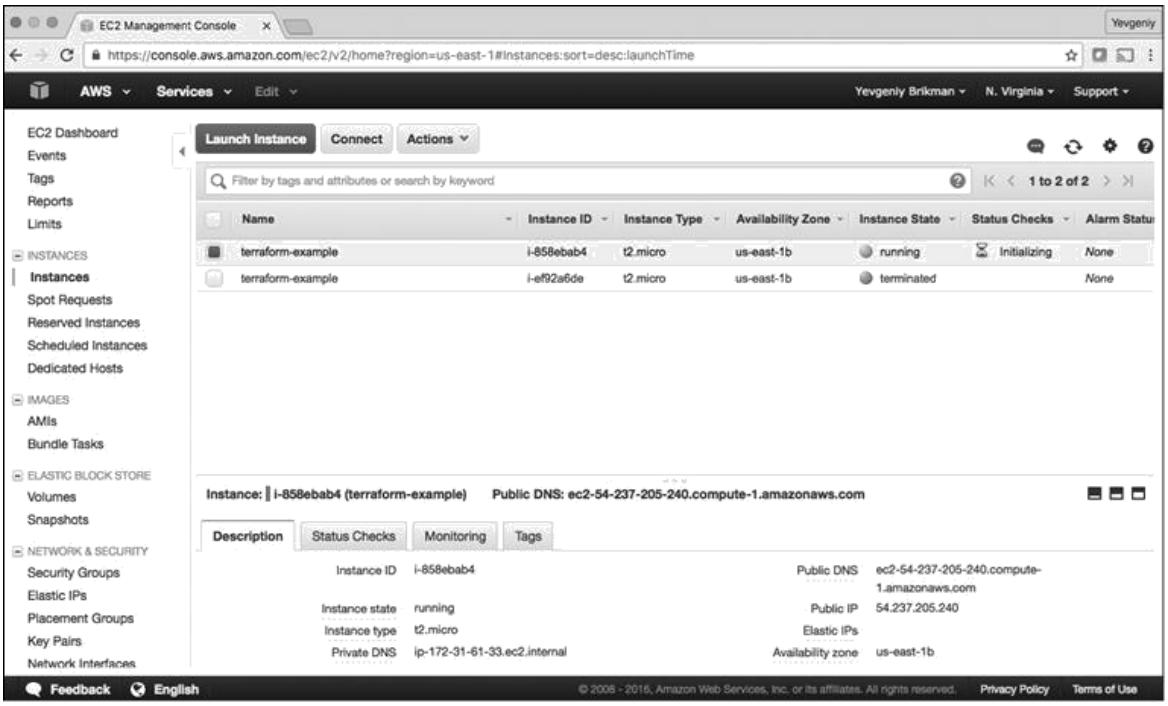


Рис. 3.12. Замість старого сервера EC2 ми отримуємо новий, з кодом вебсервера

3.2.5. Мережева безпека

Щоб не ускладнювати приклади в цій книзі, розгортання відбувається не тільки в VPC за замовчуванням (як згадувалося раніше), але і в стандартні підмережі цього VPC. VPC складається з однієї або декількох підмереж,

кожна з яких має власні IP-адреси. Всі підмережі в VPC за замовчуванням є публічними – їх IP-адреси доступні з Інтернету. Завдяки цьому ви можете перевірити роботу свого сервера EC2 на домашньому комп'ютері.

Розміщення сервера в публічній підмережі підходить для швидкого експерименту, але в реальних умовах це потенційний ризик безпеки. Хакери з усього світу постійно сканують IP-адреси випадковим чином у надії знайти якісь вразливості. Якщо ваші сервери доступні зовні, досить лише залишити незахищеним один порт або скористатися застарілим кодом з відомою вразливістю і хтось зможе проникнути всередину.

Таким чином, в промислових системах всі сервери і вже точно всі сховища даних слід розгортати в закритих підмережах, IP-адреси яких доступні тільки всередині VPC, але не з публічного Інтернету. Все, що має перебувати в публічних підмережах – це невелика кількість зворотних проксі і балансувальник навантаження, в яких закрито всі, що тільки можна (пізніше ви побачите приклад того, як розгорнути балансувальник навантаження).

3.2.6. Розгортання конфігуруємого вебсервера

Ви, напевно, помітили, що код вебсервера дублює порт 8080 в групі безпеки і конфігурації користувальницьких даних. Це суперечить принципу "не повторювати" (do not repeat yourself, або DRY): кожен елемент інформації в системі повинен мати єдине, однозначне достовірне представлення. Якщо номер порту зазначений в двох місцях, легко опинитися в ситуації, коли одне із значень оновлюється, а інше – ні.

Щоб ваш код можна було зробити більш конфігурованим і таким, що відповідає принципу DRY, Terraform дозволяє визначати вхідні змінні. Для цього передбачений наступний синтаксис:

variable "NAME" {[CONFIG ...]
}

Тіло оголошення змінної може містити три необов'язкові параметри.

description. Цей параметр завжди бажано вказувати для документування того, як використовується змінна. Ваші колеги зможуть переглянути цей опис не тільки при читанні коду, але і під час виконання команд plan або apply (приклад цього показаний трохи нижче).

default. Ви можете присвоїти значення змінній декількома способами, в тому числі через командний рядок (за допомогою параметра – var), файл (вказуючи параметр – var-file) або змінну середовища (Terraform шукає змінні середовища виду TF_VAR_ <ім'я_змінної>). Якщо змінна не ініціалізована, їй присвоюється значення за замовчуванням. Якщо такого немає, Terraform запросить його у користувача в інтерактивному режимі.

type. Дозволяє застосувати до змінних, які передає користувач, обмеження типів. Terraform підтримує ряд обмежень для таких типів, як string, number, bool, list, map, set, object, tuple і any.

Якщо тип не вказано, Terraform приймає значення як any.

Ось приклад вхідної змінної, яка перевіряє, чи є передане значення числом:

```
variable "number_example" {
description = "An example of a number variable in Terraform"
type = number
default = 42
}
```

Ось приклад змінної, яка перевіряє, чи є значення списком:

```
variable "list_example" {
description = "An example of a list in Terraform"
type = list
default = [ "a", "b", "c"]
}
```

Обмеження типів можна поєднувати. Наприклад, ось вхідна змінна, яка приймає список і вимагає, щоб всі значення цього списку були числовими:

```
variable "list_numeric_example" {
description = "An example of a numeric list in Terraform"
type = list (number)
default = [1, 2, 3]
}
```

Ось асоціативний масив, який вимагає, щоб всі значення були строковими:

```
variable "map_example" {
description = "An example of a map in Terraform"
type = map (string)
default = {
key1 = "value1"
key2 = "value2"
key3 = "value3"
}
}
```

Ви також можете створювати більш складні *структурні типи*, використовуючи обмеження object і tuple:

```
variable "object_example" {
description = "An example of a structural type in Terraform"
type = object ({
name = string
age = number
tags = list (string)
enabled = bool
})
default = {
```

```
name = "value1"
age = 42
tags = [ "a", "b", "c"]
enabled = true
}  }
```

Вище створюється вхідні змінна, яка вимагає, щоб значення було об’єктом з ключами name (рядок), age (число), tags (список рядків) і enabled (логічне значення). Якщо спробувати присвоїти такій змінній значення, яке не відповідає цьому типу, Terraform негайно поверне помилку типізації. У наступному прикладі демонструється спроба привласнити enabled рядок замість булевого значення:

```
variable "object_example_with_error" {
description = "An example of a structural type in Terraform with an error"
type = object ({
name = string
age = number
tags = list (string)
enabled = bool
})
default = {
name = "value1"
age = 42
tags = [ "a", "b", "c"]
enabled = "invalid"
}
}
```

Ви отримаєте наступну помилку:

```
$ Terraform apply
Error: Invalid default value for variable
on variables.tf line 78, in variable "object_example_with_error":
default = {
name = "value1"
80: age = 42
81: tags = [ "a", "b", "c"]
enabled = "invalid"
83:}
```

This default value is not compatible with the variable's type constraint: a bool is required.

Для прикладу з вебсервером досить змінної, яка зберігає номер порту:

```
variable "server_port" {
description = "The port the server will use for HTTP requests"
type = number
}
```

Зверніть увагу, що біля вхідної змінної server_port немає поля default, тому, якщо виконати команду apply прямо зараз, Terraform відразу ж попросить ввести значення для server_port і покаже вам опис description:

```
terraform apply
var.server_port
The port the server will use for HTTP
requests Enter a value:
```

Якщо ви не хочете мати справу з інтерактивним рядком введення, можете надати значення змінній за допомогою параметра командного рядка -var:

```
$ Terraform plan -var "server_port = 8080"
```

Те ж саме можна зробити, додавши змінну середовища виду TF_VAR_ <name>, де name – ім’я змінної, яку ви хочете встановити:

```
export TF_VAR_server_port = 8080
terraform plan
```

Якщо ж ви не хочете тримати в голові додаткові аргументи командного рядка при кожному виконанні команди plan або apply, можете вказати значення за замовчуванням:

```
variable "server_port" {
description = "The port the server will use for HTTP requests"
type = number
default = 8080
}
```

Щоб використовувати значення вхідної змінної в коді Terraform, слід скористатися виразом типу “посилання на змінну”, який має наступний синтаксис:

```
var. <VARIABLE_NAME>
```

Наприклад, так можна присвоїти параметрам групи безпеки from_port і to_port значення змінної server_port:

```
resource "aws_security_group" "instance"
 {name = "terraform-example-instance"
 ingress {
   from_port = var.server_port
   to_port = var.server_port
   protocol = "tcp"
   cidr_blocks = [ "0.0.0.0/0"]
  }
}
```

Доброю ідеєю буде використання однієї і тієї ж змінної при завданні порту в скрипті user_data. Вказати посилання всередині рядкового літерала можна за допомогою рядкової інтерполяції, яка має наступний синтаксис:

```
"$ {...}"
```

Усередині фігурних дужок можна розмістити будь-яке коректне посилання, і Terraform перетворює його в рядок. Наприклад, ось як можна скористатися значенням var.server_port всередині рядка user_data:

```
user_data = << - EOF
#! / Bin / bash
echo "Hello, World"> index.html
nohup busybox httpd -f -p $ {var.server_port} &
EOF
```

Крім вхідних змінних, Terraform дозволяє визначати і вихідні. Для цього передбачено такий синтаксис:

```
output "<NAME>" {
value = <VALUE>
[CONFIG ...]
}
```

NAME – це ім'я вихідної змінної, а в якості VALUE можна вказати будь-який вираз Terraform, який ви хочете вивести. CONFIG може мати два додаткових (і необов'язкових) параметри:

description. Цей параметр завжди бажано застосовувати для документування того, як використовується вихідна змінна;

sensitive. Якщо присвоїти даному параметру true, Terraform не стане зберігати це виведення в журнал, після виконання команди terraformapply. Це корисно, коли змінна містить делікатний матеріал або конфіденційні дані, такі як паролі або секретні ключі.

Наприклад, замість того, щоб вручну бродити по консолі EC2 в пошуках IP-адреси свого сервера, ви можете вивести його у вигляді вихідної змінної:

```
 output "public_ip" {
value = aws_instance.example.public_ip description =
"The public IP address of the web server"
}
```

Тут ми знову посилаємося на атрибути, на цей раз на атрибут public_ip ресурсу aws_instance. Якщо знову виконати команду apply, Terraform не внесе жодних змін (оскільки ви не міняли ніякі ресурси), але покаже вам в самому кінці новий висновок:

```
$ terraform apply
(...)
aws_security_group.instance: Refreshing state ... [id = sg-
078ccb4f9533d2c1a]
aws_instance.example: Refreshing state ... [id = i-028cad2d4e6bddec6]
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
public_ip = 54.174.13.5
```

Як бачите, після виконання terraformapply вихідна змінна виводиться в консолі, що може стати в нагоді користувачам Terraform (наприклад, ви

будете знати, яку IP-адресу потрібно перевірити після розгортання вебсервера). Ви також можете ввести команду terraformoutput, щоб вивести список всіх вихідних значень без застосування будь-яких змін:

terraform output public_ip = 54.174.13.5

Щоб подивитися значення певної вихідної змінної, можна скористатися командою terraformoutput <ім'я_змінної>:

terraform output

public_ip 54.174.13.5

Це буде особливо корисно при написанні скриптів. Наприклад, ви можете створити скрипт розгортання, який розгортає вебсервер за допомогою команди terraformapply, бере його публічну IP-адресу з terraformoutputpublic_ip і звертається до цієї адреси, використовуючи curl. У підсумку вийде перевірка по гарячих слідах, яка підтвердить, що розгортання працює.

Вхідні і вихідні змінні також є невід'ємними складовими при створенні конфігуруємого інфраструктурного коду, придатного до повторного застосування.

3.2.7. Розгортання кластеру вебсерверів

Запуск одного сервера – гарний початок. Однак в реальному світі це означає наявність єдиної точки відмови. Якщо цей сервер вийде з ладу або перестане справлятися з навантаженням через занадто великий обсяг трафіку, користувачі не зможуть відкрити ваш сайт. В якості вирішення можна запустити кластер серверів: якщо один з них відмовить, запити допускається перенаправити до іншого сервера, а розмір самого кластера можна збільшувати і зменшувати в залежності від трафіка.

Ручне управління таким кластером потребує багато зусиль. На щастя, як показано на рис. 3.13, AWS може подбати про це за вас, використовуючи групу автомасштабування (англ. Auto scaling group, або ASG). ASG автоматично виконує безліч завдань, включаючи запуск кластера серверів EC2, моніторинг працездатності кожного сервера, заміну несправних серверів і зміна розміру кластера в залежності від навантаження.

Перше, що потрібно зробити при створенні ASG – це написати конфігурацію запуску, яка визначає, як потрібно налаштувати кожен сервер EC2 у вашій групі. Ресурс aws_launch_configuration використовує майже всі ті ж параметри, що і aws_instance (тільки у двох з них відрізняються імена: image_id замість ami і security_groups замість vpc_security_group_ids), тому ви можете їх легко замінити:

```
resource "aws_launch_configuration" "example"
{ image_id = "ami-0c55b159cbfafe1f0"
 instance_type  = "t2.micro"
security_groups = [aws_security_group.instance.id] user_data = <<-EOF
#!/bin/bash
```

```
echo "Hello, World" > index.html
nohup busybox httpd -f -p ${var.server_port} &
EOF
}
```

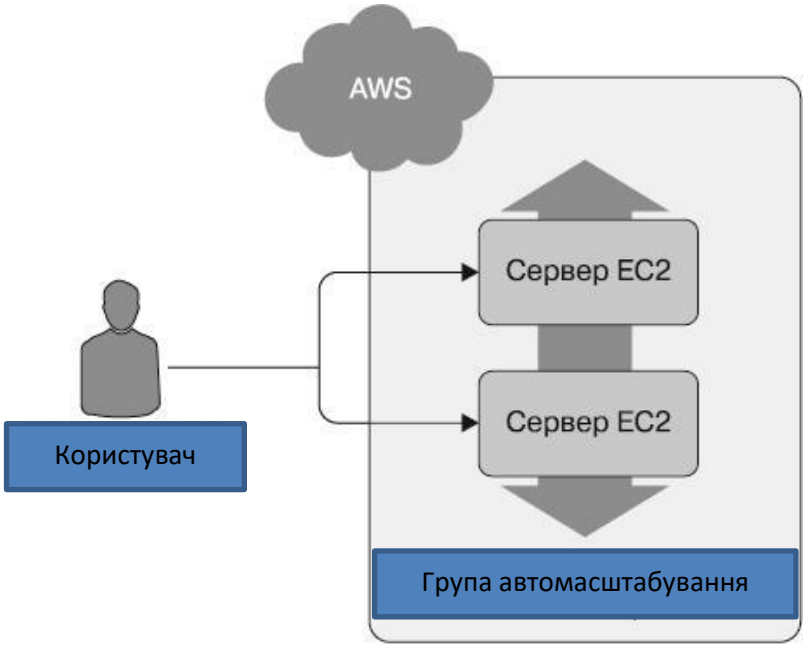


Рис. 3.13. Замість одного вебсервера група автомасштабування запускає кластер вебсерверів

Тепер ви можете створити саму групу ASG, використовуючи ресурс aws_autoscaling_group:

```
resource "aws_autoscaling_group" "example" {
launch_configuration = aws_launch_configuration.example.name
min_size = 2
max_size = 10
tag {
key    = "Name"
value  = "terraform-asg-example"
propagate_at_launch = true
  }
}
```

Ця група ASG включає в себе від двох до десяти серверів EC2 (спочатку запускається тільки два), кожен з яких має тег terraform-asg-example. Зверніть увагу, що ASG використовує посилання в якості імені конфігурації запуску. Це призводить до проблеми: конфігурація запуску незмінна, тому, якщо ви зміните будь-який її параметр, Terraform спробує замінити її цілком. Зазвичай, при заміні ресурсу, Terraform спочатку видаляє його стару версію і потім створює нову, але, оскільки ASG тепер посилається на старий ресурс, Terraform не зможе його видалити.

Щоб вирішити цю проблему, можна скористатися параметром життєвого циклу. Він підтримується всіма ресурсами Terraform і визначає їх

створення, оновлення і / або видалення. Особливо корисним параметром життєвого циклу є create_before_destroy. Якщо привласнити йому true, Terraform змінить порядок заміни ресурсів на протилежний. В результаті спочатку буде створена заміна (з відновленням всіх посилань таким чином, щоб вони вказували на неї, а не на старий ресурс) і тільки потім відбудеться видалення старого ресурсу. Додайте розділ lifecycle в aws_launch_configuration, як показано нижче:

```
resource "aws_launch_configuration" "example"
{ image_id = "ami-0c55b159cbfafe1f0"
 instance_type  = "t2.micro"
 security_groups = [aws_security_group.instance.id]
 user_data = <<-EOF
#!/bin/bash
echo "Hello, World" > index.html
nohup busybox httpd -f -p ${var.server_port} &
EOF
```

Необхідна для використання групи автомасштабування в конфігурації запуску.

https://www.terraform.io/docs/providers/aws/r/launch_configuration.html

```
 lifecycle {
   create_before_destroy = true
 }
}
```

Для роботи групи ASG потрібно ще один параметр: subnet_ids. Він визначає підмережі VPC, в яких повинні бути розгорнуті сервери EC2. Кожна підмережа знаходиться в ізольованій зоні доступності AWS (тобто в окремому обчислювальному центрі), тому, розгортаючи свої сервери за різними підмережами, ви гарантуєте, що ваш сервіс продовжить працювати, навіть якщо деякі з обчислювальних центрів вийдуть з ладу. Список підмереж можна прописати прямо в коді, але таке рішення буде складно підтримувати і переносити. Замість цього краще використовувати відповідне джерело даних для отримання списку підмереж, що належать до вашого профілю AWS.

Джерело даних являє собою фрагмент інформації, доступної суто для читання, який витягується з провайдера (в нашому випадку з AWS) при кожному запуску Terraform. Додаючи джерело даних в конфігурацію Terraform, ви не створюєте нічого нового. Це просто можливість запитати інформацію з API провайдера, щоб зробити її доступною для решти коду Terraform. Кожен провайдер надає цілий ряд джерел. Наприклад, провайдер AWS дозволяє запитувати дані про VPC і підмережі, ідентифікатори AMI, діапазони IP-адрес, ідентифікатор поточного користувача і багато іншого.

Синтаксис використання джерел даних дуже схожий на синтаксис ресурсу:

```
data "<PROVIDER> _ <TYPE>" "<NAME>" {[CONFIG ...] }
```

PROVIDER – ім’я провайдера (наприклад, aws), TYPE – тип джерела даних, який ви хочете використовувати (скажімо, vpc), NAME – ідентифікатор, за допомогою якого можна посилатися на це джерело даних в коді Terraform, а CONFIG складається з одного або декількох аргументів, передбачених спеціально для цього джерела. Ось як можна скористатися джерелом даних aws_vpc, щоб запросити інформацію по вашій хмарі VPC за замовчуванням:

```
data "aws_vpc" "default" {
default = true
}
```

Варто відзначити, що у випадку з джерелами даних в якості аргументів зазвичай передаються фільтри, які вказують на те, яку інформацію ви шукаєте. Шукаючи джерела даних aws_vpc, потрібно вказати лише один фільтр, default = true, який ініціює в ваш профіль AWS пошук VPC за замовчуванням.

Щоб отримати дані з джерела, потрібно використовувати наступний синтаксис доступу до атрибутів:

data. <PROVIDER> _ <TYPE>. <NAME>. <ATTRIBUTE>

Наприклад, щоб отримати ідентифікатор VPC з джерела даних aws_vpc, треба написати наступне:

```
data.aws_vpc.default.id
```

Ви можете додати до цього ще одне джерело даних, aws_subnet_ids, щоб знайти підмережі всередині цієї хмари VPC:

```
data "aws_subnet_ids" "default" {
vpc_id = data.aws_vpc.default.id
}
```

Ви можете отримати ідентифікатори підмереж з джерела aws_subnet_ids і скористатися аргументом з досить дивною назвою vpc_zone_identifier, щоб ваша група ASG використовувала ці підмережі:

```
resource "aws_autoscaling_group" "example" {
launch_configuration = aws_launch_configuration.example.name
vpc_zone_identifier = data.aws_subnet_ids.default.ids min_size = 2
 max_size = 10
tag {
key = "Name"
value = "terraform-asg-example"
propagate_at_launch = true
  }
}
```

3.2.8. Розгортання балансувальника навантаження

Ви вже навчилися розгортати групу ASG, але при цьому виникає невелика проблема: у вас є кілька серверів з окремими IP-адресами, однак

кінцевим користувачам зазвичай потрібна єдина точка входу. Одне з рішень полягає в розгортанні балансувальника навантаження, який буде розподіляти трафік між вашими серверами і надавати всім вашим користувачам власну IP-адресу (або, якщо бути точним, доменне ім'я). Створення балансувальника навантаження з високою доступністю і масштабованістю вимагає багато зусиль. І знову ви можете покластися на AWS, скориставшись сервісом Elastic Load Balancer (ELB) від Amazon (рис. 3.14).

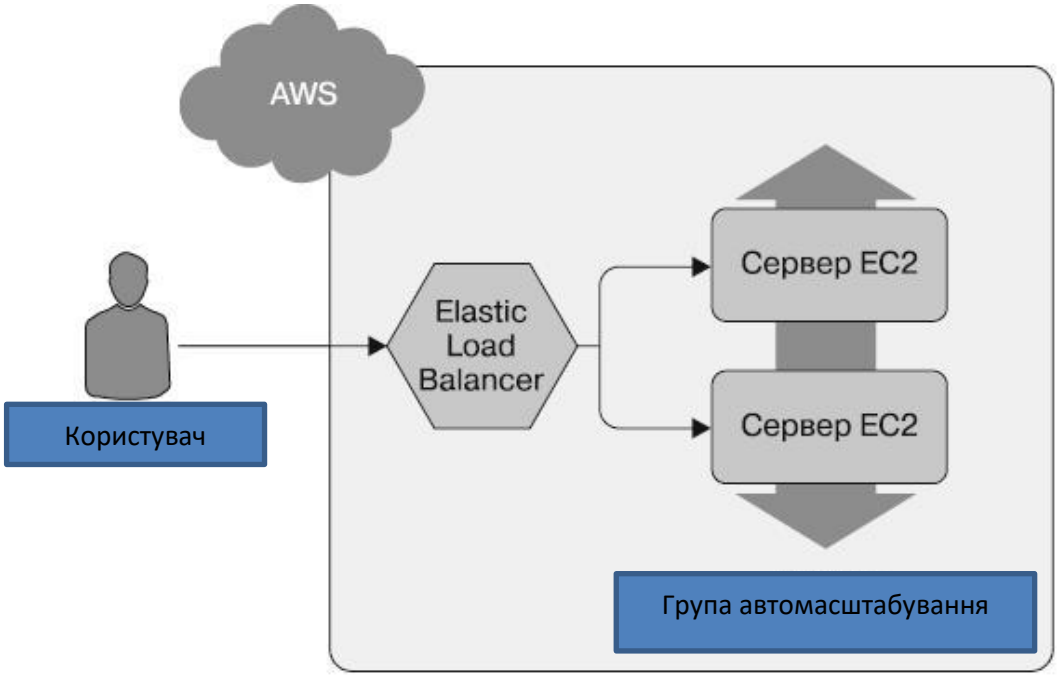


Рис. 3.14. Використання Amazon ELB для розподілу трафіку по групі автомасштабування

AWS пропонує три типи балансувальник навантаження:

Application Load Balancer (ALB). Найкраще підходить для балансування трафіку за протоколами HTTP і HTTPS. Працює на прикладному рівні (рівень 7) моделі OSI.

Network Load Balancer (NLB). Найкраще підходить для балансування трафіку за протоколами TCP, UDP і TLS. Якщо порівнювати з ALB, може швидше масштабуватися в обидва боки залежно від навантаження (NLB розрахований на масштабування до десятків мільйонів запитів у секунду). Працює на транспортному рівні (рівень 4) моделі OSI.

Classic Load Balancer (CLB). Це "старий" балансувальник навантаження, який з'явився раніше, ніж ALB і NLB. Він здатний працювати з трафіком за протоколами HTTP, HTTPS, TCP і TLS, але обмежений в можливостях у порівнянні зі своїми "наступниками". Працює як на прикладному, так і на транспортному рівні (L7 і L4) моделі OSI.

Сучасні програми повинні використовувати в основному ALB або NLB. Оскільки наш простий приклад з вебсервером є HTTP-додатком без серйозних вимог до продуктивності, нам найкраще підійде ALB.

Як показано на рис. 3.15, ALB складається з декількох частин:

Прослуховувач (слухач). Прослуховує певні порти (наприклад, 80) і протокол (скажімо, HTTP);

Правило прослуховувача. Бере запити, спрямовані до Прослуховувача, і передає ті з них, які відповідають певним шляхам (наприклад, / foo або / bar) або мережевим іменav (на кшталт foo.example.com або bar.example.com), заданих цільових груп;

Цільові групи. Один або кілька серверів, які приймають запити від балансувальника навантаження. Цільова група також стежить за працездатністю цих серверів і шле запити тільки “здоровим” вузлам.

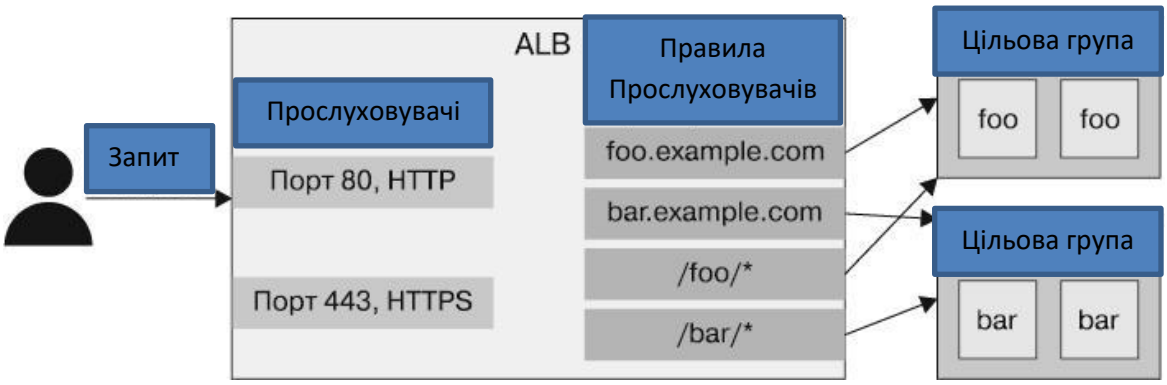


Рис. 3.15. Огляд Application Load Balancer (ALB)

Насамперед, потрібно створити сам балансувальник ALB, використовуючи ресурс aws_lb:

```
resource "aws_lb" "example" {
  name = "terraform-asg-example"
  load_balancer_type = "application"
  subnets = data.aws_subnet_ids.default.ids
}
```

Зверніть увагу, що параметр subnets налаштовує балансувальник навантаження для використання всіх підмереж у вашій хмарі VPC за замовчуванням за допомогою джерела даних aws_subnet_ids33. Балансувальник навантаження в AWS складається не з одного, а відразу з декількох серверів, які можуть працювати в різних підмережах (і, отже, в окремих обчислювальних центрах). AWS автоматично масштабує кількість балансувальників в залежності від трафіку і відпрацьовує відмова, якщо один з цих серверів вийде з ладу. Таким чином, ви отримуєте як масштабованість, так і високу доступність.

Наступним кроком буде визначення Прослуховувача для цього ALB з допомогою ресурсу aws_lb_listener:

```
resource "aws_lb_listener" "http" {
load_balancer_arn = aws_lb.example.arn
port = 80
protocol = "HTTP"
```

За замовчуванням повертає просту сторінку з кодом 404 default_action {

```
type = "fixed-response"
fixed_response {
content_type = "text / plain"
 message_body = "404: page not found" status_code = 404
  }
 }
}
```

Цей прослуховувач налаштовує ALB для прослуховування стандартного HTTP-порту (80), використання HTTP-протоколу і повернення простої сторінки з кодом 404 у разі, якщо запит не відповідає жодному з правил прослуховування.

Варто відзначити, що за замовчуванням для всіх ресурсів AWS, включаючи ALB, закритий вхідний і вихідний трафік, тому потрібно створити нову групу безпеки спеціально для балансувальника навантаження. Ця група повинна дозволяти вхідні запити на порт 80, щоб ви могли звертатися до ALB по HTTP, а також вихідні запити на всіх портах, щоб балансувальник міг перевіряти працездатність:

```
resource "aws_security_group" "alb" {
name = "terraform-example-alb"
Дозволяємо всі вхідні HTTP-запити ingress {
from_port = 80
to_port        = 80
protocol       = "tcp"
cidr_blocks = ["0.0.0.0/0"]
  }
        Дозволяємо всі вихідні запити egress {
from_port  = 0
to_port        = 0
protocol       = "-1"
cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Потрібно зробити так, щоб ресурс aws_lb використовував зазначену групу безпеки. Для цього встановіть аргумент security_groups:

```
resource "aws_lb" "example" {
name = "terraform-asg-example"
load_balancer_type = "application"
subnets = data.aws_subnet_ids.default.ids
security_groups = [aws_security_group.alb.id]
}
```

Потім необхідно створити цільову групу для ASG, використовуючи ресурс

aws_lb_target_group:
resource "aws_lb_target_group" "asg" {
name = "terraform-asg-example"
port = var.server_port
protocol = "HTTP"
vpc_id = data.aws_vpc.default.id
health_check {
path = "/"
matcher = "200"
interval = 15
timeout = 3
healthy_threshold = 2
unhealthy_threshold = 2
}
}

Зверніть увагу на те, що дана цільова група буде перевіряти працездатність ваших серверів, відправляючи їм періодично HTTP-запити; сервер вважається працездатним, тільки якщо його відповідь збігається з заданим співставлювачем (наприклад, ви можете зробити так, щоб співставлювач очікував відповіді 200OK). Якщо сервер не відповідає (можливо, через перебої в роботі або перевантаження), він буде позначений як непрацездатний і цільова група автоматично припинить відправляти йому трафік, щоб мінімізувати порушення обслуговування ваших користувачів.

Але звідки цільова група знає, яким серверам EC2 слід відправляти запити? Ви могли б прописати в ній статичний список серверів ресурсом aws_lb_target_group_attachment, однак при роботі з ASG сервери можуть запускатися і віддалятися в будь-який момент, тому такий підхід не годиться. Замість цього можна скористатися першокласною інтеграцією між ASG і ALB. Поверніться до ресурсу aws_autoscaling_group

зробіть так, щоб його аргумент target_group_arns вказував на нову цільову групу:

resource "aws_autoscaling_group" "example" {
launch_configuration = aws_launch_configuration.example.name vpc_zone_identifier = data.aws_subnet_ids.default.ids target_group_arns = [aws_lb_target_group.asg.arn] health_check_type = "ELB"
min_size = 2 max_size = 10
tag {
key = "Name"
value = "terraform-asg-example"
propagate_at_launch = true
}
}

Ви також повинні поміняти значення health_check_type на "ELB". Значення за замовчуванням, "EC2", має на увазі мінімальну перевірку працездатності, яка вважає сервер несправним, тільки якщо гіпервізор AWS стверджує, що ВМ зовсім не працює або є недоступною. Значення "ELB" надійніше, оскільки змушує ASG перевіряти працездатність цільової групи. До того ж сервери автоматично замінюються, якщо цільова група оголошує їх несправними. Таким чином, сервери підлягають заміні не тільки в разі повної відмови, але і коли вони, наприклад, перестають обслуговувати запити через брак пам'яті або зупинки критично важливого процесу.

Прийшов час зібрати це все докупи. Для цього ми створимо Правило прослуховувача, використовуючи ресурс aws_lb_listener_rule:

```
resource "aws_lb_listener_rule" "asg" {
listener_arn = aws_lb_listener.http.arn
priority = 100
condition {
field = "path-pattern"
values = [ "*"]
}
action {
type = "forward"
target_group_arn = aws_lb_target_group.asg.arn
}
}
```

Цей код додає Правило прослуховувача, яке відправляє запити, що відповідають будь-якому шляху до цільової групи з ASG всередині.

Перш ніж розгортати балансувальник навантаження, потрібно зробити ще дещо – поміняти старий вивід public_ip одного сервера EC2 на вивід доменного імені ALB:

```
output "alb_dns_name" {
value = aws_lb.example.dns_name
description = "The domain name of the load balancer"
}
```

Виконайте terraformapply і почитайте отриманий план. Згідно з ним наш вихідний сервер EC2 видаляється, а замість нього Terraform створює конфігурацію запуску, ASG, ALB і групу безпеки. Якщо з планом все в порядку, введіть yes і натисніть клавішу Enter. Коли команда apply завершить роботу, ви повинні побачити висновок alb_dns_name:

```
Outputs:
alb_dns_name=terraform-asg-example-123.us-east-2.elb.amazonaws.com
```

Скопіюйте цю URL-адресу. Зачекайте, поки наші сервери завантажаться, ALB позначить їх як працездатні. Тим часом можете переглянути те, що ви розгорнули. Відкривши розділ ASG консолі EC2 (https://amzn.to/2MH3mId), ви повинні побачити, що група автомасштабування вже створена (рис. 3.16).

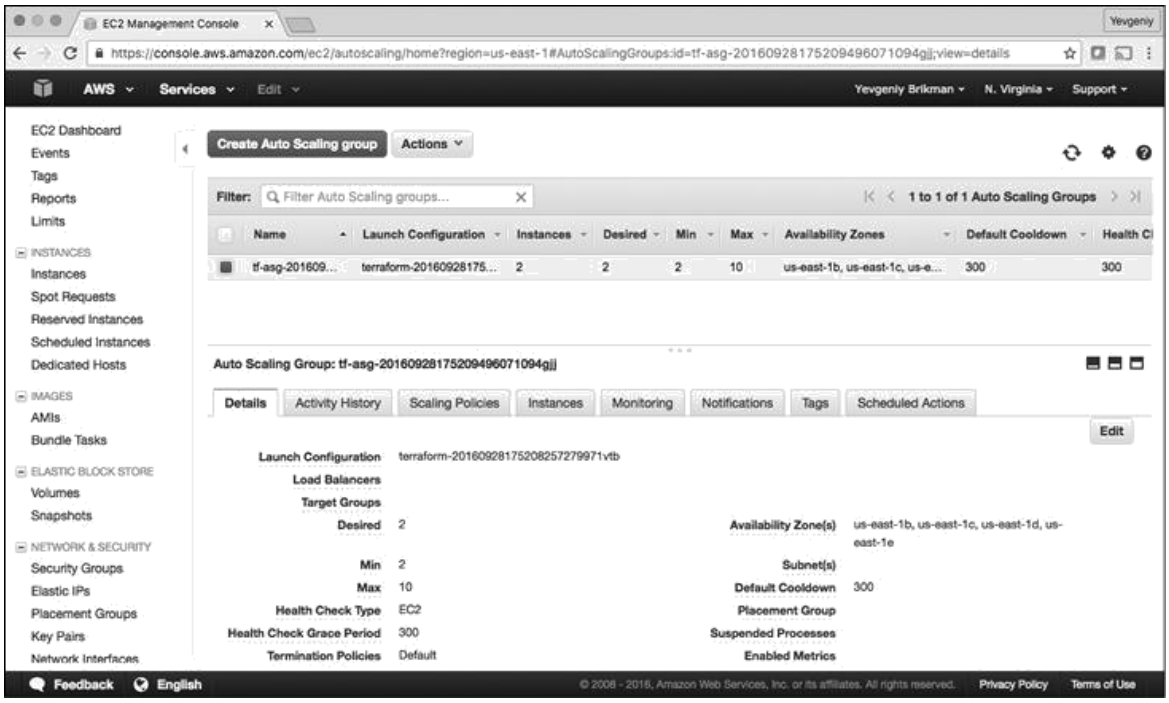


Рис. 3.16. Група автомасштабування

Якщо перейти на вкладку Instances (Сервери), можна побачити запуск двох серверів EC2, як показано на рис. 3.17.

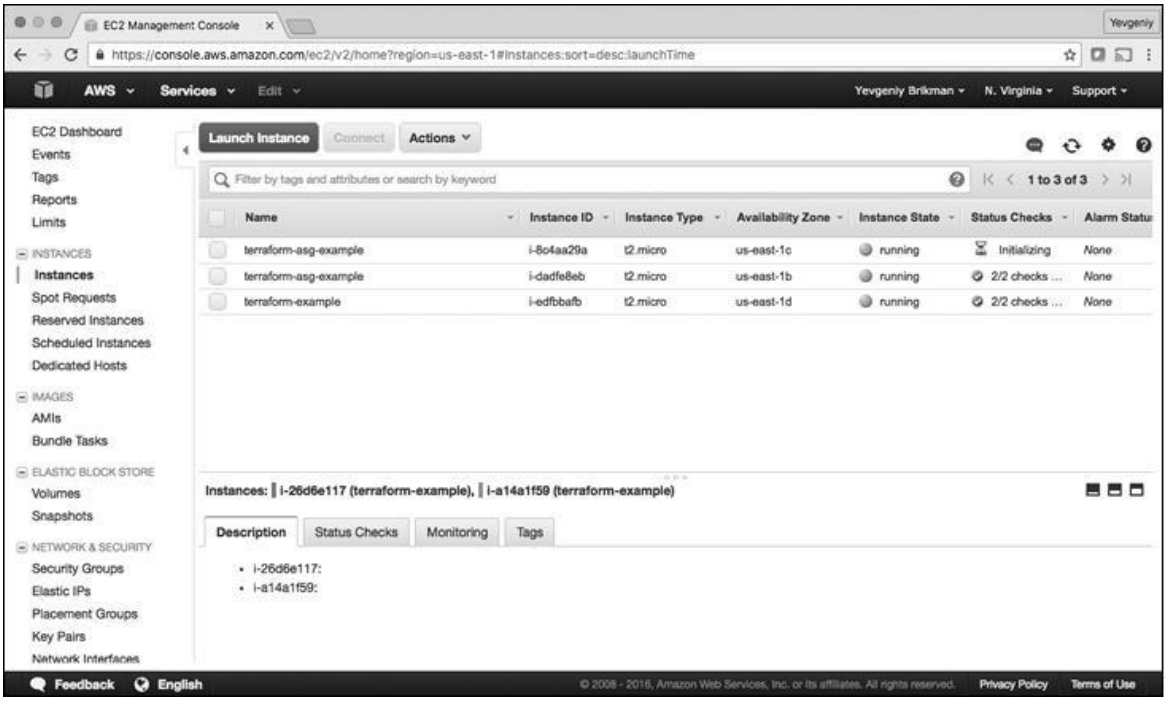


Рис. 3.17. Запуск серверів EC2 в ASG

Клацнувши на вкладці Load Balancers (балансувальник навантаження), ви побачите свій екземпляр ALB, як показано на рис. 3.18.

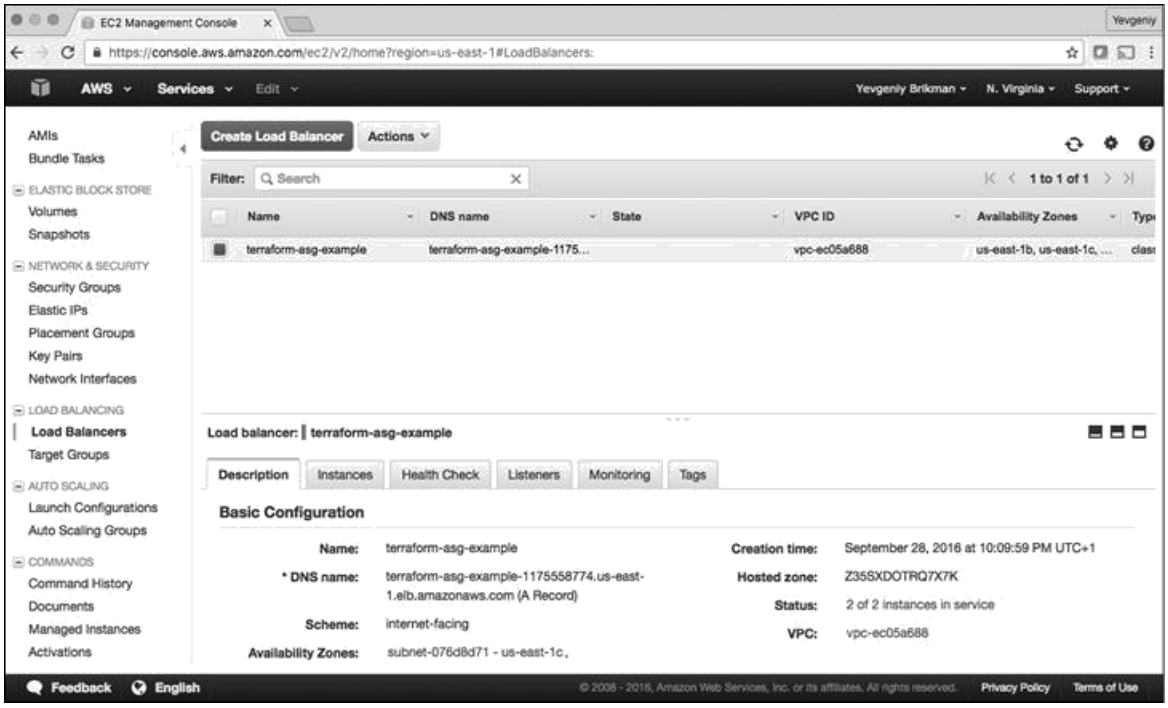


Рис. 3.18. Application Load Balancer

Щоб знайти цільову групу, як показано на рис. 3.19, перейдіть на вкладку Target Groups (Цільові групи).

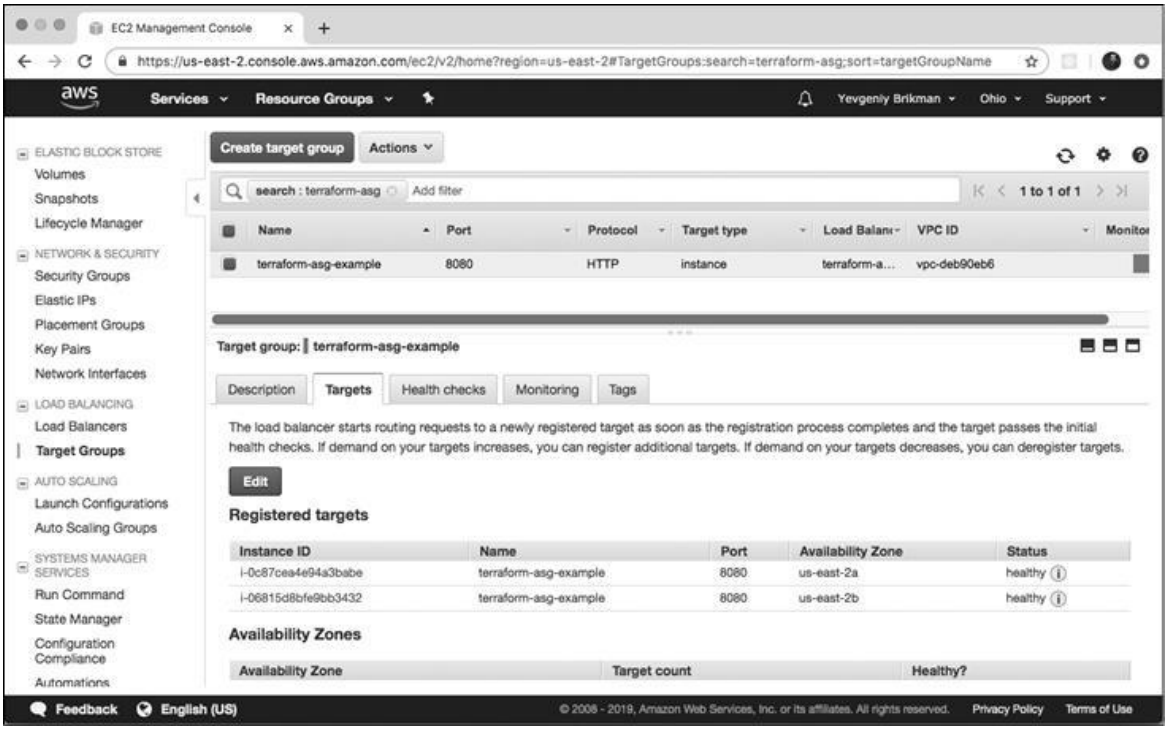


Рис. 3.19. Цільова група

Якщо ви виберете свою цільову групу і клацнете на вкладці Targets (Цілі) в нижній частині екрана, то зможете побачити, як ваші сервери реєструються в цільовій групі і проходять перевірку працездатності.

Зачекайте, поки їх індикатор стану не почне показувати healthy. Це зазвичай займає одну-дві хвилини. Після перевірте висновок alb_dns_name, який ви скопіювали раніше:

curl http: // <alb_dns_name> Hello, World

Вийшло! ALB направляє трафік до ваших серверів EC2. Кожного разу, коли ви звертаєтеся за цією URL-адресою, він вибирає інший сервер для обробки запиту. Ви отримали повністю робочий кластер вебсерверів!

На цьому етапі можна бачити, як ваш кластер реагує на створення нових і видалення старих серверів. Наприклад, перейдіть на вкладку Instances (Сервери) і видаліть один з серверів: встановіть прапорець, натисніть вгорі кнопку Actions (Дії) і потім поміняйте стан сервера на Terminate (Видалити). Якщо ви знову спробуєте звернутися за URL-адресою ALB, кожен ваш запит повинен отримати відповідь 200OK, навіть після видалення сервера. ALB автоматично виявить, що сервера більше немає, і перестане направляти до нього трафік. Що ще цікаво, незабаром після видалення ASG зрозуміє, що у нас залишився один сервер замість двох, і автоматично запустить заміну (самовідновлення!). Щоб побачити, як ASG змінює свій розмір, можете додати в свій код Terraform параметр desired_capacity і знову виконати команди apply.

3.2.9. Видалення непотрібних ресурсів

Коли ви закінчите експериментувати з Terraform, бажано видалити всі створені вами ресурси, щоб за них не довелося платити. Оскільки Terraform відстежує все, що ви створюєте, це не складе проблеми. Досить виконати команду destroy:

terraform destroy (...)

Terraform will perform the following actions:

aws_autoscaling_group.example will be destroyed - resource "aws_autoscaling_group" "example" {

(...)

}

aws_launch_configuration.example will be destroyed - resource "aws_launch_configuration" "example" {

(...)

}

aws_lb.example will be destroyed - resource "aws_lb" "example" {

(...)

}

(...)

Plan: 0 to add, 0 to change, 8 to destroy. Do you really want to destroy all resources?

Terraform will destroy all your managed infrastructure, as shown above.

There is no undo. Only 'yes' will be accepted to confirm.

Enter a value:

Ви, напевно, і самі розумієте, що команда destroy в промисловому середовищі повинна використовуватися рідко (або взагалі ніколи). Її не можна скасувати, тому Terraform дасть вам останній шанс поглянути на свої дії, відобразивши на екрані список всіх ресурсів, які будуть видалені, і запросить у вас підтвердження. Якщо все виглядає добре, введіть yes і натисніть клавішу Enter. Terraform побудує граф залежностей і видалить всі ваші ресурси в коректному порядку, як можна сильніше розпаралеливши цей процес. Через одну-дві хвилини ваш обліковий запис AWS знову буде порожній.

Принадність концепції IaC в тому, що вся інформація про ресурси описана в коді, тому при бажанні ви можете відтворити їх всі за допомогою однієї команди: terraformapply. Насправді останні внесені зміни краще зафіксувати в Git, щоб ви могли відслідковувати історію своєї інфраструктури.

Резюме

У вас тепер є загальне уявлення про використання Terraform. Декларативна мова дозволяє легко описувати саме ту інфраструктуру, яку ви хочете створити. За допомогою команди plan можна переглянути потенційні зміни і усунути помилки, перш ніж приступати до розгортання. Змінні, посилання і залежності дозволяють позбавити ваш код від повторюваних фрагментів і зробити його максимально конфігурованим.

Якщо технології AWS здаються вам заплутаними, обов’язково почитайте статтю AWS in Plain English за адресою expeditedsecurity.com/aws-in-plain-english/.

Детальніше про рекомендовані підходи до управління користувачами в AWS можна почитати за посиланням amzn.to/2lvJ8Rf.

Більше про політики IAM можна дізнатися на сайті AWS за адресою docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html.

Код Terraform можна також писати на чистому JSON і зберігати в файлах з розширенням .tf.json. Більше про синтаксис HCL і JSON можна дізнатися на сайті Terraform: bit.ly/2MDZyaN.

На GitHub за адресою bit.ly/2OIIrr2 можна знайти корисний список однорядкових HTTP-серверів.

Більше про принцип роботи CIDR можна дізнатися на сторінці в "Вікіпедії" за адресою bit.ly/2l8Ki9g. Для перетворення діапазонів IP-адрес в CIDR і назад можете використовувати вебкалькулятор cidr.xyz або встановити команду ipcalc в своєму терміналі.

Докладніше обговорення побудови високодоступних і масштабованих систем в AWS наводиться в статті за адресою bit.ly/2mpSXUZ.

Щоб не ускладнювати ці приклади, ми запускаємо сервери EC2 і ALB в одній підмережі. У промислових умовах їх розміщують в різних підмережах: сервери EC2 в закритій (щоб вони не були доступні безпосередньо з Інтернету), а ALB – у відкритій (щоб користувачі могли звертатися до нього безпосередньо).

3.2.10. Як керувати станом Terraform

В розглянутому вище матеріалі, використовуючи Terraform для створення і відновлення ресурсів, ви могли помітити, що при кожному виконанні команд terraformplan і terraformapply цій системі вдавалося знаходити створені раніше ресурси і оновлювати їх відповідним чином. Але звідки їй було відомо про те, які з них знаходяться під її управлінням? У вашого профілю AWS може перебувати будь-яка інфраструктура, розгорнута за допомогою різних механізмів (частково вручну, частково через Terraform, частково за допомогою утиліти командного рядка). Так як же Terraform визначає свої ресурси?

В нижченаведеному матеріалі ви дізнаєтеся, як Terraform відстежує стан вашої інфраструктури і яким чином це впливає на структуру файлів і каталогів, ізоляцію і блокування в проекті Terraform. Ось ключові теми, за якими ми пройдемося:

- Що собою являє стан Terraform;
- Загальна сховище для файлів стану;
- Обмеження внутрішніх сховищ Terraform;
- Ізоляція файлів стану;
- Ізоляція за допомогою робочих областей;
- Ізоляція за допомогою опису структури файлів;
- Джерело даних terraform_remote_state.

Що собою являє стан Terraform

При кожному своєму запуску система Terraform записує інформацію створеної нею інфраструктури в свій файл стану. За замовчуванням, якщо запуск відбувається в /foo/bar, Terraform створює файл /foo/bar/terraform.tfstate. Цей файл має нестандартний формат JSON і пов'язує ресурси Terraform в ваших конфігураційних файлах з їх поданням в реальному світі. Уявіть, наприклад, що у Terraform наступна конфігурація:

```
resource "aws_instance" "example" {
ami = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
}
```

Нижче показано невеликий фрагмент файла terraform.tfstate (урізаний, щоб його було легше читати), який буде створений після виконання terraformapply:

```
{
"Version": 4,
"Terraform_version": "0.12.0",
"Serial": 1,
"Lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
 "Outputs": {},
"Resources": [
{
        "Mode": "managed",
        "Type": "aws_instance",
        "Name": "example",
        "Provider": "provider.aws",
        "Instances": [
        {
                "Schema_version": 1,
                "Attributes": {
                "Ami": "ami-0c55b159cbfafe1f0",
                "Availability_zone": "us-east-2c",
                "Id": "i-00d689a0acc43af0f",
                "Instance_state": "running",
                "Instance_type": "t2.micro",
                "(...)": "(truncated)"
                }
    }
  ]
```

```
    }
  ]
}
```

Завдяки формату JSON Terraform знає, що ресурс типу aws_instance з ім'ям example відповідає серверу EC2 з ідентифікатором i-00d689a0acc43af0f у вашому обліковому записі AWS. При кожному запуску Terraform може вимагати від AWS поточний стан цього сервера і порівнювати його з вашою конфігурацією, щоб визначити, які зміни слід внести. Іншими словами, виведення команди plan – це розбіжність між кодом на вашому комп'ютері і інфраструктурою, яка розгорнута в реальному світі (згідно з ідентифікаторами в файлі стану).

Файл стану є приватним API

Файл стану – це приватний API, який змінюється з кожним новим випуском і призначений суто для внутрішнього використання в Terraform. Ви ніколи не повинні редагувати його вручну або зчитувати його безпосередньо.

Якщо вам з якоїсь причини потрібно модифікувати файл стану (що має бути рідкістю), використовуйте команди terraform import або terraform state.

Якщо ви використовуєте Terraform в особистому проєкті, можна спокійно зберігати файл terraform.tfstate локально на своєму комп'ютері. Але якщо ви ведете командну розробку реального продукту, може виникнути кілька проблем.

Загальна сховище для файлів стану. Щоб оновлювати інфраструктуру за допомогою Terraform, у кожного члена команди повинен бути доступ до одних і тих же файлів стану. Це означає, що вам потрібно зберігати ці файли в загальнодоступному місці.

Блокування файлів стану. Поділ даних відразу ж створює нову проблему: блокування. Якщо два члени команди запускають Terraform одночасно, може виникнути стан гонки, так як оновлення файлів стану відбувається паралельно з боку двох різних процесів. Без блокування це може привести до конфліктів, втрати даних і пошкодження файлів стану.

Ізоляція файлів стану. При зміні інфраструктури рекомендується ізолювати різні оточення. Наприклад, при виправленні стану в середовищі попереднього або фінального тестування слід переконатися, що це ніяк не зашкодить промислової системі.

Загальне сховище для файлів стану

Найпоширеніший метод, який дозволяє декільком членам команди працювати з загальним набором файлів, полягає в використанні системи

управління версіями (наприклад, Git). Хоча ваш код Terraform точно повинен зберігатися саме таким чином, застосування того ж підходу до стану Terraform – погана ідея з кількох причин.

Людський фактор. Ви можете легко забути завантажити останні зміни з системи управління версіями перед запуском Terraform або зберегти свої власні поновлення постфактум. Рано чи пізно хтось у вашій команді випадково запустить Terraform з застарілими файлами стану, що приведе до зниження або дублювання вже розгорнутих ресурсів.

Блокування. Більшість систем управління версіями не пропонують жодних коштів блокування, які могли б запобігти одночасного виконання terraformapply двома різними членами команди.

Наявність конфіденційних даних. Всі дані в файлах стану Terraform зберігаються у вигляді звичайного тексту. Це може спричинити проблеми, оскільки деяким ресурсам Terraform необхідно зберігати чутливі дані. Наприклад, якщо ви створюєте базу даних за допомогою ресурсу aws_db_instance, Terraform збереже ім'я користувача і пароль до неї в файлі стану у відкритому вигляді. Відкрите зберігання конфіденційних даних, де б то не було, включаючи систему управління версіями, – погана ідея. Станом на травень 2019 року спільнота Terraform обговорює відкриту заявку (http://bit.ly/33gqaVe), створену з цього приводу, хоча для вирішення даної проблеми є обхідні шляхи, які ми незабаром обговоримо.

Замість системи управління версіями, для спільного управління файлами стану, краще використовувати віддалені сховища, підтримка яких вбудована в Terraform. Сховище визначає те, як Terraform завантажує і зберігає свій стан. За замовчуванням для цього застосовується локальне сховище, з яким ви працювали весь цей час. Воно зберігає файли стану на вашому локальному диску. Але підтримуються також віддалені сховища з можливістю спільного доступу. Серед них можна виділити Amazon S3, Azure Storage, Google Cloud Storage і такі продукти, як Terraform Cloud, Terraform Pro і Terraform Enterprise від HashiCorp.

Дистанційні сховища вирішують всі три проблеми, перераховані вище.

Людський фактор. Після конфігурації віддаленого сховища Terraform буде автоматично завантажувати з нього файл стану при кожному виконанні команд plan і apply і аналогічно зберігати його після виконання apply. Таким чином, можливість ручної помилки виключається.

Блокування. Більшість вилучених сховищ мають вбудовану підтримку блокування. При виконанні terraform apply, Terraform автоматично встановлює блокування. Якщо в цей момент дану команду виконує хтось

інший, блокування вже встановлено і вам доведеться почекати. Команду apply можна ввести з параметром -lock-timeout = <TIME>. Так Terraform знатиме, скільки часу потрібно чекати зняття блокування (наприклад, якщо вказати -lock-timeout = 10m, очікування триватиме десять хвилин).

Конфіденційні дані. Більшість вилучених сховищ мають вбудовану підтримку активного і пасивного шифрування файлів стану. Більш того, вони зазвичай дозволяють налаштовувати права доступу (наприклад, при використанні політик IAM в поєднанні з Бакета Amazon S3), щоб ви могли управляти тим, хто може звертатися до ваших файлів стану і конфіденційних даних, які можуть в них перебувати. Звичайно, було б краще, якби в Terraform підтримувалося шифрування конфіденційних даних прямо в файлах стану, але ці віддалені сховища мінімізують більшість ризиків безпеки (файл стану не зберігається у відкритому вигляді де-небудь на вашому диску).

Якщо ви використовуєте Terraform в зв'язці з AWS, кращим вибором в якості віддаленого сховища буде S3, керований сервіс зберігання файлів від Amazon. Цьому є кілька причин:

це керований сервіс, тому для його використання непотрібно розгортати і обслуговувати додаткову інфраструктуру;

він розрахований на +99,999999999%-ву стійкість і 99,99%-ву доступність. Це означає, що вам не варто сильно хвилюватися про втрату даних і перебоях у роботі;

він підтримує шифрування, що знижує ризик зберігання чутливих даних у файлах стану. Хоча це лише часткове вирішення, так як будь-який член вашої команди з доступом до Бакета S3 зможе переглядати ці файли в відкритому вигляді, але так дані будуть шифруватися при збереженні (Amazon S3 підтримує шифрування на стороні сервера за допомогою AES-256) і передачі (Terraform використовує SSL для читання і запису даних в Amazon S3);

він підтримує блокування за допомогою DynamoDB (докладніше про це чуть пізніше);

він підтримує управління версіями, тому ви зможете зберігати кожну ревізію свого стану і в разі виникнення проблем відкочуватися на старішу версію;

він недорогий, тому більшість сценаріїв застосування Terraform легко вписуються в безкоштовний тарифний план.

Щоб включити віддалене зберігання стану в Amazon S3, для початку потрібно підготувати Бакет S3. Створіть файл main.tf в новій папці (це не

повинна бути папка, в якій ви зберігаєте конфігурацію) вгорі вкажіть AWS як провайдера:

```
provider "aws" {
region = "us-east-2"
}
```

Потім створіть Бакет S3, використовуючи ресурс aws_s3_bucket:

```
resource "aws_s3_bucket" "terraform_state" {bucket = "terraform-up-and-running-state"
```

Запобігаємо випадкове видалення цього Бакета S3 lifecycle {

```
prevent_destroy = true
}
```

Включаємо управління версіями, щоб ви могли переглядати всю історію ваших файлів стану

```
versioning {
enabled = true
}
```

Включаємо шифрування за умовчанням на стороні сервера server_side_encryption_configuration {

```
rule {
apply_server_side_encryption_by_default {sse_algorithm = "AES256"
   }
  }
 }
}
```

Цей код встановлює чотири аргументи:

bucket. Це ім’я Бакета S3. Майте на увазі, що імена Бакета повинні бути унікальними на глобальному рівні серед усіх клієнтів AWS. Тому замість "terraform-up-and-running-state" ви повинні підібрати свою власну. Запам’ятайте його і зверніть увагу на те, який регіон AWS ви використовуєте: трохи пізніше вам знадобляться обидва ці фрагменти інформації;

prevent_destroy. Це другий параметр життєвого циклу, з яким ви зустрічаєтеся. Якщо привласнити йому true, при спробі видалення відповідного ресурсу (наприклад, при виконанні terraformdestroy) Terraform поверне помилку. Це дозволяє запобігти випадковому видаленню важливих ресурсів, таких як Бакет S3 з усім вашим станом Terraform. Звичайно, якщо ви дійсно хочете його видалити, просто закоментуйте цей параметр;

versioning. В даному параметрі включається управління версіями Бакета S3, в результаті чого кожне оновлення зберігається та створює його

нову версію. Це дозволяє переглядати старі версії та повертатися до них в будь-який момент;

server_side_encryption_configuration. В цьому параметрі включається шифрування за замовчуванням на стороні сервера для всіх даних, які записуються в Бакет S3. Завдяки цьому ваші файли стану і будь-які конфіденційні дані, які можуть в них міститися, завжди будуть шифруватися при збереженні в S3.

Далі потрібно створити таблицю DynamoDB, яка буде використовуватися для блокування. DynamoDB – це розподілене сховище типу “ключ – значення” від Amazon. Воно підтримує узгоджене читання і умовний запис – все, що необхідно для розподіленої системи блокування. Більш того, воно повністю кероване, тому вам непотрібно займатися ніякою додатковою інфраструктурою, а більшість сценаріїв застосування Terraform легко впишуться в безкоштовний тарифний план.

Щоб використовувати DynamoDB для блокування в поєднанні з Terraform, потрібно створити таблицю з первинним ключем під назвою LockID (з написанням). Це можна зробити за допомогою ресурсу

aws_dynamodb_table:

```
resource "aws_dynamodb_table" "terraform_locks" {
name = "terraform-up-and-running-locks"
billing_mode = "PAY_PER_REQUEST"
hash_key = "LockID"
attribute {
name = "LockID"
type = "S"
}
}
```

Виконайте terraforminit, щоб завантажити код провайдера, а потім terraformapply, щоб розгорнути ресурси. Примітка: щоб мати можливість розгортати цей код, у вашого користувача IAM повинні бути права на створення Бакета S3 і таблиць DynamoDB. Після завершення розгортання ви отримаєте Бакет S3 і таблицю DynamoDB, але ваш стан Terraform як і раніше буде зберігатися локально. Щоб зберігати його в Бакета S3 (з шифруванням і блокуванням), потрібно додати свій код розділ backend. Це конфігурація самої системи Terraform, тому вона знаходиться всередині блоку terraform і має наступний синтаксис:

```
terraform {
backend "<BACKEND_NAME>" {[CONFIG ...]
}
```

}

BACKEND_NAME – це ім’я сховища, яке ви хочете використовувати (наприклад, "s3"), а CONFIG містить один або кілька аргументів, передбачених спеціально для цього сховища (скажімо, ім’я Бакета S3, який потрібно використовувати). Так виглядає конфігурація backend для Бакета S3:

terraform {

backend "s3" {

Поміняйте це на ім'я свого Бакета!

bucket = "terraform-up-and-running-state"

key = "global / s3 / terraform.tfstate"

region = "us-east-2"

Замініть це ім’ям своєї таблиці DynamoDB! dynamodb_table = "terraform-up-and-running-locks"

encrypt = true

}

}

Пройдемося по цих параметрах:

bucket. Ім’я потрібного Бакета S3. Не забудьте поміняти його на назву створеного раніше Бакета;

key. Файловий шлях всередині Бакета S3, за яким Terraform буде записувати файл стану. Пізніше ви побачите, чому в попередньому прикладі цього параметра присвоєно global / s3 / terraform.tfstate;

region. Регіон AWS, в якому знаходиться Бакет S3. Не забудьте вказати регіон того Бакета, який ви створили раніше;

dynamodb_table. Таблиця DynamoDB, яка буде використовуватися для блокування. Не забудьте вказати ім’я тієї таблиці, яку ви створили раніше;

encrypt. Якщо вказати true, стан Terraform шифруватиметься при збереженні в S3. Це додаткова міра, яка гарантує шифрування даних у всіх ситуаціях, так як ми вже включили шифрування за умовчанням для S3.

Щоб стан Terraform зберігся в цьому Бакеті, потрібно знову виконати terraforminit. Ця команда не тільки завантажить код провайдера, але сконфігурує сховище Terraform (ще одне її застосування ви побачите трохи пізніше). Більш того, вона ідемпотентна, тому її повторне виконання безпечно:

$ Terraform init Initializing the backend ...

Acquiring state lock. This may take a few moments ...

Do you want to copy existing state to the new backend?

Pre-existing state was found while migrating the previous "local" backend to the newly configured "s3" backend. No existing state was found in the newly configured "s3" backend. Do you want to copy this state to the new "s3" backend? Enter "yes" to copy and "no" to start with an empty state. Enter a value:

Terraform автоматично визначить, що у вас вже є локальний файл стану, і з вашого дозволу скопіює його в нове сховище S3. Якщо ввести yes, можна побачити наступне:

Successfully configured the backend "s3"! Terraform will automatically use this backend unless the backend configuration changes.

Після виконання цієї команди ваш стан Terraform буде збережено в Бакета S3. Щоб в цьому переконатися, відкрийте консоль управління S3 (https://amzn.to/ 2Kw5qAc) в своєму браузері і виберіть свій Бакет. Ви повинні побачити щось схоже на рис. 3.20.

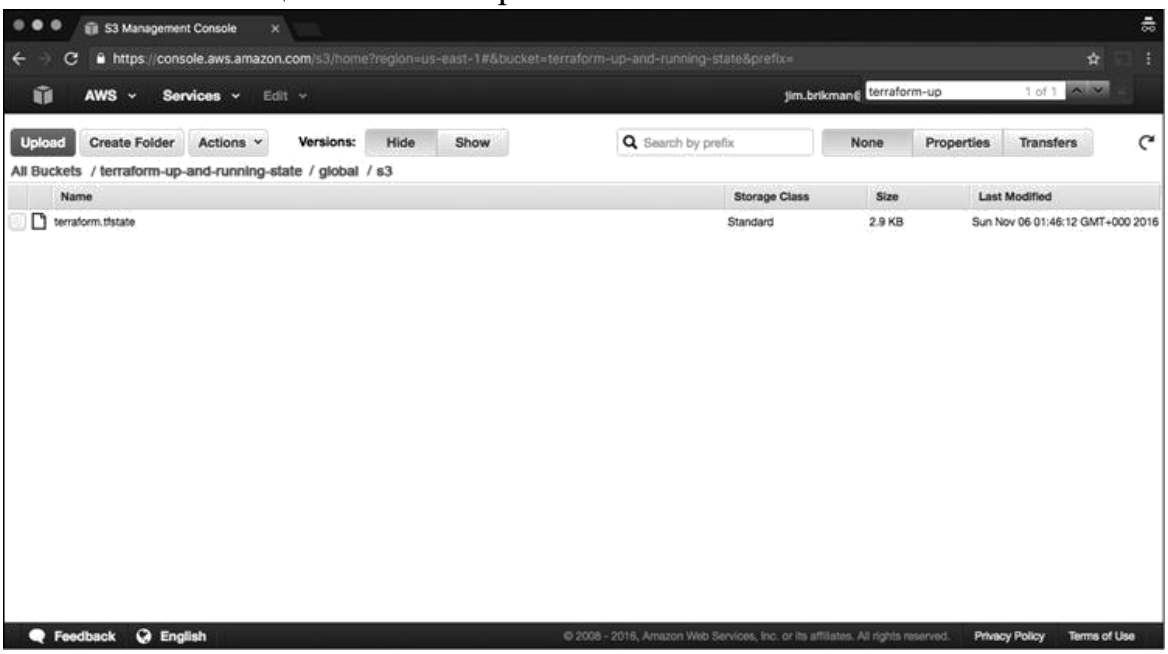


Рис. 3.20. Файл стану Terraform, що зберігається в S3

Після включення цього сховища Terraform буде автоматично завантажувати останній стан з Бакета S3 перед виконанням команди і зберігати його туди після того, як команда буде виконана. Щоб побачити, як це працює, додайте наступні вихідні змінні:

output "s3_bucket_arn" {

value = aws_s3_bucket.terraform_state.arn description = "The ARN of the S3 bucket"

}

output "dynamodb_table_name" {

value = aws_dynamodb_table.terraform_locks.name description = "The name of the DynamoDB table"

}

Ці змінні виведуть на екран ARN (Amazon Resource Name) вашого бакета S3 і ім’я вашої таблиці DynamoDB. Щоб в цьому переконатися, виконайте terraformapply:

$ Terraform apply

Acquiring state lock. This may take a few moments ...

aws_dynamodb_table.terraform_locks: Refreshing state ...

aws_s3_bucket.terraform_state: Refreshing state ...

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Releasing state lock. This may take a few moments ...

Outputs:

dynamodb_table_name = terraform-up-and-running-locks s3_bucket_arn = arn: aws: s3 ::: terraform-up-and-running-state

Зауважте, що тепер Terraform встановлює блокування перед запуском команди apply і знімає його після!

Ще раз зайдіть в консоль S3 за адресою https://amzn.to/2Kw5qAc, поновіть сторінку і натисніть сіру кнопку Show (Показати) поруч з написом Versions (Версії). На екрані має з’явитися кілька версій вашого файла terraform.tfstate, що зберігається в бакеті S3 (рис. 3.21).

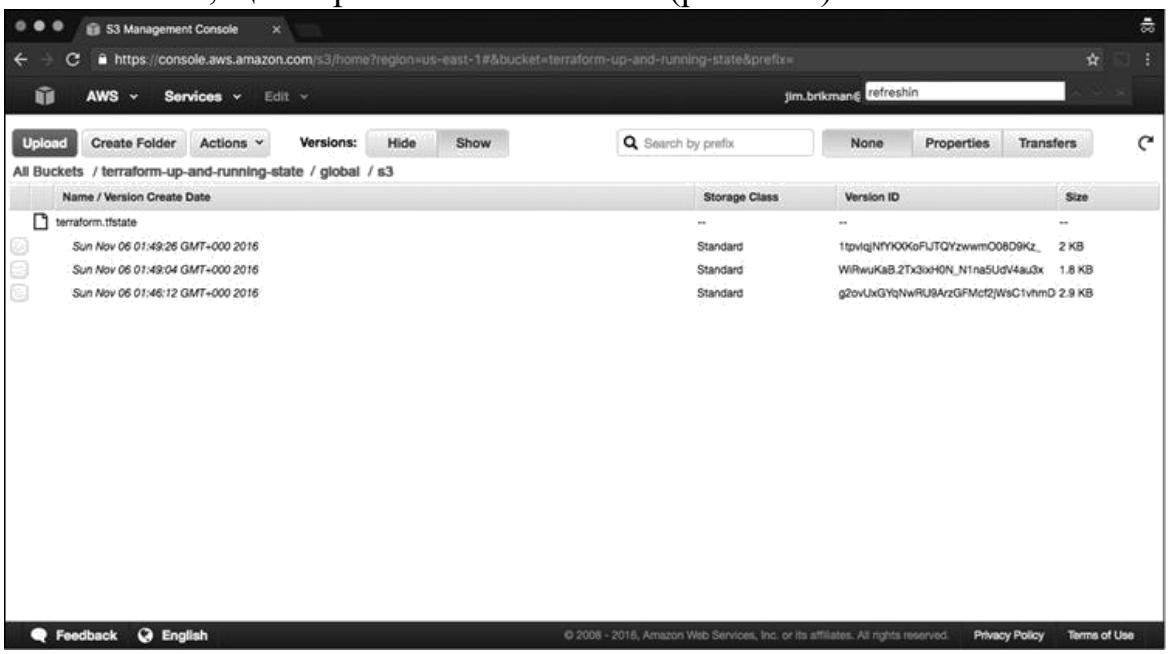


Рис. 3.21. Декілька версій стану Terraform в S3

Це означає, що Terraform дійсно завантажує дані стану S3 і з нього і ваш бакет зберігає кожну ревізію файла стану, що може стати в нагоді для налагодження і відкату до старішої версії, якщо щось піде не так.

Обмеження сховищ Terraform

В сховищ Terraform є кілька обмежень і підводних каменів, про які вам слід знати. Перш за все, коли ви використовуєте Terraform для

створення бакета S3, в якому ви хочете зберігати стан Terraform, це схоже на ситуацію з куркою і яйцем. Щоб цей підхід працював, вам доведеться виконати наступне:

1. Написати код Terraform, щоб створити Бакет S3 і таблицю DynamoDB, а потім розгорнути цей код з використанням локального сховища;

2. Повернутися до коду Terraform, додати в нього конфігурацію для віддаленого сховища, щоб застосувати свіжостворені Бакет S3 і таблицю DynamoDB та виконати команду terraforminit, щоб скопіювати ваший локальний стан в S3.

Якщо ви коли-небудь захочете видалити Бакет S3 і таблицю DynamoDB, доведеться виконати зворотні дії:

1. Перейти до коду Terraform, видалити конфігурацію backend і знову виконати команду terraforminit, щоб скопіювати стан Terraform назад на локальний диск.

Виконати terraformdestroy, щоб видалити Бакет S3 і таблицю DynamoDB.

Цей двоступінчастий процес досить непростий, зате ви можете використовувати у всій своїй коді Terraform одні і ті ж Бакет S3 і таблицю DynamoDB, тому потрібно виконати його лише раз (або один раз для кожного облікового запису AWS, якщо у вас їх декілька). Якщо у вас вже є Бакет S3, ви можете відразу вказувати конфігурацію backend в своєму коді Terraform без додаткових дій.

Друге обмеження більш вразливе: в розділі backend в Terraform не можна застосовувати ніякі змінні або посилання. Наступний код не буде працювати.

```
# Це НЕ буде працювати. У конфігурації сховища не можна використовувати змінні.
terraform {backend "s3" {
bucket = var.bucket
region = var.region
dynamodb_table = var.dynamodb_table
key = "example / terraform.tfstate"
encrypt = true
}
}
```

Значить, необхідно вручну копіювати і вставляти ім’я і регіон Бакета S3, а також назву таблиці DynamoDB в кожен ваш модуль Terraform. Ви докладно познайомитеся з модулями Terraform пізніше. Поки досить розуміти, що модулі – це спосіб організації і повторного використання коду Terraform і що справжній код Terraform зазвичай складається з безлічі дрібних модулів. Що ще гірше, вам потрібно бути дуже обережними, щоб не скопіювати значення key. Щоб різні модулі, бува, не перезаписували стан один одного, це значення має бути унікальним для кожного з них! Часте

копіювання і ручне редагування можуть призвести до появи помилок, особливо якщо потрібно розгортати і адмініструвати безліч модулів Terraform в багатьох середовищах.

Станом на травень 2021 року єдиним рішенням є використання часткової конфігурації, в якій можна опустити певні параметри розділу backend і передавати їх замість цього в аргументі командного рядка – backend-config при виклику terraforminit. Наприклад, ви можете винести повторювані параметри сховища, такі як bucket і region, в окремий файл під назвою backend.hcl:

```
# backend.hcl
bucket = "terraform-up-and-running-state"
region = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt = true
```

В коді Terraform залишиться тільки параметр key, оскільки вам все одно потрібно встановлювати йому різні значення в різних модулях:

```
# Часткова конфігурація. Інші параметри (такі як bucket, region) будуть
# Передані команді 'terraform init' у вигляді файлу з використанням
# Аргументів -backend-config terraform {
backend "s3" {
key = "example / terraform.tfstate"
}
}
```

Щоб зібрати воєдино всі фрагменти вашої конфігурації, виконайте команду terraforminit з аргументом -backend-config: $ terraform init -backend-config = backend.hcl

Terraform об’єднає часткову конфігурацію з файлу backend.hcl і вашого коду Terraform, щоб отримати повний набір параметрів для вашого модуля.

Ще один варіант полягає в застосуванні Terragrunt, інструменту з відкритим вихідним кодом, який намагається компенсувати те, чого не вистачає в Terraform. Terragrunt може допомогти уникнути дублювання базових параметрів сховища (ім’я та регіон Бакета, ім’я таблиці DynamoDB) за рахунок визначення їх в єдиному файлі та автоматичного застосування відносного файлового шляху модуля в якості значення key.

Ізоляція файлів стану

Завдяки віддаленим сховищам і блокуванню, спільна робота більше не проблема. Але одна проблема у нас все ж залишається: ізоляція. Коли ви

починаєте використовувати Terraform, може з’явитися спокуса описати всю свою інфраструктуру в одному файлі або єдиному наборі файлів в одній папці. Недолік цього підходу в тому, що в одному файлі зберігається не тільки код, але і стан Terraform і, щоб все зламати, достатньо однієї похибки в будь-якому місці.

Наприклад, при спробі розгортання нової версії свого додатка в середовищі фінального тестування ви можете порушити його роботу в промислових умовах. Або ще гірше, ви можете пошкодити весь файл стану (скажімо, через відсутність блокування або через рідкісну програмну похибку в Terraform), в результаті чого ваша інфраструктура вийде з ладу в усіх середовищах.

Весь сенс підтримки декількох середовищ полягає в тому, що вони ізольовані один від одного, тому, якщо ви керуєте усіма ними з одного набору конфігураційних файлів Terraform, ви порушуєте цю ізоляцію. За аналогією з перегородками, які не дають затопити всі секції судна через протікання в одній з них, ви повинні передбачити “перегородки” для своєї архітектури Terraform (рис. 3.22).

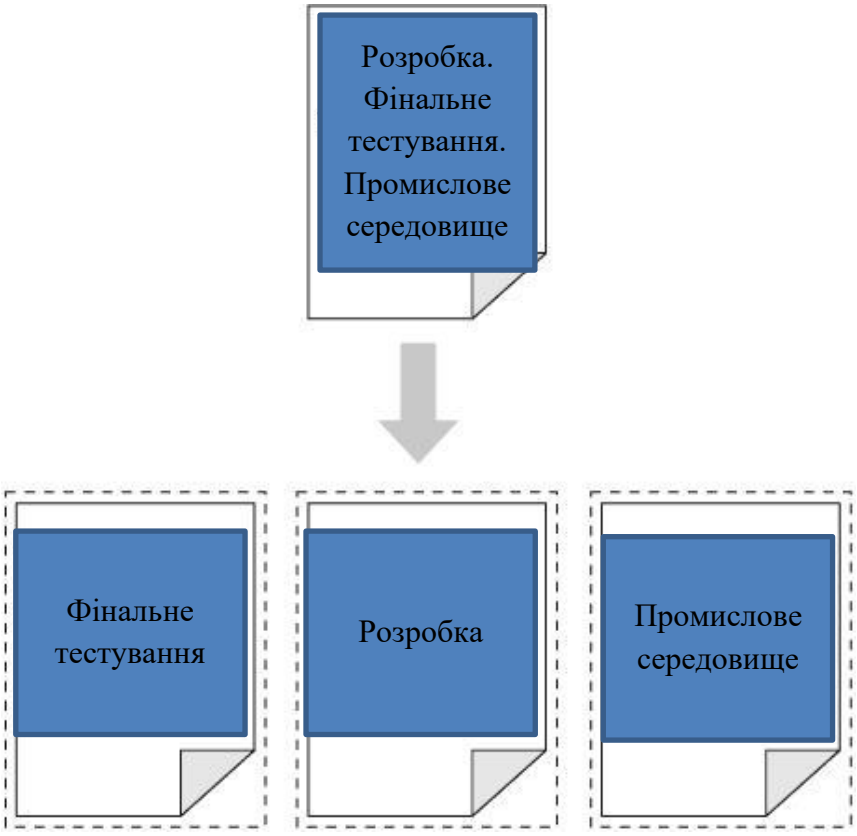


Рис. 3.22. Додавання “перегородок” в архітектуру Terraform

Як проілюстровано на рис. 3.22, ми описуємо всі наші середовища не в одному наборі конфігураційних файлів (вгорі), а в різних наборах (внизу),

тому проблема в одному середовищі повністю ізольована від інших. Файли стану можна ізолювати двома способами:

ізоляція через робочі області. Підходить для швидких ізольованих перевірок з однією і тією ж конфігурацією;

ізоляція за допомогою опису структури файлів. Підходить для промислового використання, коли потрібно сувора ізоляція між середовищами.

Ізоляція через робочі області

Ви можете зберігати свій стан Terraform в декількох окремих іменованих робочих областях. У Terraform спочатку одна робоча область, яка використовується за замовчуванням. Щоб створити нову робочу область або переключитися між областями, потрібно виконати команду terraformworkspace. Поекспериментуємо з певним кодом Terraform, який розгортає сервер EC2:

```
resource "aws_instance" "example" {
ami = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
}
```

Налаштуйте для цього сервера сховище на основі Бакета S3 і таблиці DynamoDB, які ви створили раніше в цьому розділі, але в якості значення для key вкажіть workspaces-example / terraform.tfstate:

```
terraform {
backend "s3" {
Поміняйте це на ім'я свого бакета!
bucket = "terraform-up-and-running-state"
key = "workspaces-example / terraform.tfstate"
region = "us-east-2"
Замініть це ім'ям своєї таблиці DynamoDB!
dynamodb_table = "terraform-up-and-running-locks"
encrypt = true
  }
}
```

Виконайте команди terraforminit і terraformapply, щоб розгорнути цей код:

```
terraform init Initializing the backend ...
Successfully configured the backend "s3"!
Terraform will automatically use
this backend unless the backend configuration changes.
Initializing provider plugins ...
```

(...)

Terraform has been successfully initialized!

terraform apply (...)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Стан цього розгортання зберігається в робочій області замовчуванням. У цьому можна переконатися за допомогою команди terraformworkspaceshow, яка показує, в якій робочої області ви зараз перебуваєте:

terraform workspace show default

Робоча область за замовчуванням зберігає ваш стан саме в тому місці, яке ви вказали в параметрі key. Як видно на рис. 3.23, глянувши на свій Бакет S3, ви побачите файл terraform.tfstate і папку workspaces-example.

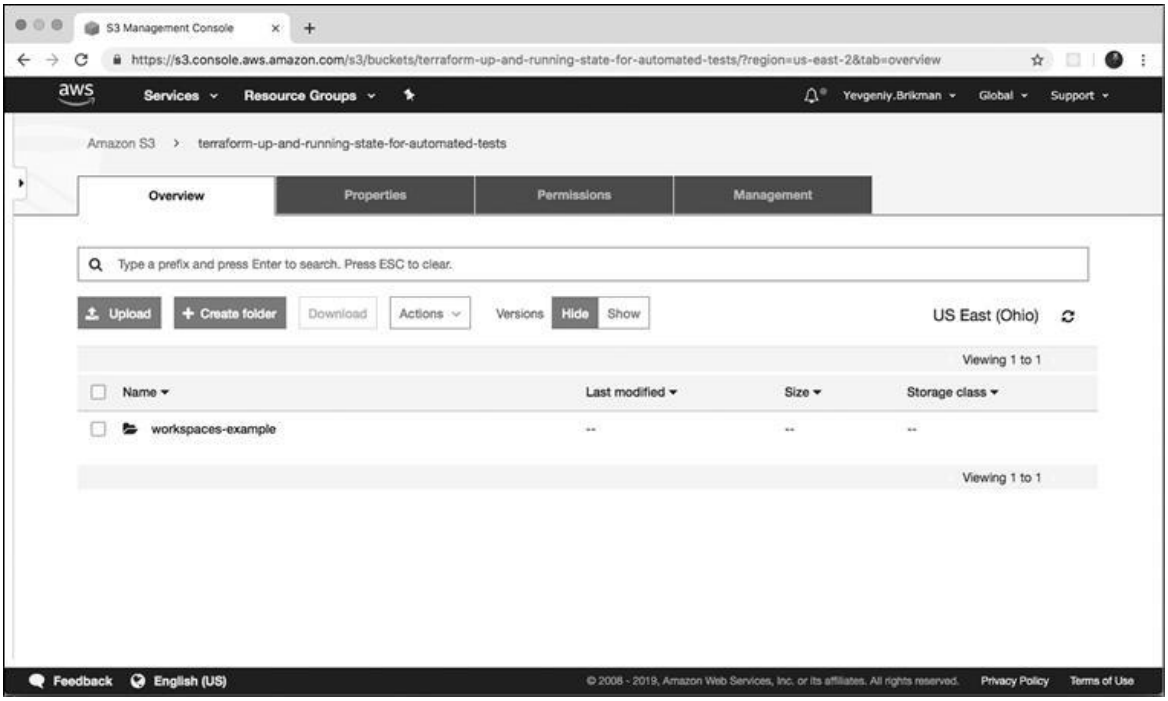


Рис. 3.23. Бакет S3 після збереження стану в робочій області за замовчуванням

Створимо нову робочу область над назвою example1, використовуючи команду terraformworkspacenew:

$ Terraform workspace new example1

Created and switched to workspace "example1"!

You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.

Тепер подивіться, що вийде, якщо ми спробуємо виконати terraformplan:

$ terraform plan

Terraform will perform the following actions:

aws_instance.example will be created + resource "aws_instance" "example" {

+ ami = "ami-0c55b159cbfafe1f0"

+ instance_type = "t2.micro"

(...)

}

Plan: 1 to add, 0 to change, 0 to destroy.

Terraform хоче створити з нуля абсолютно новий сервер EC2! Справа в тому, що файли стану в кожній робочій області ізольовані один від одного і, оскільки ви тепер в робочій області example1, Terraform більше не використовує файл стану з робочою області за замовчуванням. Отже, не бачить сервер EC2, який був там створено.

Спробуйте виконати команду terraformapply, щоб розгорнути цей другий сервер EC2 в новій робочій області:

terraform apply (...)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Повторимо цей приклад ще один раз і створимо нову робочу область

під назвою example2:

$ Terraform workspace new example2

Created and switched to workspace "example2"!

You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.

Знову виконайте terraformapply, щоб розгорнути третій сервер EC2:

terraform apply (...)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Тепер у вас є три робочі області; в цьому можна переконатися з допомогою команди terraformworkspacelist:

$ Terraform workspace list

default

example1

* example2

Ви можете переключитися між ними в будь-який момент, використовуючи команду terraformworkspaceselect:

terraform workspace select example1 Switched to workspace "example1".

Щоб зрозуміти, як це працює всередині, ще раз загляньте в свій бакет. Ви повинні побачити нову папку під назвою env :, як показано на рис. 3.24.

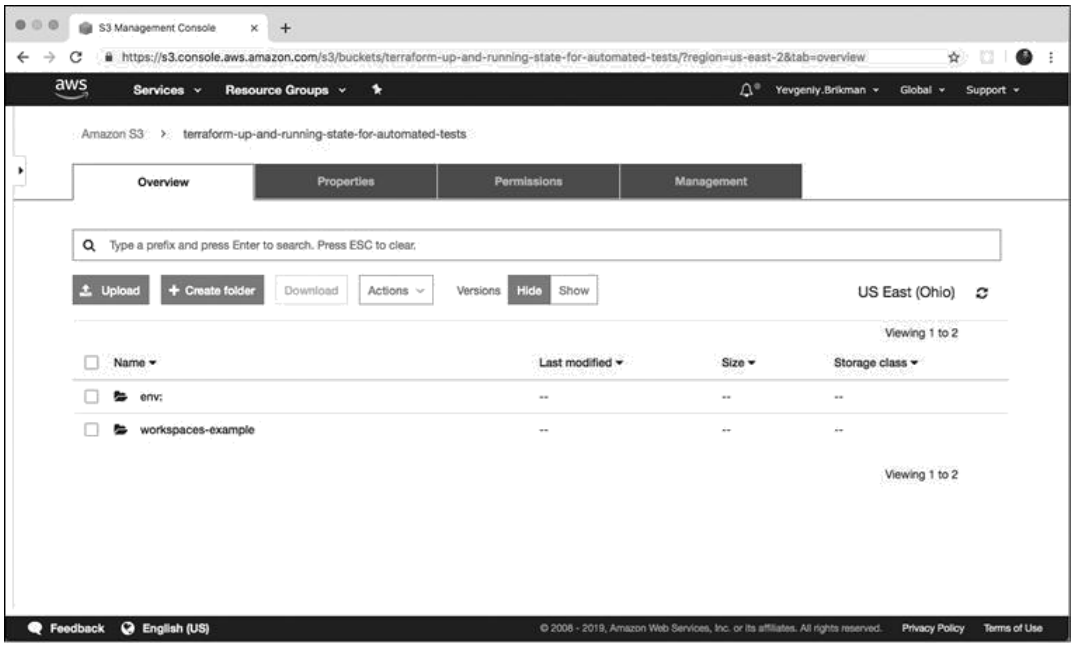


Рис. 3.24. Бакет S3 після того, як ви почали використовувати власні робочі області

Усередині env: ви знайдете по одній папці для кожної з ваших робочих областей (рис. 3.25).

Усередині кожної з цих робочих областей Terraform використовує значення key, яке ви вказали в конфігурації сховища, тому ви повинні знайти файли example1/workspaces-example/terraform.tfstate і example2/workspaces-example/terraform.tfstate. Іншими словами, перемикання на іншу робочу область рівнозначно зміни шляху зберігання вашого файла стану.

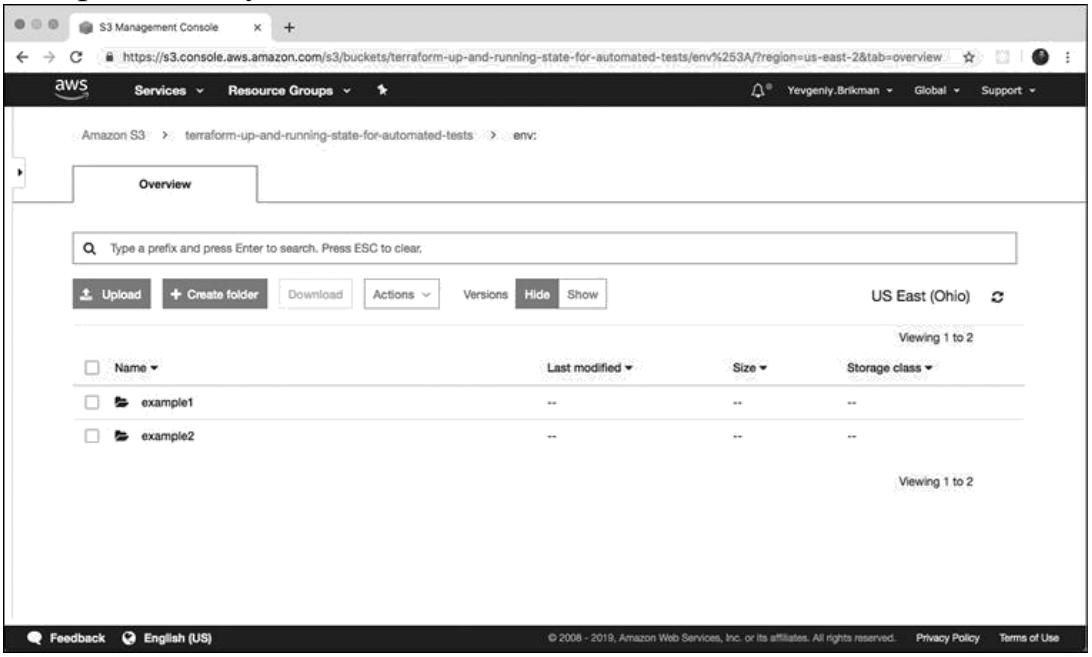


Рис. 3.25. Terraform створює по одній папці для кожної робочої області

Це зручно, коли ви вже розгорнули модуль Terraform і хочете з ним поекспериментувати (наприклад, спробувати змінити структуру коду), але так, щоб ваші експерименти не позначилися на стані вже розгорнутої інфраструктури. Ви можете виконати команду terraformworkspacenew і розгорнути нову копію тієї ж інфраструктури, але з окремим файлом стану.

Ви можете навіть зробити так, щоб цей модуль міняв свою поведінку в залежності від того, в якій робочої області ви перебуваєте. Для цього він може зчитувати ім'я робочої області за допомогою виразу terraform.workspace. Наприклад, в робочій області за замовчуванням можна вказати тип сервера EC2 t2.medium, а у всіх інших областях - t2.micro (щоб, скажімо, зробити свої експерименти більш економними):

```
resource "aws_instance" "example" {
ami = "ami-0c55b159cbfafe1f0"
instance_type = terraform.workspace == "default"? "t2.medium":
"t2.micro"
}
```

Цей код використовує *тернарний синтаксис*, щоб привласнити параметру instance_type значення t2.medium або t2.micro в залежності від значення terraform.workspace.

Робочі області Terraform відмінно підходять для швидкого розгортання і видалення різних версій вашого коду, але у них є кілька недоліків.

Файли стану всіх ваших робочих областей знаходяться в одному і тому ж сховищі (наприклад, в одному бакеті S3). Це означає, що всі вони використовують аналогічні механізми аутентифікації і управління доступом, що є однією з основних причин, чому вони не підходять для ізоляції різних оточень, таких як середовища для фінального тестування і промислового застосування.

Робочі області не видно в коді або терміналі без використання команд terraformworkspace. При читанні коду неможливо сказати, в скількох робочих областях розгорнуто модуль: в одній або десяти, так як виглядають вони ідентично. Це ускладнює обслуговування, оскільки у вас немає повноцінного уявлення про вашу інфраструктуру.

З двох попередніх пунктів випливає той факт, що робочі області можуть бути досить сильно схильні до помилок. Через брак прозорості можна легко забути, в якій робочій області ви перебуваєте, внести зміни не в тому місці (наприклад, випадково виконати команду terraformdestroy в робочій області production замість staging). Оскільки вам доводиться

використовувати один і той же механізм аутентифікації для всіх робочих областей, у вас немає іншого рівня захисту від подібних помилок.

Щоб як слід ізолювати різні оточення, замість робочих областей краще використовувати структуру файлів і каталогів, про яку піде мова в наступному матеріалі. Але перш, ніж рухатися далі, не забудьте видалити три сервери EC2, які ви тільки що розгорнули. Для цього виконайте terraformworkspaceselect <ім'я> і terraformdestroy в кожній з трьох робочих областей.

Ізоляція за допомогою опису структури файлів

Щоб досягти повної ізоляції між оточеннями, потрібно зробити наступне.

Помістити конфігураційні файли Terraform для кожного середовища в окрему папку. Наприклад, вся конфігурація для середовища фінального тестування може перебувати в папці stage, а в папці prod може зберігатися конфігурація промислового середовища.

Передбачити для кожного середовища різні сховища з різними механізмами аутентифікації і управління доступом (припустимо, у кожного оточення може бути окремий обліковий запис AWS зі своїм бакета S3 як сховище).

Завдяки застосуванню окремих папок вам буде набагато легше зрозуміти, в якому середовищі відбувається розгортання, а використання окремих файлів стану з окремими механізмами аутентифікації значно зменшує ймовірність того, що випадкова помилка в одному середовищі матиме будь-який вплив на інші.

Концепцію ізоляції краще опустити на рівень нижче, аж до компонентів – наборів пов’язаних між собою ресурсів, які зазвичай розгортаються спільно. Наприклад, після настройки базової мережевої топології для своєї інфраструктури (в термінології AWS це віртуальна приватна хмара (VPC) і всі пов’язані з нею підмережі, правила маршрутизації, VPN і мережеві списки доступу) ви будете міняти її не частіше одного разу на місяць. Але нові версії серверів можуть розгортатися по кілька разів на день. Керуючи інфраструктурою VPC і вебсервера в одній і тій же конфігурації Terraform, ви постійно ризикуєте порушити роботу всієї своєї мережевої топології (скажімо, через просту помилку в коді або випадкового виконання не тієї команди) без причини.

У зв’язку з цим рекомендовано використовувати окремі папки Terraform (і, отже, окремі файли стану) для кожного оточення (тестового, промислового і т. д.) і кожного компонента (VPC, сервісів, баз даних). Щоб

подивитися, як це виглядає на практиці, розберемо рекомендовану структуру файлів і каталогів для проектів Terraform.

На рис. 3.26 показана типова структура файлів у проектах.

На самому верхньому рівні знаходяться окремі папки для кожного “оточення”. Набір оточень залежить від проекту, але зазвичай він включає в себе наступне:

stage – середовище для передпромислового навантаження (тестування);

prod – середовище для промислового навантаження (додатки, які взаємодіють з користувачами).

mgmt – середовище для інструментарію DevOps (наприклад Jenkins);

global – папка з ресурсами, які використовуються у всіх середовищах (Скажімо, S3, IAM).

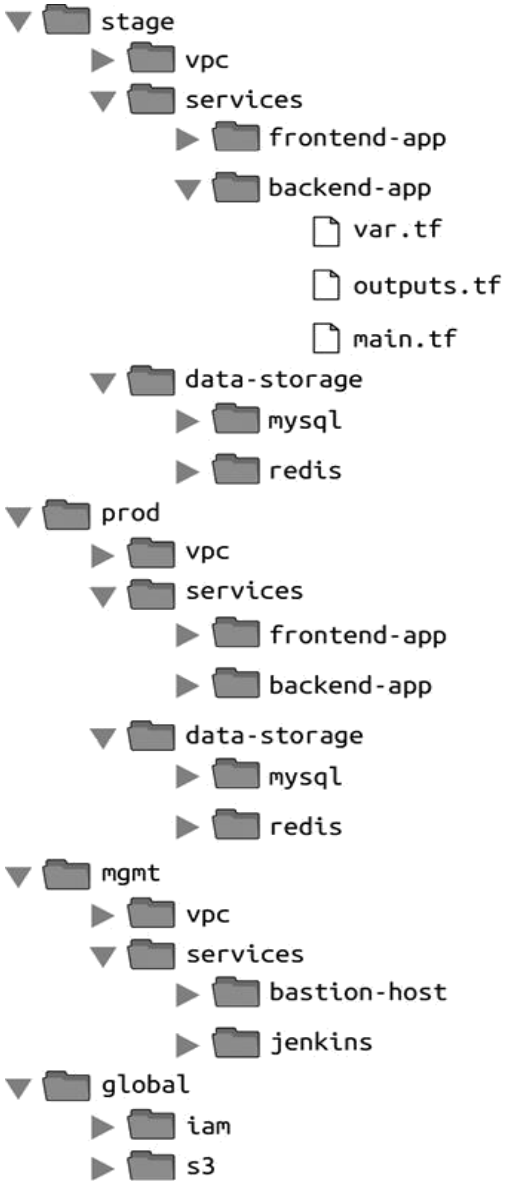


Рис. 3.26. Типова структура файлів для проєкту Terraform

Усередині кожного оточення є окремі папки для кожного компонента. Компоненти залежать від проекту, але зазвичай в їх число входять наступні:

vpc – мережева топологія для цього середовища;

services – додатки або мікросервіси, які запускаються в цьому середовищі, наприклад, Ruby on Rails на стороні клієнта або Scala на стороні сервера. Ви можете навіть ізолювати всі додатки, помістивши їх в окремі папки;

data-storage – сховища даних для цього середовища, такі як MySQL або Redis. Сховища можна ізолювати один від одного, розмістивши їх в окремих папках.

Усередині кожного компонента знаходяться конфігураційні файли Terraform, які названі за наступним принципом:

variables.tf – вхідні змінні;

outputs.tf – вихідні змінні;

main.tf – ресурси.

При запуску Terraform просто шукає в цій папці файли з розширенням .tf, тому ви можете називати свої файли як завгодно. Але, незважаючи на це, вашим колегам не все одно, які назви файлів ви використовуєте. Якщо задати послідовні і передбачувані назви, це спростить перегляд коду. Ви завжди будете знати, де шукати змінні, виведення або ресурси. Якщо якийсь файл (особливо main.tf) стає занадто великим, можете вільно винести з нього частину функціональності (і назвати її, наприклад, iam.tf, s3.tf або database.tf), але це також може бути ознакою того, що вам слід розбити свій код на більш дрібні модулі.

Як уникнути дублювання коду

Структура файлів і каталогів, описана в цьому розділі, має багато повторювальних елементів. Наприклад, одні і ті ж додатки frontend-app і backend-app знаходяться відразу в двох папках: stage і prod. Але не хвилюйтеся, вам не доведеться копіювати і вставляти весь цей код! У наступному матеріалі ви побачите, як уникати дублювання і дотримуватися принципу DRY за допомогою модулів Terraform.

Візьмемо код кластера з вебсервером, який ви написали раніше, об’єднаємо його з кодом для Amazon S3 і DynamoDB та організуємо це все у вигляді структури каталогів, представленої на рис. 3.27.

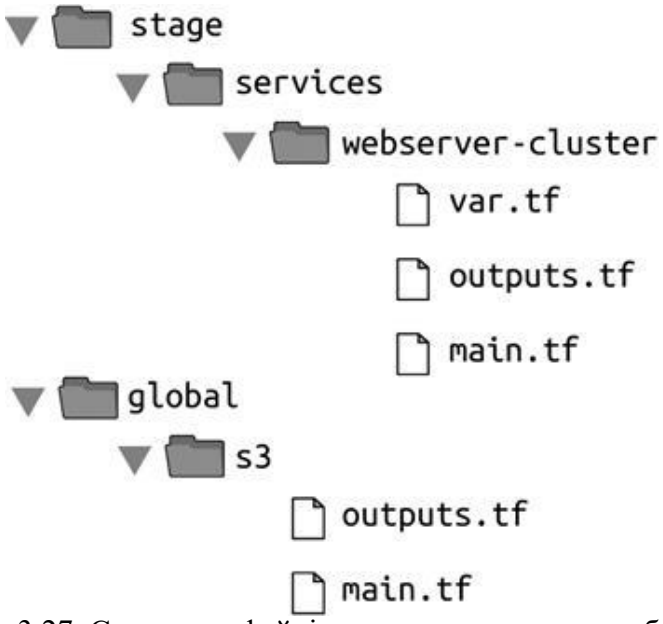


Рис. 3.27. Структура файлів для коду кластера з вебсервером

Бакет S3, створений вами в цьому розділі, необхідно перемістити в папку global/s3. Помістіть вихідні змінні (s3_bucket_arn і dynamodb_table_name) в файл outputs.tf. При переміщенні файлів в нове місце переконайтеся, що ви не забули скопіювати (приховану) папку .terraform, інакше доведеться заново все форматувати.

Кластер вебсерверів, який ви створили в розділі 2, слід помістити в папку stage / services / webserver-cluster (свого роду тестова версія цього кластера; промислову ми додамо в наступному розділі). Не забудьте скопіювати папку .terraform і помістити вхідні і вихідні змінні файли variables.tf і відповідно outputs.tf.

Вам також необхідно оновити кластер вебсерверів, щоб він використовував S3 в якості сховища. Ви можете просто скопіювати і вставити розділ backend з файлу global /s3/main.tf без особливих змін, але не забудьте привласнити параметру key шлях до папки, в якій знаходиться код вебсервера: stage/services/webserver-cluster/terraform.tfstate. Таким чином, код Terraform в системі управління версіями і структура ваших файлів стану будуть повністю співпадати і вам буде очевидно те, як вони між собою пов’язані. Модуль s3 використовує цей же принцип для установки параметра key.

Така структура файлів і каталогів спрощує перегляд коду і допомагає зрозуміти, які саме компоненти розгорнуті в тому чи іншому середовищі. Крім того, вона забезпечує хорошу ступінь ізоляції між оточеннями і між

компонентами всередині одного оточення. Завдяки цьому, якщо щось піде не так, це торкнеться лише невеликої частини вашої інфраструктури.

Звичайно, ця властивість в якомусь сенсі є і недоліком: поділ компонентів на окремі папки не дає випадкової помилки порушити роботу всієї інфраструктури, але при цьому позбавляє вас можливості розгортати всю інфраструктуру однією командою. Якби всі компоненти для одного оточення були описані в одному файлі конфігурації Terraform, ви б могли запустити все це оточення за допомогою єдиного виклику terraformapply. Але якщо всі компоненти знаходяться в окремих папках, потрібно виконати terraformapply для кожної з них. Варто відзначити, що в разі застосування Terragrunt ви можете автоматизувати цей процес за допомогою команди apply-all39.

В цієї структури файлів є ще одна проблема: вона ускладнює використання залежностей ресурсів. Додаток має прямий доступ до атрибутів бази даних (тобто може послатися на адресу бази даних через aws_db_instance.foo.address), якщо код цих двох компонентів знаходиться в одних і тих же конфігураційних файлах Terraform. Але якщо код програми та бази даних розміщений в різних папках, як я радив, ця можливість втрачається. На щастя, Terraform пропонує вирішення цієї проблеми: джерело даних terraform_remote_state.

Джерело даних terraform_remote_state

Раніше ви використовували джерела даних для вилучення з AWS інформації, доступної тільки для читання. Наприклад, джерело aws_subnet_ids повертав список підмереж вашої хмари VPC. Але існує інше джерело, terraform_remote_state, який особливо корисний при роботі зі станом. З його допомогою можна витягти файл стану, який є частиною іншої конфігурації Terraform, і зробити це суто для читання.

Розглянемо приклад. Уявіть, що вашому кластеру вебсерверів необхідно взаємодіяти з базою даних MySQL. Обслуговування масштабуємої, безпечної, стійкої і високодоступної БД вимагає багато зусиль. Ви можете дозволити Amazon подбати про це за допомогою сервісу RDS (Relational Database Service), як показано на рис. 3.28. RDS може використовувати різні бази даних, включаючи MySQL, PostgreSQL, SQL Server і Oracle.

Базу даних MySQL краще не оголошувати в тому ж наборі конфігураційних файлів, що і кластер вебсерверів, тому що останній оновлюється значно частіше і вам навряд чи захочеться ризикувати при кожному такому оновленні. Перше, що ви повинні зробити – це створити

нову папку stage/data-stores/mysql і помістити в неї три основних файли Terraform (main.tf, variables.tf, outputs.tf), як показано на рис. 3.29.

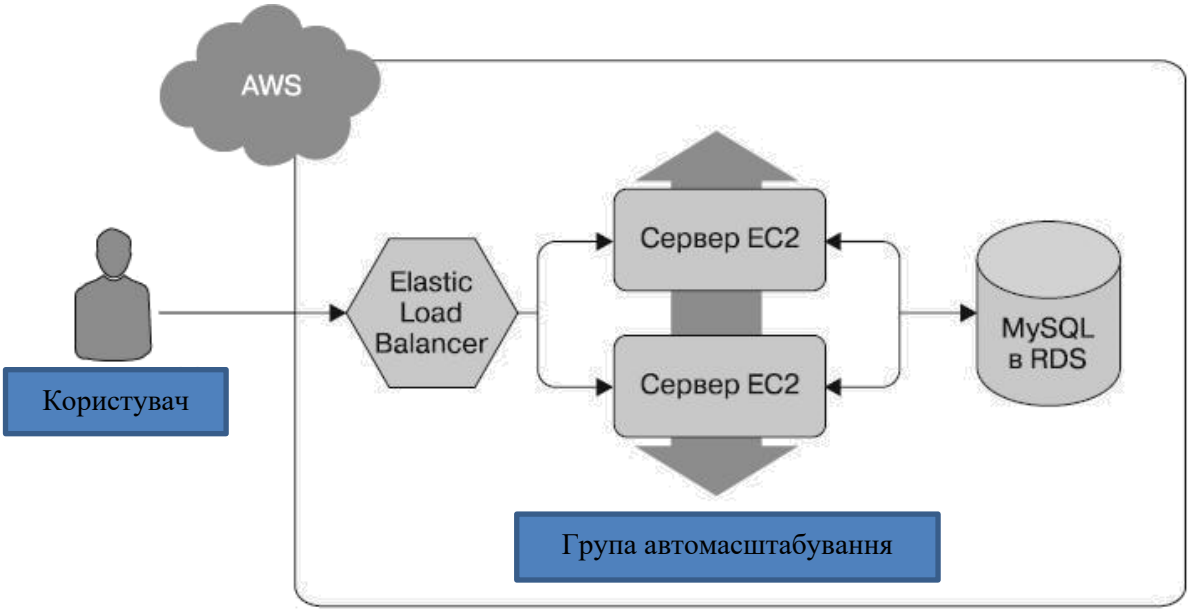


Рис. 3.28. Кластер веб-серверів взаємодіє з базою даних MySQL, розгорнутої поверх Amazon RDS

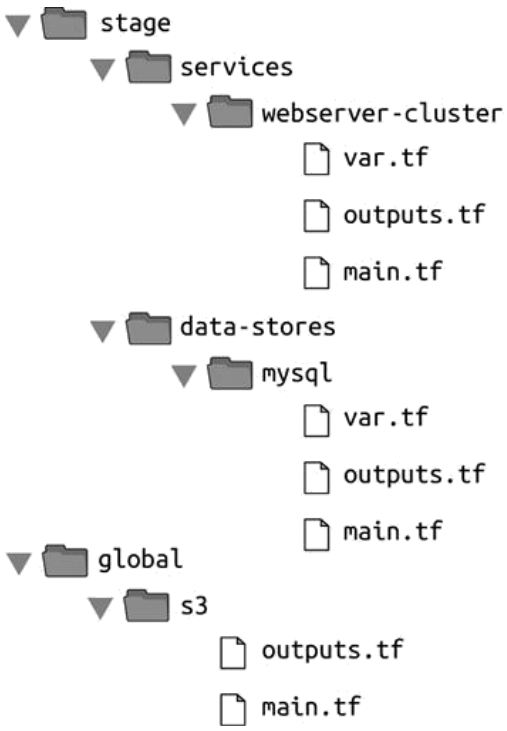


Рис. 3.29. Код бази даних в папці stage/data-stores

Слідом за цим створіть ресурс бази даних у файлі stage/data-stores/mysql / main.tf:

```
provider "aws" {
region = "us-east-2"
}
```

```
resource "aws_db_instance" "example" {
identifier_prefix = "terraform-up-and-running"
engine = "mysql"
allocated_storage = 10
instance_class = "db.t2.micro"
name = "example_database"
username = "admin"
Як нам задати пароль?
password = "???"
}
```

Відразу під стандартним ресурсом provider у верхній частині файла знаходиться новий: aws_db_instance. Він створює базу даних в RDS.

Параметри в цьому коді налаштовують RDS для запуску MySQL зі сховищем розміром 10 Гбайт на сервері db.t2.micro, який має один віртуальний процесор, 1 Гбайт пам'яті і входить в безкоштовний тариф AWS.

Зверніть увагу, що одним з параметрів, які ви повинні передати ресурсу aws_db_instance, є головний пароль до бази даних. Оскільки він конфіденційний, його не можна прописувати прямо в коді у вигляді звичайного тексту! Замість цього можна скористатися одним з двох способів передачі конфіденційних даних в ресурси Terraform.

Перший спосіб для роботи з конфіденційними даними полягає в застосуванні джерела Terraform, який зчитує їх з секретного сховища. Наприклад, ви можете розміщувати таку інформацію, як паролі до бази даних, в керованому сервісі AWS Secrets Manager, призначеному спеціально для зберігання чутливих даних. Ви можете скористатися його графічним інтерфейсом, щоб зберегти свій пароль, і потім прочитати його в своєму коді Terraform за допомогою джерела даних aws_secretsmanager_secret_version:

```
provider "aws" {
region = "us-east-2"
}
resource "aws_db_instance" "example" {identifier_prefix = "terraform-up-and-running"
engine = "mysql"
allocated_storage = 10
instance_class = "db.t2.micro"
name = "example_database"
username = "admin"
password =
```

data.aws_secretsmanager_secret_version.db_password.secret_string

}

data "aws_secretsmanager_secret_version" "db_password" {secret_id = "mysql-master-password-stage"

}

Ось кілька підтримуваних комбінацій секретних сховищ і джерел даних, які можуть вас зацікавити:

AWS Secrets Manager і джерело даних aws_secretsmanager_secret_version (попередній лістинг);

AWS Systems Manager Parameter Store і джерело даних aws_ssm_parameter;

AWS Key Management Service (AWS KMS) і джерело даних aws_kms_secrets;

Google Cloud KMS і джерело даних google_kms_secret;

Azure Key Vault і джерело даних azurerm_key_vault_secret;

HashiCorp Vault і джерело даних vault_generic_secret.

Другий спосіб роботи з конфіденційними даними – повний винос управління ними за межі Terraform (наприклад, ви можете делегувати це таким диспетчерам паролів, як 1Password, LastPass або OS X Keychain) і передача їх в систему у вигляді змінних середовища. Для цього потрібно оголосити змінну під назвою db_password в файлі stage / data-stores / mysql / variables.tf:

variable "db_password" {

description = "The password for the database"

type = string

}

Зверніть увагу на те, що у неї немає параметра default. Це зроблено навмисно. Ви не повинні зберігати свій пароль до бази даних або будь-яку чутливу інформацію у відкритому вигляді. Замість цього значення слід брати з змінної середовища.

Нагадуємо, що кожній вхідній змінній, яка визначена в конфігурації Terraform (як foo), можна надати значення, яке береться зі змінної середовища (на кшталт TF_VAR_foo). Для вхідної змінної db_password потрібно встановити змінну середовища TF_VAR_db_password. Ось як це робиться в системах Linux / Unix / OS X:

export TF_VAR_db_password = "(YOUR_DB_PASSWORD)"

terraform apply (...)

Варто відзначити, що пробіл перед командою export зазначений не випадково.

Він потрібен, щоб ваші конфіденційні дані не були збережені на диск в історії bash40. Але є кращий спосіб запобігти випадковому запису конфіденційних даних на диск у відкритому вигляді: зберігати їх в секретному сховищі, сумісному з командним рядком, такому як pass (https://www.passwordstore.org/) і безпечно зчитувати їх звідти в змінні середовища за допомогою дочірньої командної оболонки:

export TF_VAR_db_password = $ (pass database-password)

terraform apply (...)

Конфіденційні дані завжди зберігаються в стані Terraform

Зчитування конфіденційних даних з секретного сховища або змінних середовища – хороший спосіб запобігти їх зберігання у відкритому вигляді всередині вашого коду. Але пам’ятайте: незалежно від того, як ви їх читаєте, якщо вони передаються як аргумент в ресурс, такий як aws_db_instance, вони автоматично зберігаються в стані Terraform у вигляді звичайного тексту.

Це відоме слабке місце Terraform, у якого немає ефективних рішень. Тому будьте максимально пильні з тим, як ви зберігаєте файли стану (наприклад, завжди включайте шифрування) і хто має до них доступ (скажімо, обмежуйте доступ до свого бакета S3 за допомогою привілеїв IAM)!

Слідом за конфігуруванням пароля потрібно зробити так, щоб модуль зберігав свій стан в бакеті S3, який ви створили раніше в файлі stage/data-stores/mysql/terraform.tfstate:

terraform {

backend "s3" {

Поміняйте це на ім'я свого бакета!

bucket = "terraform-up-and-running-state"

key = "stage / data-stores / mysql / terraform.tfstate"

region = "us-east-2"

Замініть це ім'ям своєї таблиці DynamoDB!

dynamodb_table = "terraform-up-and-running-locks"

encrypt = true

}

}

Виконайте команди terraforminit і terraformapply, щоб створити базу даних. Потрібно враховувати, що на ініціалізацію навіть невеликої бази даних в Amazon RDS може піти близько десяти хвилин, тому будьте терплячі.

Отже, ви створили базу даних. Але як передати її адресу і порт вашому кластеру вебсерверів? Для початку потрібно додати дві вихідні змінні в файл stage/data-stores/mysql/outputs.tf:

output "address" {

value = aws_db_instance.example.address description = "Connect to the database at this endpoint"

}

output "port" {

value = aws_db_instance.example.port description = "The port the database is listening on"

}

Виконайте terraformapply ще раз, і в терміналі повинні з’явитися ваші вихідні змінні:

terraform apply (...)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed. Outputs:

address = tf-2016111123.cowu6mts6srx.us-east-2.rds.amazonaws.com port = 3306

Тепер вони зберігаються і в стані Terraform для бази даних, яке знаходиться в вашому бакеті S3 в файлі stage/data-stores/mysql/terraform.tfstate. Щоб код вашого кластера вебсерверів прочитав вміст цього файла стану, додайте в файл stage/services/webserver-cluster main.tf джерело даних terraform_remote_state:

data "terraform_remote_state" "db" {

backend = "s3"

config = {

bucket = "(YOUR_BUCKET_NAME)"

key = "stage / data-stores / mysql / terraform.tfstate"

region = "us-east-2"

}

}

Завдяки цьому джерелу даних, код кластера вебсерверів зчитує файл стану з того ж бакета S3 і папки, де свої стани зберігає база даних (рис. 3.30).

Важливо розуміти, що, як і всі джерела даних в Terraform, terraform_remote_state повертає тільки доступну для читання інформацію. У вас немає ніякої можливості поміняти цей стан в коді кластера вебсерверів, тому те, що ви витягаєте стан бази даних, не ставить під жодний ризик.

Важливо розуміти, що, як і всі джерела даних в Terraform, terraform_remote_state повертає тільки доступну для читання інформацію. У

вас немає ніякої можливості поміняти цей стан в коді кластера вебсерверів, тому те, що ви витягаєте стан бази даних, не ставить під жодний ризик.

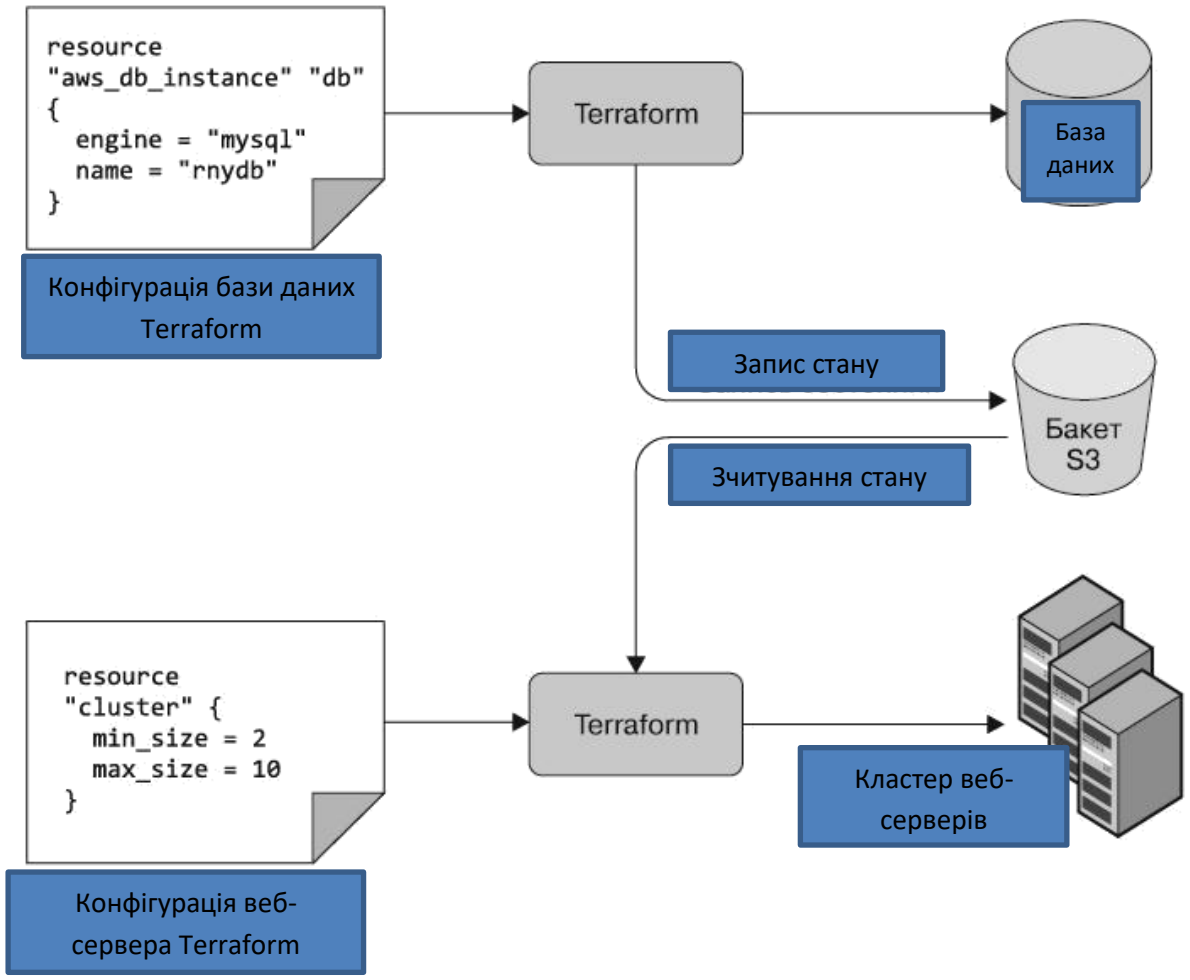


Рис. 3.30. База даних записує свій стан в бакет S3 (вгорі), а кластер вебсерверів зчитує його з того ж бакета (внизу)

Важливо розуміти, що, як і всі джерела даних в Terraform, terraform_remote_state повертає тільки доступну для читання інформацію. У вас немає ніякої можливості поміняти цей стан в коді кластера вебсерверів, тому те, що ви витягаєте стан бази даних, не ставить під жодний ризик.

Всі вихідні змінні БД зберігаються в файлі стану, і ви можете зчитувати їх з джерела даних terraform_remote_state, використовуючи посилання на атрибут такого вигляду:

data.terraform_remote_state. <NAME> .outputs. <ATTRIBUTE>

Наприклад, ось як можна оновити призначені для користувача дані вебсерверів кластера, щоб вони відкликали з джерела terraform_remote_state адресу і порт бази даних і повертали їх у вигляді HTTP-відповіді:

user_data = << EOF

#! / Bin / bash

echo "Hello, World" >> index.html

echo "$ {data.terraform_remote_state.db.outputs.address}" >> index.html

echo "$ {data.terraform_remote_state.db.outputs.port}" >> index.html nohup busybox httpd -f -p $ {var.server_port} & EOF

Чим довше стає скрипт в параметрі user_data, тим більш неохайним виглядає ваш код. Вбудовування однієї мови програмування (bash) в інший (Terraform) ускладнює підтримку обох, тому давайте на секунду зупинимося і винесемо bash-скрипт в окремий файл. Для цього можна використовувати вбудовану функцію file і джерело даних template_file. Розглянемо їх окремо.

Terraform включає в себе ряд вбудованих функцій, які можна виконувати за допомогою виразу такого виду:

function_name (...)

Візьмемо для прикладу функцію format:

format (<FMT>, <ARGS>, ...)

Ця функція форматує аргументи в ARGS відповідно до синтаксиса sprintf, заданим в рядку FMT41. Відмінним способом поекспериментувати з вбудованими функціями є використання команди terraformconsole. Вона створює інтерактивну консоль, в якій ви можете спробувати синтаксис Terraform, запросити стан вашої інфраструктури і відразу ж отримати результати:

$ Terraform console

format ("%. 3f", 3.14159265359)

3.142

Варто відзначити, що консоль Terraform призначена тільки для читання, тому не потрібно хвилюватися про випадковій зміні інфраструктури або стану.

Існує цілий ряд інших вбудованих функцій для роботи з рядками, числами, списками і асоціативними масивами. До їх числа відноситься функція file:

file (<PATH>)

Ця функція читає файл з шляхом PATH і повертає його вміст у вигляді рядка. Наприклад, ви могли б зберегти скрипт призначених для користувача даних в файл stage/services/webserver-cluster/user-data.sh і завантажувати його в такий спосіб:

file ("user-data.sh")

Підступ в тому, що скрипту користувальницьких даних для кластера вебсерверів необхідна певна динамічна інформація з Terraform, включаючи порт сервера, а також адреса і порт бази даних. Коли скрипт був вбудований в код Terraform, він отримував ці значення за допомогою посилань і

інтерполяції. З функцією file це не спрацює. Однак замість цього ви можете скористатися джерелом даних template_file.

Джерело даних template_file приймає два аргументи: template (рядок, який потрібно обробити) і vars (асоціативний масив зі змінними, які мають бути доступні під час обробки). Він містить одну вихідну змінну, rendered, яка є результатом обробки template. Щоб побачити це в роботі, додайте наступний код джерела template_file в файл stage/services/webserver-cluster/main.tf:

```
data "template_file" "user_data" {
template = file ( "user-data.sh")
vars = {
server_port = var.server_port
db_address = data.terraform_remote_state.db.outputs.address db_port = data.terraform_remote_state.db.outputs.port
 }
}
```

Як бачите, цей код привласнює параметрам template і vars вміст скрипта user-data.sh і, відповідно, три змінні, які потрібні цьому скрипту: порт сервера, адреса і порт бази даних. Для використання цих змінних потрібно відповідним чином оновити скрипт stage/services/webserver-cluster/user-data.sh:

```
#! / Bin / bash
cat> index.html << EOF
<H1> Hello, World </ h1>
<P> DB address: $ {db_address} </ p>
<P> DB port: $ {db_port} </ p>
EOF
nohup busybox httpd -f -p $ {server_port} &
```

Зверніть увагу на кілька змін в цьому bash-скрипт в порівнянні з оригіналом.

Він шукає змінні за допомогою стандартного синтаксису інтерполяції Terraform. Єдиними змінними в даному випадку є ті, що знаходяться в асоціативному масиві vars джерела template_file. Варто зазначити, що для доступу до них не потрібен ніякий префікс: наприклад, замість var.server_port слід писати server_port.

Тепер в скрипті можна помітити синтаксис HTML (скажімо, <h1>), який робить виведення зручнішим для читання в браузері.

Зауваження по зовнішніх файлах

Одним з переваг виносу скрипта користувальницьких даних в окремий файл є можливість написання для нього модульних тестів. Код тесту може навіть заповнити інтерпольовані значення за допомогою змінних середовища, оскільки для пошуку останніх bash використовує той же синтаксис, який в Terraform застосовується для інтерполяції. Ви можете написати автоматичний тест для user-data.sh приблизно такого вигляду:

export db_address = 12.34.56.78
export db_port = 5555
export server_port = 8888
./user-data.sh
output = $ (curl "http: // localhost: $ server_port")
if [[$ output == * "Hello, World" *]]; then
echo "Success! Got expected text from server."
else
echo "Error. Did not get back expected text 'Hello, World'."
fi

Заключним кроком буде оновлення параметра user_data в ресурсі aws_launch_configuration. Дайте йому оброблений вихідний атрибут джерела даних template_file:

resource "aws_launch_configuration" "example" {
image_id = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
security_groups = [aws_security_group.instance.id] user_data = data.template_file.user_data.rendered

Необхідна для використання групи автомасштабування в конфігурації запуску.

https://www.terraform.io/docs/providers/aws/r/launch_configuration.html lifecycle {
create_before_destroy = true
}
}

Виглядає набагато акуратніше, ніж вбудовування bash-скриптів!

Якщо розгорнути цей кластер за допомогою команди terraformapply, почекати, поки сервери не зареєструються в ALB, і відкрити URL-адресу ALB в браузері, можна побачити щось схоже на рис. 3.31.

Ура! Ваш кластер веб-серверів тепер може програмно звертатися за адресою і портом бази даних через Terraform. Якщо ви використовуєте цей вебфреймворк (на кшталт Ruby on Rails), можете задати адресу і порт у

вигляді змінних середовища або записати їх в конфігураційний файл, щоб їх могла використовувати ваша бібліотека для роботи з БД (як ActiveRecord).

Рис. 3.31. Кластер вебсерверів може програмно звертатися за адресою і портом бази даних

Резюме

Причина, по якій вам слід приділяти стільки уваги ізоляції, блокування і станом, полягає в тому, що IaC відрізняється від звичайного програмування. При написанні коду для типового додатку більшість помилок виявляються несуттєвими і псують тільки невелику його частину. Але коли ви пишете код для управління інфраструктурою, програмні помилки мають більш серйозні наслідки, оскільки можуть торкнутися всіх ваших програм разом з усіма джерелами даних, топологією мережі і практично всім іншим. Тому при роботі над IaC радимо використовувати більше “захисних механізмів”, ніж при написанні звичайного кода.

Застосування рекомендованої структури файлів і каталогів часто призводить до дублювання коду. Якщо ви хочете запускати кластер веб-серверів як в тестовій, так і в промисловому середовищі, як уникнути копіювання і вставки великої кількості фрагментів між stage/services/webserver-cluster і prod/services/webserver-cluster? Відповідь: використовувати модулі Terraform.

Дізнайтеся більше про гарантії, які дає S3, за адресою amzn.to/31ihjAg.

Ознайомтеся з тарифами для S3 за адресою amzn.to/2yTtnw1.

Детальніше про імена бакета S3 можна почитати за адресою bit.ly/2b1s7eh.

Ознайомтеся з тарифами для DynamoDB за адресою amzn.to/2OJiyHp.

За адресою bit.ly/2lTsewM представлений яскравий приклад того, що може трапитися, якщо не ізолювати стан Terraform.

Детальніше про це читайте в документації Terragrunt за адресою bit.ly/2M48S8e.

Більшість командних оболонок в Linux / Unix / OS X зберігають кожну введену вами команду в файл історії (наприклад, ~ / .bash_history). Але якщо почати команду з пробілу, майже всі оболонки не стануть її туди записувати. Майте на увазі, що ця можливість може бути відключена в вашій командній оболонці. Щоб її включити, доведеться привласнити змінній середовища HISTCONTROL значення ignoreboth.

На сторінці golang.org/pkg/fmt/ можна знайти документацію для синтаксису sprintf.

На сторінці bit.ly/2GNCxOM представлений повний список вбудованих функцій.

Детальніше про захисні механізми в ПО можна почитати за адресою bit.ly/2YJuqJb.

3.2.11. Повторне використання інфраструктури за допомогою модулів Terraform

На даному етапі у вас розгорнута інфраструктура, показана на рис. 3.32.

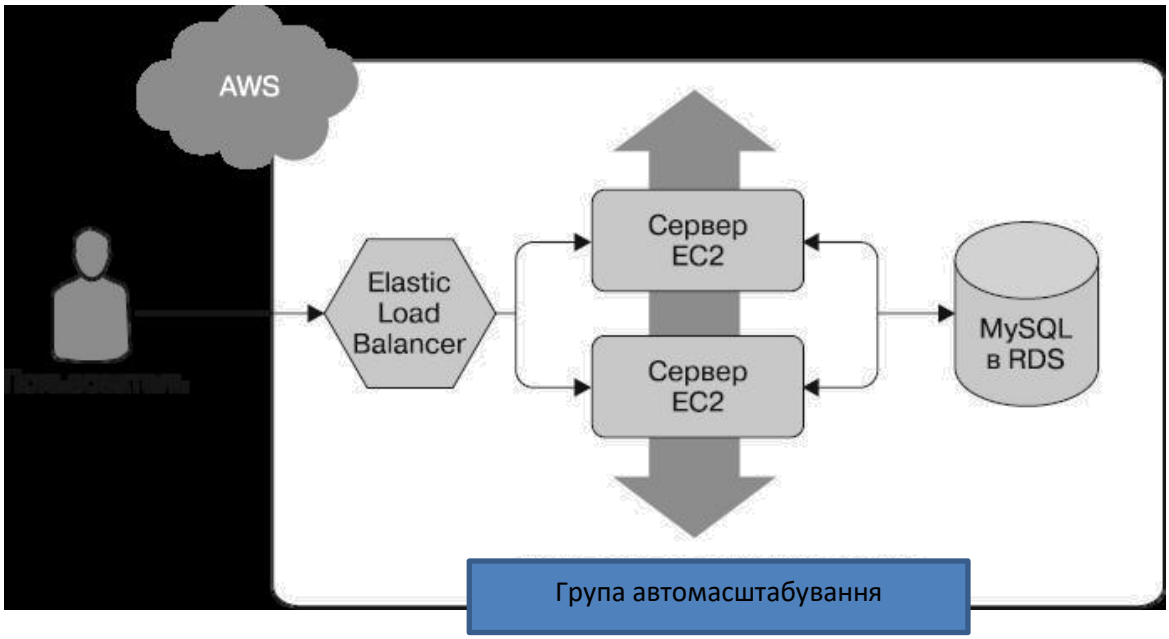


Рис. 3.32. Балансувальник навантаження, кластер вебсерверів і база даних

Це добре працює в якості першого середовища, але їх зазвичай потрібно як мінімум два: одне для фінального тестування всередині

команди, а інше, промислове – для обслуговування реальних користувачів (рис. 3.33). В ідеалі обидва середовища повинні бути майже ідентичними, хоча, щоб заощадити, для середовища фінального тестування важливо ініціювати трохи менше серверів (або сервери меншого розміру).

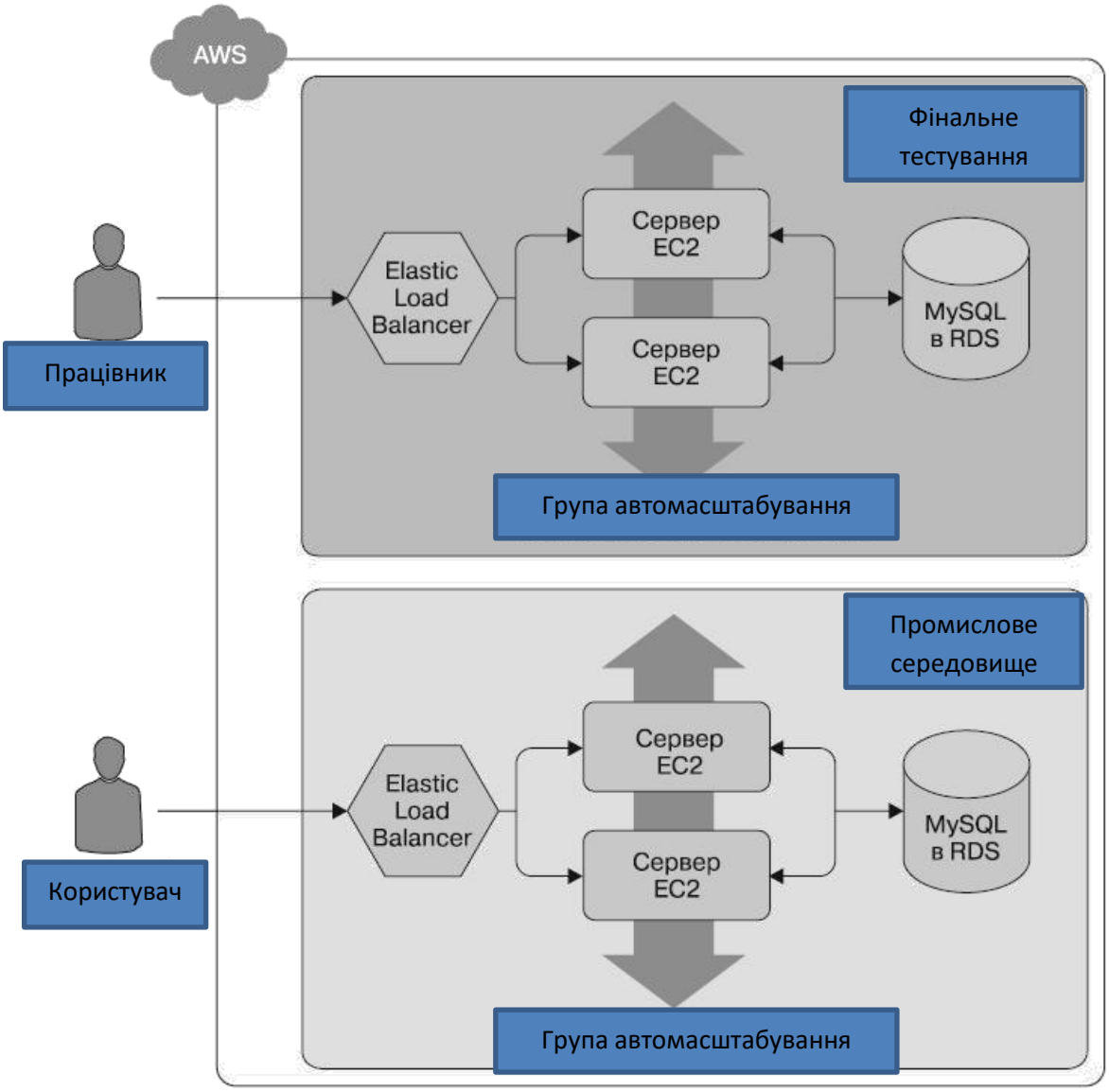


Рис. 3.33. Два середовища, кожен зі своїм балансувальником навантаження, кластером веб-серверів і базою даних

Як додати це промислове середовище без копіювання всього коду з середовища тестування? Наприклад, як уникнути дублювання всього вмісту stage / services / webserver – clusteristage / data – stores / mysql в prod/services/webserver-cluster і відповідно prod/data-stores/mysql?

В мовах загального призначення, таких як Ruby, код, який копіюється і вставляється в декількох місцях, можна оформити у вигляді функції і використовувати в інших частинах програми:

def example_function ()

puts "Hello, World" end

Інші ділянки вашого коду

example_function ()

Terraform дозволяє помістити код всередину модуля, який можна буде повторно застосовувати на різних ділянках вашої конфігурації. Замість того щоб копіювати і вставляти код в тестове середовище та середовище розробки ми зробимо так, щоб обидва середовища використовували код з одного і того ж модуля, як показано на рис. 3.34.

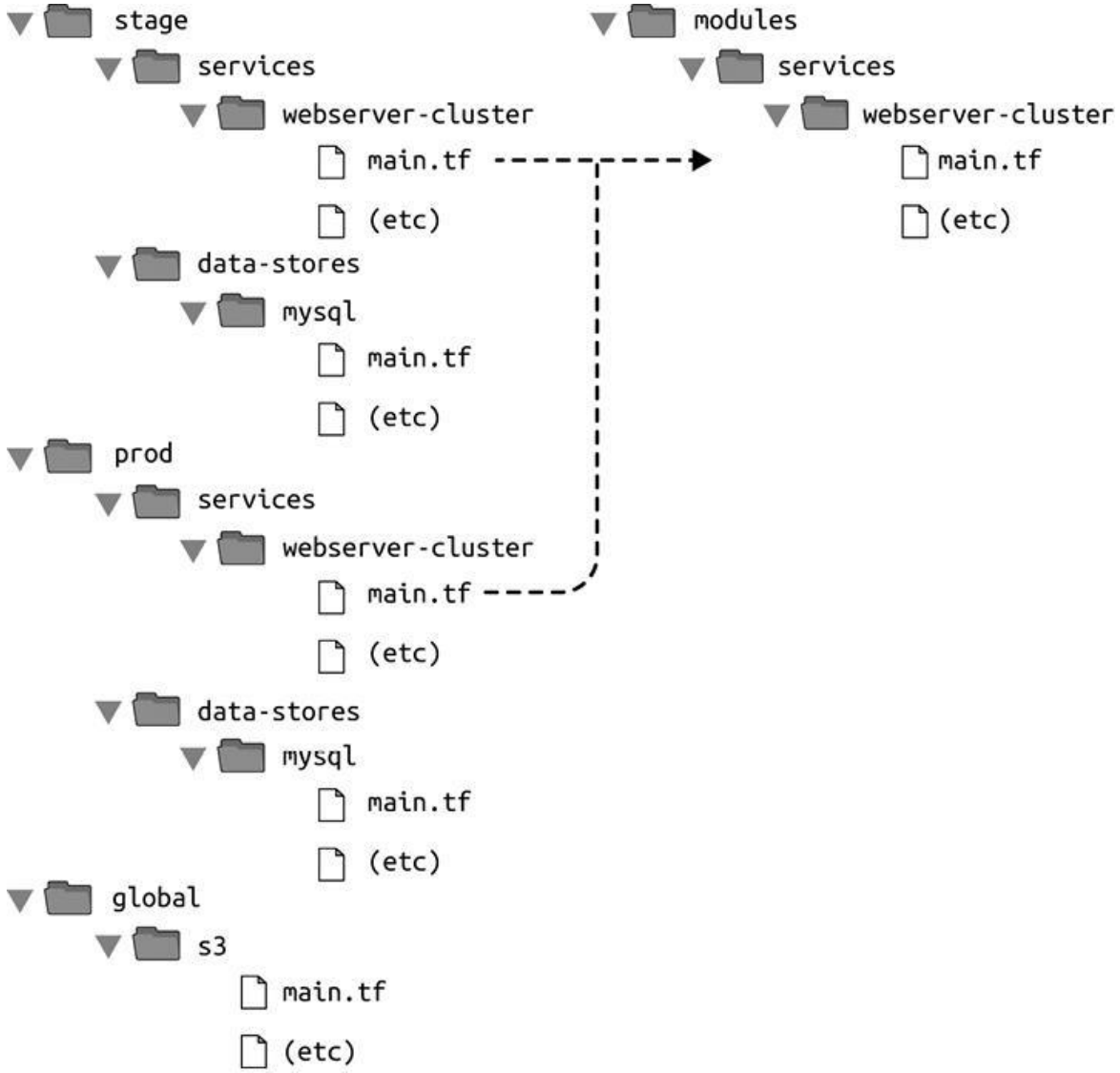


Рис. 3.34. Помістивши код в модуль, ви зможете застосувати його повторно в різних середовищах

Це дуже важливо. Модулі є ключовим аспектом написання універсального коду Terraform, який легко підтримувати і тестувати. Почавши їх використовувати, ви вже не зможете без них обійтися. Ви почнете оформляти все у вигляді модулів, об’єднувати їх в бібліотеки для

зручного використання в компанії, завантажувати сторонні модулі з Інтернету і сприймати всю свою інфраструктуру як набір універсальних модулів.

В цьому підпункті ми розглянемо, як створювати і застосовувати модулі Terraform. Ми розглянемо такі питання:

основні характеристики модулів;
вхідні параметри модулів;
локальні змінні модулів;
вихідні змінні модулів;
підводні камені;
управління версіями.

Що таке модуль?

В Terraform будь-який набір конфігураційних файлів, розміщених в одній папці, вважається **модулем**. Вся конфігурація, яку ви вже написали, формально складається з модулів, хоч і не дуже цікавих, так як ви розгортали їх безпосередньо (модуль в поточній робочій папці називається кореневим). Щоб побачити всю їх міць, потрібно використовувати один модуль з іншого.

Як приклад перетворимо в універсальний модуль код з stage/services/webserver-cluster, який включає в себе Auto Scaling Group (ASG), Application Load Balancer (ALB), групи безпеки і багато інших ресурсів.

Для початку виконайте terraformdestroy в stage/services/webserver-cluster, щоб видалити всі ресурси, створені вами раніше. Потім створіть нову папку верхнього рівня під назвою modules і перемістіть всі файли з stage/services/webserver-cluster в modules/services/webserver-cluster. В результаті ваша структура папок повинна виглядати так, як на рис. 3.35.

Відкрийте файл main.tf в modules/services/webserver-cluster і приберіть з нього визначення provider. Провайдери повинні налаштовуватися не самим модулем, а його користувачами.

Тепер можна скористатися цим модулем в тестовому середовищі, використовуючи наступний синтаксис:

```
module "<NAME>" {
source = "<SOURCE>"
[CONFIG ...]
}
```

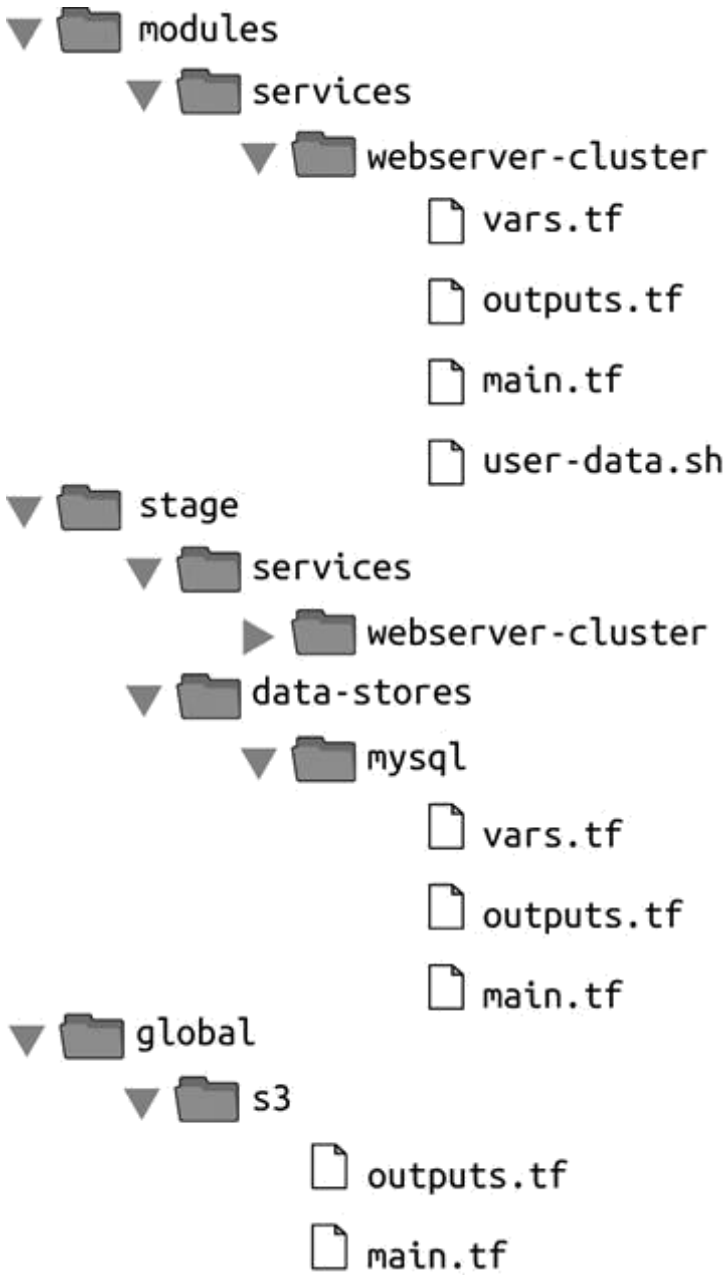


Рис. 3.35. Структура папок з модулем і тестовим середовищем

NAME – це ідентифікатор, який можна використовувати в коді Terraform для звернення до модуля (на кшталт web-service), SOURCE – шлях до коду модуля (скажімо, modules/services/webserver-cluster), а CONFIG складається з одного/декількох аргументів, призначених спеціально для цього модуля. Наприклад, ви можете створити новий файл stage/services/webserver-cluster/main.tf і використовувати в ньому модуль webserver-cluster наступним чином:

```
provider "aws" {
region = "us-east-2"
}
module "webserver_cluster" {
```

source = "../../../modules/services/
webserver-cluster "
}

Потім ви можете повторно використовувати той же модуль в промисловому середовищі, створивши новий файл prod/services/webserver-cluster/main.tf такого змісту:

provider "aws" {
region = "us-east-2"
}
module "webserver_cluster" {
source = "../../../modules/services/webserver-cluster"
}

Ось і все: повторне використання коду в різних середовищах з мінімальним дублюванням. Зверніть увагу, що при додаванні модуля в конфігурацію Terraform або зміні параметра модуля source необхідно спочатку виконати команду init, а тільки потім plan або apply:

terraform init Initializing modules ...

- webserver_cluster in ../../../modules/services/webserver-cluster Initializing the backend ...

Initializing provider plugins ...

Terraform has been successfully initialized!

Тепер ви знаєте про всі хитрощі в арсеналі команди init. Вона сама вміє завантажувати провайдери та модулі, а також конфігурувати ваші сховища.

Перш ніж застосовувати цей код, потрібно згадати про один недолік модуля webserver-cluster: всі імена в ньому прописані вручну. Це стосується груп безпеки, ALB та інших ресурсів. Таким чином, при спробі повторного використання цього модуля ви отримаєте конфлікти імен. Прямо в коді прописані навіть параметри для звернення до бази даних, оскільки файл main.tf, який ви скопіювали в modules/services/webserver-cluster, бере адресу і порт БД з джерела даних terraform_remote_state, а той написаний лише з розрахунком на тестове середовище.

Щоб виправити ці проблеми, необхідно додати в модуль webserver-cluster конфігуруємі вхідні параметри. Це дозволить йому міняти свою поведінку в залежності від оточення.

Вхідні параметри модуля

В мові програмування загального призначення, такій як Ruby, функцію можна зробити конфігурується, передавши їй вхідні параметри:

def example_function (param1, param2) puts "Hello, # {param1} # {param2}" end

Інші ділянки вашого коду example_function ("foo", "bar")

Модулі Terraform теж можуть мати параметри. Для їх визначення використовується вже знайомий вам механізм: вхідні змінні. Відкрийте файл modules/services/webserver-cluster/variables.tf і додайте три нові блоки variable:

```
variable "cluster_name" {
description = "The name to use for all the cluster resources"
type = string
}
variable "db_remote_state_bucket" {
description = "The name of the S3 bucket for the database's remote state"
type = string
}
variable "db_remote_state_key" {
description = "The path for the database's remote state in S3"
type = string
}
```

Далі пройдіться по файлу modules /service /webserver-cluster/main.tf і підставте var.cluster_name прописаних вручну імен (скажімо, "terraform-asgexample"). Наприклад, ось як це зробити в групі безпеки ALB:

```
resource "aws_security_group" "alb" {
name = "$ {var.cluster_name} -alb"
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"]
}
}
```

Зверніть увагу на те, що параметру name присвоюється "$ {var.cluster_name} -alb". Аналогічні зміни потрібно внести і в інший ресурс

aws_security_group (можете назвати його "$ {var.cluster_name} -instance"), а також в aws_alb і розділ tag ресурсу aws_autoscaling_group.

Ви також повинні оновити джерело даних terraform_remote_state, щоб воно використовувало db_remote_state_bucket і db_remote_state_key в якості параметрів bucket і відповідно key. Це дозволить йому зчитувати файл стану з правильного середовища:

```
data "terraform_remote_state" "db" {
backend = "s3"
config = {
bucket = var.db_remote_state_bucket
key = var.db_remote_state_key
region = "us-east-2"
}
}
```

Тепер можете аналогічним чином задати ці нові вхідні змінні в тестовому середовищі в файлі stage/services/webserver-cluster/main.tf:

```
module "webserver_cluster" {
source = "../../../modules/services/webserver-cluster"
cluster_name = "webservers-stage"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key = "stage / data-stores / mysql / terraform.tfstate"
}
```

Те ж саме потрібно зробити для середовища розробки в файлі prod/services/webserver-cluster/main.tf:

```
module "webserver_cluster" {
source = "../../../modules/services/webserver-cluster"
cluster_name = "webservers-prod"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key = "prod / data-stores / mysql / terraform.tfstate"
```

Промислова база даних ще не існує. В якості вправи спробуйте додати її самостійно за аналогією з тестовою.

Як бачите, для установки вхідних змінних модуля і аргументів ресурсу використовується один і той же синтаксис. Вхідні змінні є API модуля і визначають те, як він себе веде в різних середовищах. У цьому прикладі ми задаємо різні імена для різних середовищ, але ви можете зробити конфігуруємі і інші параметри. Припустимо, щоб заощадити гроші, в тестовому середовищі можна запускати невеликий кластер вебсерверів, але умовах розробки вам знадобиться великий кластер, здатний впоратися з

сильними навантаженнями. Для цього в файл modules/services/webserver-cluster/variables.tf можна додати ще три вхідні змінні:

variable "instance_type" {
description = "The type of EC2 Instances to run (e.g. t2.micro)"
type = string
}
variable "min_size" {
description = "The minimum number of EC2 Instances in the ASG" type = number
}
variable "max_size" {
description = "The maximum number of EC2 Instances in the ASG"
type = number
}

Далі потрібно оновити конфігурацію запуску в файлі modules/services/ webserver-cluster/main.tf, присвоївши параметру instance_type нову вхідну змінну var.instance_type:

resource "aws_launch_configuration" "example" {
image_id = "ami-0c55b159cbfafe1f0"
instance_type = var.instance_type
security_groups = [aws_security_group.instance.id] user_data = data.template_file.user_data.rendered

Необхідна для використання групи автомасштабування в конфігурації запуску
https://www.terraform.io/docs/providers/aws/r/launch_configuration.html

lifecycle {
create_before_destroy = true
}
}

Схожим чином слід оновити визначення ASG в тому ж файлі. Дайте параметрам min_size і max_size вхідні змінні var.min_size і var.max_size відповідно:

resource "aws_autoscaling_group" "example" {
launch_configuration = aws_launch_configuration.example.name
vpc_zone_identifier = data.aws_subnet_ids.default.ids target_group_arns = [aws_lb_target_group.asg.arn] health_check_type = "ELB" min_size = var.min_size

max_size = var.max_size

```
tag {
key = "Name"
value = var.cluster_name
propagate_at_launch = true
}
}
```

Тепер кластер у тестовому середовищі (stage / services / webserver-cluster / main.tf) можна зробити менше й дешевше, вказавши "t2.micro" для instance_type і 2 для min_size і max_size:

```
module "webserver_cluster" {
source = "../../../modules/services/webserver-cluster"
cluster_name = "webservers-stage"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key = "stage / data-stores / mysql / terraform.tfstate"
instance_type = "t2.micro"
min_size = 2
max_size = 2
}
```

Водночас в середовищі розробки можна використовувати більший тип кластеру серверів (наприклад, m4.large) з великою кількістю серверів і пам’яті. Майте на увазі, що цей тип не входить в безкоштовний тариф AWS, тому, якщо кластер вам потрібен тільки в освітніх цілях і ви не хочете платити, залиште в полі instance_type значення "t2.micro". Параметру max_size можна привласнити 10, що дозволить кластеру розширюватися і стискатися залежно від навантаження (не хвилюйтеся, спочатку він запуститься з двома серверами):

```
 module "webserver_cluster" {
source = "../../../modules/services/webserver-cluster"
cluster_name = "webservers-prod"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key = "prod / data-stores / mysql / terraform.tfstate"
instance_type = "m4.large"
min_size = 2
max_size = 10
}
```

Локальні змінні модулів

Визначення параметрів модуля за допомогою вхідних змінних – відмінний підхід, але що, якщо вам потрібно визначити змінну всередині модуля для якихось проміжних обчислень або просто для того, щоб не

дублювати код, але при цьому ви не хочете робити її доступною в якості конфігуруємого введення? Наприклад, балансувальник навантаження в модулі webserver-cluster (modules/services/webserver-cluster/main.tf) прослуховує стандартний для HTTP порт під номером 80. Зараз нам доводиться копіювати і вставляти цей номер в різних місцях, в тому числі в Прослуховувач балансувальника: resource "aws_lb_listener" "http" {load_balancer_arn = aws_lb.example.arn

port = 80

protocol = "HTTP"

За замовчуванням повертає просту сторінку з кодом

404 default_action {

type = "fixed-response" fixed_response {content_type = "text / plain"

message_body = "404: page not found" status_code = 404

}

}

}

ось група безпеки балансувальника: resource "aws_security_group" "alb" {

name = "$ {var.cluster_name} -alb" 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"]

}

}

Значення в цій групі безпеки, включаючи блок CIDR 0.0.0.0/0 (будь-які IP-адреси), номер порту 0 (будь-який порт) і довільний протокол “-1”, теж копіюються і вставляються на декількох ділянках модуля. Явне і багаторазове завдання цих “магічних” значень ускладнює читання і підтримку коду. Їх можна винести у вхідні змінні, але в такому випадку ваш модуль може (випадково) перевизначити ці значення, що може бути

небажаним. Замість цього можна визначити локальні значення в блоці locals:

```
locals {
http_port
= 80
any_port = 0
any_protocol = "-1"
tcp_protocol = "tcp"
all_ips = [ "0.0.0.0/0"]
}
```

Локальні змінні дозволяють призначити будь-якому виразу Terraform ім’я, яке потім можна використовувати в коді модуля. Такі імена видно тільки в самому модулі, тому вони не мають ніякого впливу на зовнішній код, при цьому ви не можете перезаписати їх ззовні. Щоб прочитати локальне значення, потрібне локальне посилання з наступним синтаксисом:

```
local. <NAME>
```

Використовуйте цей синтаксис для поновлення Прослуховувача балансувальника навантаження:

```
resource "aws_lb_listener" "http" {
load_balancer_arn = aws_lb.example.arn
port = local.http_port

protocol
= "HTTP"
```

За замовчуванням прослуховувач повертає просту сторінку з кодом 404

```
default_action {
type = "fixed-response"
fixed_response {
content_type = "text / plain"
message_body = "404: page not found"
status_code = 404
}
}
}
```

У всіх груп безпеки в модулі, включаючи ту, яка відноситься до балансувальника:

```
resource "aws_security_group" "alb" {name = "$ {var.cluster_name} -alb"
ingress {
```

```
from_port = local.http_port
to_port = local.http_port
protocol = local.tcp_protocol
cidr_blocks = local.all_ips
}
egress {
from_port = local.any_port
to_port = local.any_port
protocol = local.any_protocol
cidr_blocks = local.all_ips
}
}
```

Локальні змінні спрощують читання і підтримку коду, тому використовуйте їх як можна частіше.

Вихідні змінні модуля

Потужною особливістю груп ASG є можливість конфігурувати їх для збільшення і зменшення кількості запущених серверів в залежності від навантаження. Для цього можна скористатися *запланованою дією*, яка буде змінювати розмір кластера в заданий час доби. Наприклад, якщо ваш кластер відчуває підвищене навантаження в робочий час, ви можете запланувати збільшення і зменшення кількості серверів на 9 ранку і 5 вечора відповідно.

Запланована дія, визначена в модулі webserver-cluster, відноситься як до тестового, так і до промислового середовища. Оскільки вам не потрібно подібного роду масштабування під час тестування, можете поки визначити графік автомасштабування прямо в промисловій конфігурації.

Щоб визначити заплановану дію, додайте наступних два ресурси aws_autoscaling_schedule в файл prod/services/webserver-cluster/main.tf:

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours"
{
scheduled_action_name = "scale-out-during-business-hours"
min_size = 2
max_size = 10
desired_capacity = 10
recurrence = "09 ***"
}
resource "aws_autoscaling_schedule" "scale_in_at_night" {
scheduled_action_name = "scale-in-at-night"
min_size = 2
```

max_size = 10

desired_capacity = 2

recurrence = "017 ***"

}

Перший ресурс aws_autoscaling_schedule використовується для збільшення кількості серверів до десяти в ранковий час (в параметрі recurrence використовується синтаксис cron, тому "09 ***" означає “в 9 ранку кожен день”), а другий зменшує цей показник на ніч ("017 ** * “означає” в 5 вечора кожен день “). Однак в обох випадках не вистачає параметра autoscaling_group_name, який задає ім’я ASG. Сама група ASG визначається всередині модуля webserver-cluster. Як же отримати доступ до її імені? У мовах загального призначення, таких як Ruby, функції можуть повертати значення:

def example_function (param1, param2)

return "Hello, # {param1} # {param2}"

end

Інші ділянки вашого коду

return_value = example_function ("foo", "bar")

Terraform модулі теж можуть повертати значення. Для цього використовується вже знайомий вам механізм: вихідні змінні. Ви можете додати ім'я ASG в якості вихідної змінної у файлі modules/services/webserver-cluster/outputs.tf, як показано нижче:

output "asg_name" {

value = aws_autoscaling_group.example.name description = "The name of the Auto Scaling Group"

}

Для звернення до вихідних змінних модуля використовується наступний синтаксис:

module. <MODULE_NAME>. <OUTPUT_NAME>

Наприклад:

module.frontend.asg_name

Цей синтаксис можна використовувати у файлі prod / services / webserver-cluster / main.tf, щоб встановити параметр autoscaling_group_name в кожному з ресурсів aws_autoscaling_schedule:

resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {scheduled_action_name = "scale-out-during-business-hours"

min_size = 2

max_size = 10

desired_capacity = 10

recurrence = "09 ***"

autoscaling_group_name = module.webserver_cluster.asg_name

}

resource "aws_autoscaling_schedule" "scale_in_at_night" {scheduled_action_name = "scale-in-at-night"

min_size = 2

max_size = 10

desired_capacity = 2

recurrence = "017 ***"

autoscaling_group_name = module.webserver_cluster.asg_name

}

Можливо, вам слід зробити доступним ще одне вихідне значення в модулі webserver-cluster: доменне ім'я ALB. Так ви будете знати, яку URL-адресу потрібно перевірити після розгортання кластера. Для цього в файлі /modules/services/webserver-cluster/outputs.tf необхідно ще раз додати вихідну змінну:

output "alb_dns_name" {

value = aws_lb.example.dns_name

description = "The domain name of the load balancer"

}

Після цього даний висновок можна “пропустити через” файли

stage / services / webserver-cluster / outputs.tf і prod / services / webserver-cluster / outputs.tf:

output "alb_dns_name" {

value = module.webserver_cluster.alb_dns_name description = "The domain name of the load balancer"

}

Ваш кластер вебсерверів майже готовий до розгортання. Залишилося тільки взяти до уваги кілька підводних каменів.

Підводні камені

При створенні модулів звертайте увагу:

на файлові шляхи;

вкладені блоки.

Файлові шляхи

Раніше ви помістили скрипт користувальницьких даних для кластера вебсерверів у зовнішній файл, user-data.sh, і застосували вбудовану функцію file, щоб прочитати його з диска. Неочевидний момент функції file полягає в тому, що файловий шлях, який вона використовує, повинен бути

відносним (оскільки Terraform можна запускати на безлічі різних комп'ютерів) – але відносно чого?

За замовчуванням Terraform інтерпретує цей шлях щодо поточної робочої папки. Це не викликає проблем, якщо ви використовуєте функцію file в файлі конфігурації Terraform в тій же папці, з якої виконується команда terraformapply (тобто якщо функція file застосовується в кореневому модулі). Але якщо зробити те ж саме в модулі, розміщеному в окремій папці, нічого не буде працювати.

Вирішити цю проблему можна за допомогою виразу, відомого як "**посилання на шлях**", яке має вигляд path. <TYPE>. Terraform підтримує такі типи цих посилань:

path.module – повертає шлях до модуля, в якому визначено вираз;

path.root – повертає шлях до кореневого модуля;

path.cwd – повертає шлях до поточної робочої папки. При нормальному використанні Terraform це значення збігається з path.root, але в деяких нестандартних випадках Terraform запускається не з папки кореневого модуля, що призводить до розбіжності цих шляхів.

Для скрипта користувальницьких даних потрібен шлях, взятий відносно самого модуля, тому в джерелі даних template_file в файлі modules/services/webser-vercluster/main.tf слід застосовувати path.module:

data "template_file" "user_data" {
template = file ("$ {path.module} /user-data.sh")
vars = {
server_port = var.server_port
db_address = data.terraform_remote_state.db.outputs.address db_port = data.terraform_remote_state.db.outputs.port
}
}

Вкладені блоки

Конфігурацію деяких ресурсів Terraform можна визначати окремо або у вигляді вкладених блоків. При створенні модулів завжди слід віддавати перевагу окремим ресурсам.

Наприклад, ресурс aws_security_group дозволяє визначати вхідні та вихідні правила у вигляді вкладених блоків. Ви вже це бачили в модулі webserver-cluster (modules / services / webserver-cluster / main.tf):

resource "aws_security_group" "alb" {
name = "$ {var.cluster_name} -alb"
ingress {
from_port = local.http_port

```
to_port= local.http_port
protocol= local.tcp_protocol
cidr_blocks = local.all_ips
}
egress {
from_port = local.any_port
to_port = local.any_port
protocol = local.any_protocol
cidr_blocks = local.all_ips
}
}
```

Ви повинні модифікувати цей модуль так, щоб ті ж вхідні та вихідні правила визначалися у вигляді окремих ресурсів aws_security_group_rule (не забудьте зробити це для обох груп безпеки в даному модулі):

```
resource "aws_security_group" "alb" {
name = "$ {var.cluster_name} -alb"
}
resource "aws_security_group_rule" "allow_http_inbound" {
type = "ingress"
security_group_id = aws_security_group.alb.id from_port = local.http_port

to_port = local.http_port
protocol= local.tcp_protocol
cidr_blocks = local.all_ips
}
resource "aws_security_group_rule" "allow_all_outbound" {
type = "egress"
security_group_id = aws_security_group.alb.id
from_port = local.any_port
to_port = local.any_port
protocol = local.any_protocol
cidr_blocks = local.all_ips
}
```

При спробі одночасного використання вкладених блоків і окремих ресурсів ви отримаєте помилки, коли правила маршрутизації конфліктують і скасовують один одного. Тому ви повинні вибрати щось одне. У зв’язку з цим обмеженням, при створенні модуля, завжди слід використовувати окремі ресурси замість вкладених блоків. В іншому випадку ваш модуль вийде менш гнучким і конфігурованим.

Наприклад, якщо всі вхідні і вихідні правила в модулі webserver-cluster визначені у вигляді окремих ресурсів aws_security_group_rule, ви зможете зробити цей модуль досить гнучким для того, щоб дозволити користувачам додавати власні правила за його межами. Для цього ідентифікатор aws_security_group потрібно експортувати у вигляді вихідної змінної всередині modules / services / webserver-cluster / outputs.tf:

```
output "alb_security_group_id" {
value = aws_security_group.alb.id
description = "The ID of the Security Group attached to the load balancer"
}
```

Тепер уявіть, що вам потрібно зробити доступним ззовні додатковий порт, суто для тестування. Тепер це легко зробити: потрібно лише додати в файл stage/services/webserver-cluster/main.tf ресурс aws_security_group_rule:

```
resource "aws_security_group_rule" "allow_testing_inbound" {
type = "ingress"
security_group_id = module.webserver_cluster.alb_security_group_id
from_port = 12345
to_port = 12345
protocol = "tcp"
cidr_blocks = [ "0.0.0.0/0"]
}
```

Якби ви визначили вхідне чи вихідне правило у вигляді вкладеного блоку, цей код був би неробочим. Варто зазначити, що ця ж проблема характерна для цілого ряду ресурсів Terraform, включаючи наступні:

aws_security_group і aws_security_group_rule;

aws_route_table і aws_route;

aws_network_acl і aws_network_acl_rule.

Тепер ви готові розгорнути свій кластер вебсерверів відразу в тестовому і промисловому середовищах. Виконайте terraformapply і насолоджуйтеся роботою з двома окремими копіями своєї інфраструктури.

Мережева ізоляція

Показані приклади створюють два середовища, ізольовані як в вашому коді Terraform, так і з точки зору інфраструктури: у них є окремі балансувальники навантаження, сервери і бази даних. Але, незважаючи на це, вони не ізольовані на рівні мережі. Щоб не ускладнювати приклади коду, всі ресурси в цій книзі розгортаються в одній і тій же віртуальній приватній хмарі (VPC). Це означає, що сервери в тестовому і промисловому середовищі можуть взаємодіяти між собою.

У реальних сценаріях використання запуску двох оточень в одній хмарі VPC загрожує відразу двома ризиками. По-перше, помилка, допущена в одному середовищі, може вплинути на інше. Наприклад, якщо при внесенні змін в тестовому оточенні ви випадково зламаєте правила в таблиці маршрутизації, це позначиться на перенаправлення трафіку і в промислових умовах. По-друге, якщо зловмисник отримає доступ в одному середовищі, він зможе проникнути і в інше. Якщо ви активно змінюєте код в тестовому оточенні і випадково залишите відкритим який-небудь порт, будь-який хакер, що проник всередину, зможе заволодіти не тільки тестовими, а й промисловими даними.

У зв'язку з цим, якщо не брати до уваги прості приклади і експерименти, ви повинні розміщувати кожне середовище в окремій хмарі VPC. Для більшої впевненості оточення можна навіть рознести по різних облікових записів AWS.

Управління версіями

Якщо ваші тестове і промислове середовище посилаються на папку з одним і тим же модулем, будь-яка зміна в цій папці торкнеться і того й іншого при наступному ж розгортанні. Такого роду зв'язування ускладнює тестування змін до ізоляції від промислового оточення. Замість цього краще використовувати різні версії модулів: наприклад, v0.0.2 для тестового середовища і v0.0.1 для середовища розробки, як показано на рис. 3.26.

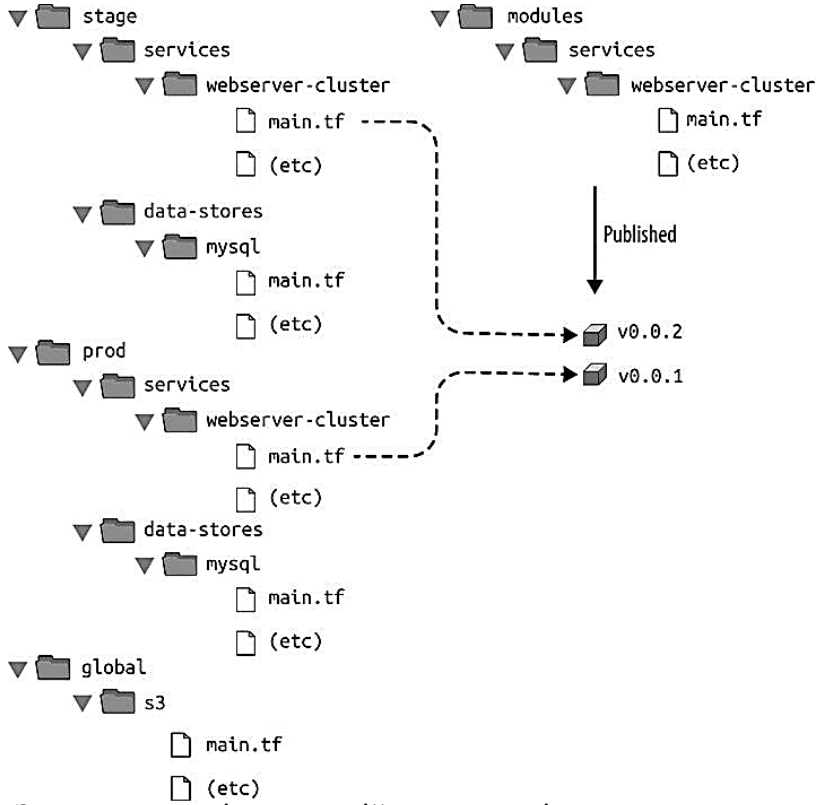


Рис. 3.36. Застосування різних версій модуля в різних середовищах

У всіх прикладах з модулями, які ви бачили до цих пір, параметр source містив локальний файловий шлях. Але, крім файлових шляхів, модулі Terraform підтримують і інші види джерел, такі як URL-адреси Git/Mercurial і довільні URL44. Найпростіший спосіб управління версіями модуля – розміщення його коду в окремому Git-репозиторії, URL-адреса якого потім прописується в параметрі source. Це означає, що код Terraform буде розподілений (як мінімум) на двох репозиторіях:

modules – в цьому репозиторії знаходяться універсальні модулі. Кожен модуль – свого роду "креслення", яке описує певну частину вашої інфраструктури;

live – репозиторій, який містить поточну інфраструктуру, розгорнуту в кожному окрузі (stage, prod, mgmt і т. Д.). Це такі "будівлі", які ви будуєте по "кресленнях", взятих зі сховищ modules.

Оновлена структура папок для вашого коду Terraform буде виглядати приблизно так, як на рис. 3.27.

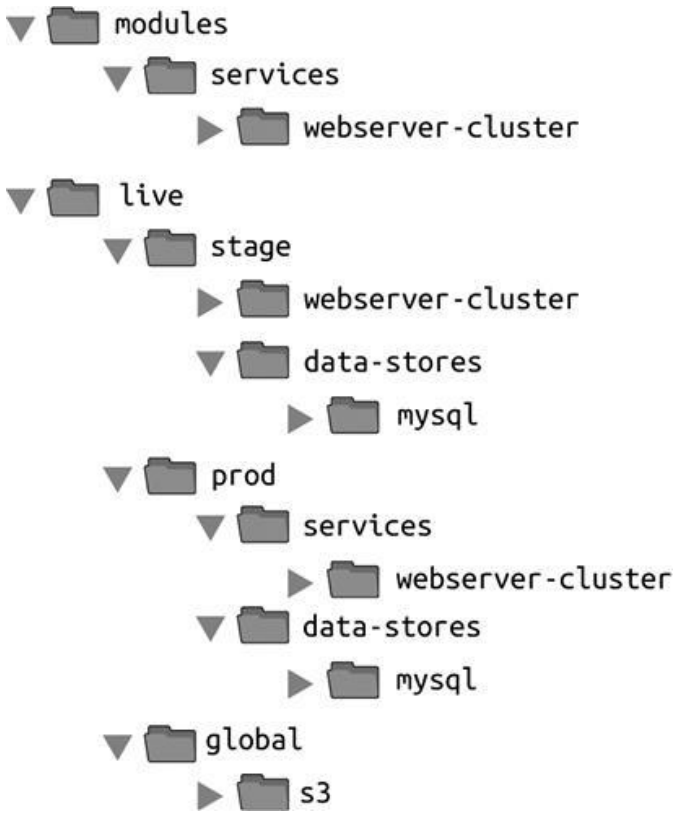


Рис. 3.27. Структура файлів і каталогів з декількома репозиторіями

Щоб організувати код таким чином, спочатку потрібно перемістити папки stage, prod і global в папку під назвою live. Потім ви повинні рознести папки live і modules по окремим Git-репозиторіям. Ось приклад того, як це робиться з папкою modules:

```
cd modules
git init
git add.
git commit -m "Initial commit of modules repo"
git remote add origin "(URL OF REMOTE GIT REPOSITORY)"
git push origin master
```

Сховища modules можна також призначити тег, який буде використовуватися в якості номера версії. Якщо ви працюєте з сервісом GitHub, це можна зробити в його інтерфейсі: створіть випуск (bit.ly/2Yv8kPg), який автоматично створить тег. Якщо ви не використовуєте GitHub, можете застосувати утиліту командного рядка Git:

```
git tag -a "v0.0.1" -m "First release of webserver-cluster module"
git push --follow-tags
```

Тепер ви можете працювати з різними версіями модуля в тестовому і промисловому середовищах, вказавши URL-адресу Git в параметрі source. Ось як це буде виглядати в файлі live/stage/services/webserver-cluster/main.tf, якщо ваше сховище modules знаходиться в GitHub за адресою github.com/foo/modules (майте на увазі, що подвійна коса риска в URL-адресі Git є обов'язковою):

```
module "webserver_cluster" {
source = "github.com/foo/modules//webserver-cluster?ref=v0.0.1"
cluster_name = "webservers-stage"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key = "stage / data-stores / mysql / terraform.tfstate"
instance_type = "t2.micro"
min_size = 2
max_size = 2
}
```

Якщо ви хочете використовувати різні версії модулів без метушні з Git-репозиторіями, можете завантажити модуль з архіву з кодом для цієї книги (https://github.com/brikis98/terraform-up-and-running-code). Мені довелося розбити цю адресу на частини, щоб вона розмістилася на сторінці. Його слід записувати одним рядком:

```
source="github.com/brikis98/terraform-up-and-running-code//
code/terraform/04-terraform-module/module-
example/modules/services/webserver-cluster?ref=v0.1.0"
```

Параметр ref дозволяє вказати певну фіксацію Git по її хешу SHA1, ім'я гілки або, як в даному прикладі, конкретний тег Git. В цілому, радять використовувати в якості версій модулів теги. Імена гілок нестабільні, так

як ви завжди отримуєте останню фіксацію в заданій гілці, яка може змінюватися при кожному виконанні команди init, а хеші SHA1 виглядають малозрозумілими. Теги Git такі ж стабільні, як і фіксації (насправді це просто покажчики на фіксації), але при цьому вони дозволяють застосовувати зручні і розбірливі назви.

Особливо корисною схемою іменування тегів є семантичне версіонування (http://semver.org). Це система управління версіями в форматі MAJOR.MINOR.PATCH (наприклад, 1.0.4) з окремими правилами щодо того, як слід інкрементувати кожен елемент номера версії. Ви повинні інкрементувати:

версію MAJOR при внесенні несумісних змін в API;

версію MINOR при додаванні можливостей з дотриманням забезпечення сумісності;

версію PATCH при виправленні помилок з дотриманням забезпечення сумісності.

Семантичне версіонування дозволяє донести до користувачів модуля, якого роду зміни ви внесли і як це позначається на процесі оновлення.

Оскільки ви оновили свою конфігурацію з використанням різних URL-адрес для різних версій вашого модуля, потрібно заново виконати команду terraforminit, щоб завантажити його код:

$ Terraform init Initializing modules ...

Downloading git@github.com: brikis98 / terraform-up-and-running-code.git? ref = v0.1.0

for webserver_cluster ...

(...)

На цей раз ви можете бачити, що Terraform завантажує код модуля з Git, а не з вашої локальної файлової системи. Коли все буде готово, ви зможете виконати команду apply як зазвичай.

Закриті Git-репозиторії

Якщо ваш модуль знаходиться в закритому Git-репозиторії, щоб застосовувати цей репозиторій як джерело модуля, потрібно дозволити Terraform в ньому аутентифікуватися. Рекомендовано використовувати аутентифікацію SSH, щоб не довелося зберігати облікові дані для доступу до сховища в самому коді. Кожен розробник зможе створити SSH-ключ і прив’язати його до користувача Git. Після додавання ключа в ssh-agent Terraform буде автоматично використовувати його для аутентифікації, якщо в якості URL джерела вказано SSH45.

URL-адреса джерела повинна виглядати так:

git@github.com: <OWNER>/<REPO>.git//<PATH>?ref=<VERSION>

Наприклад:

git@github.com: acme/modules.git//example?ref=v0.1.2

Щоб перевірити, чи коректно ви відформатували URL, спробуйте клонувати базову адресу в терміналі за допомогою git clone:

$ Git clone git@github.com: <OWNER> / <REPO> .git

Якщо ця команда виконається успішно, Terraform теж зможе використовувати ваш приватний репозиторій.

Тепер пройдемося по процесу внесення змін у проекті з різними версіями модулів. Припустимо, ви модифікували модуль webserver-cluster і хочете перевірити його в тестовому середовищі. Для початку зміни потрібно зафіксувати в репозиторії modules:

cd modules

git add.

git commit -m "Made some changes to webserver-cluster"

git push origin master

Потім в тому ж репозиторії потрібно створити новий тег:

git tag -a "v0.0.2" -m "Second release of webserver-cluster"

git push --follow-tags

Тепер ви можете перевести на нову версію тільки ту URL-адресу, яка використовується в тестовому середовищі (live/stage/services/webserver-cluster/main.tf):

module "webserver_cluster" {

source = "git@github.com: foo / modules.git // webserver-cluster? ref = v0.0.2"

cluster_name = "webservers-stage"

db_remote_state_bucket = "(YOUR_BUCKET_NAME)"

db_remote_state_key = "stage / data-stores / mysql / terraform.tfstate"

instance_type = "t2.micro"

min_size = 2

max_size = 2

}

В середовищі розробки (live/prod/services/webserver-cluster/main.tf) можна, як і раніше, використовувати версію v0.0.1 без будь-яких змін: module "webserver_cluster" {

source = "git@github.com: foo / modules.git // webserver-cluster? ref = v0.0.1"

cluster_name = "webservers-prod"

db_remote_state_bucket = "(YOUR_BUCKET_NAME)"

db_remote_state_key = "prod / data-stores / mysql / terraform.tfstate"
instance_type = "m4.large"
min_size = 2
max_size = 10
}

Після того як ви ретельно перевірили версію v0.0.2 і переконалися в її коректності в тестовому середовищі, можете оновити і середовище розробки. Але якщо в v0.0.2 виявиться помилка, це не складе великої проблеми, оскільки користувачі реальної системи не будуть потурбовані. Виправте помилку, випустіть нову версію і повторіть весь процес заново, поки ваш модуль не стане досить стабільним для промислового застосування.

Розробка модулів

Управління версіями модулів відмінно підходить, коли розгортання відбувається в загальному оточенні (тестовому або промисловому), але, якщо ви просто займаєтеся тестуванням на власному комп'ютері, краще використовувати локальні файлові шляхи. Це прискорить розробку, оскільки після внесення змін до коду модулів ви зможете відразу ж виконати команду apply в активних папках, без фіксації свого коду і публікації нової версії.

Резюме

Описуючи IaC у вигляді модулів, ви отримуєте можливість використовувати у своїй інфраструктурі різноманітні рекомендовані методики програмування: розбирати і тестувати кожну зміну, що вноситься в модуль; створювати для кожного модуля випуски з семантичними версіями; безпечно експериментувати з різними версіями модулів у різних середовищах і в разі якоїсь проблеми відкотитися до попереднього випуску.

Все це може істотно допомогти в побудові інфраструктури швидким і надійним чином, оскільки розробники зможуть повторно використовувати її компоненти, які були як слід перевірені і задокументовані. Наприклад, ви можете створити канонічний модуль, який описує процес розгортання одного мікросервіса, включаючи те, як запускати кластер, масштабувати його в залежності від навантаження і розподіляти запити між серверами. Потім кожна команда зможе застосовувати цей модуль для управління власними мікросервісами, і для цього буде достатньо лише кількох рядків коду.

Щоб такий модуль підійшов відразу декільком командам, його код повинен бути гнучким і конфігурованим. Наприклад, одна команда може

використовувати його для розгортання одного примірника свого мікросервіса без балансувальника навантаження, а інший може знадобитися десяток примірників з розподілом трафіку між ними.

Всі подробиці про URL-адреси джерел можна знайти на сторінці bit.ly/2TaXmZF.

Гарне керівництво по роботі з SSH-ключами можна знайти за адресою bit.ly/2ZFLJwe.

Контрольні запитання

1. Що таке DevOps?
2. Які основні переваги використання інфраструктури як коду?
3. Як працює Terraform?
4. Що таке Google Cloud Platform?
5. Що таке Amazon Web Services?
6. Що таке Microsoft Azure?
7. Поясніть поняття хмарного провайдера?
8. Які переваги використання хмарних технологій?
9. В яких типах хмар Terraform має можливість формувати інфраструктуру?
10. Як створити нового користувача AWS?
11. Як встановити Terraform?
12. Які способи аутентифікації підтримує Terraform?
13. Що таке LHC?
14. Як користуватися документацією в Terraform?
15. Як розгорнути сервер в Terraform?
16. Як розгорнути вебсервер в Terraform?
17. Як розгорнути кластер вебсерверів в Terraform?
18. Як розгорнути балансувальник навантаження в Terraform?
19. Як керувати станом Terraform?
20. Поясніть поняття стану в Terraform
21. Де зберігаються файли стану в Terraform?
22. Які є обмеження до сховищ в Terraform?
23. Які види ізоляцій файлів стану в Terraform ви знаєте?
24. Як уникнути дублювання стану?
25. Що таке Terraform remote state?
26. Що таке модуль Terraform?

4. БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ ТА РОЗГОРТАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Системи контролю версій

Поняття системи контролю версій

Система контролю версій (далі – СКВ) – це система, що записує зміни у файл або набір файлів протягом деякого часу, так що ви зможете повернутися до певної версії пізніше. Вона дозволяє повернути вибрані файли до попереднього стану, повернути весь проєкт до попереднього стану, побачити зміни, побачити, хто останній міняв щось і спровокував проблему, хто вказав на проблему і коли, та багато іншого. Використання СКВ також в цілому означає, що якщо ви зламали щось або втратили файли, ви просто можете все виправити. Крім того, ви отримаєте все це за дуже невеликі накладні витрати.

Локальні системи контролю версій

Багато людей в якості одного з методів контролю версій застосовують копіювання файлів в окрему директорію (можливо навіть директорію з відміткою за часом, якщо вони достатньо розумні). Даний підхід є дуже поширеним завдяки його простоті, проте він, неймовірним чином, схильний до появи помилок. Можна легко забути в якій директорії ви знаходитеся і випадково змінити не той файл або скопіювати не ті файли, які ви хотіли.

Щоб справитися з цією проблемою, програмісти давно розробили локальні СКВ, що мають просту базу даних, яка зберігає всі зміни в файлах під контролем версій.

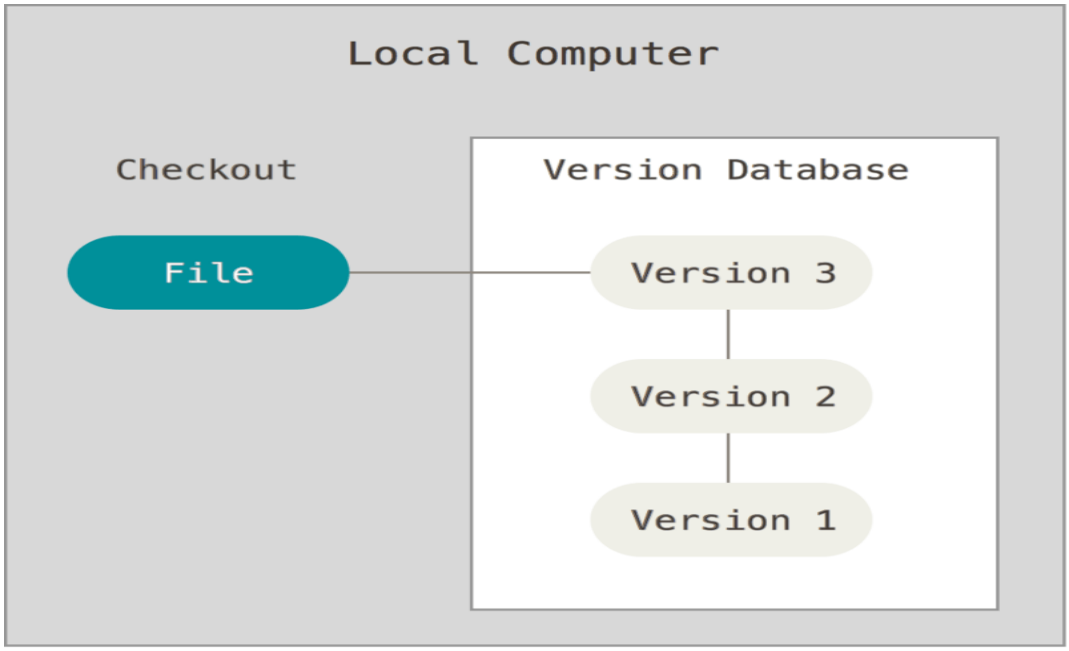


Рис. 4.1. Схема побудови локальної системи контролю версій

Одним з найбільш поширених інструментів СКВ була система під назвою RCS, яка досі поширюється з багатьма комп'ютерами сьогодні. RCS зберігає набори "латок" (тобто, відмінності між файлами) в спеціальному форматі на диску; він може заново відтворити будь-який файл, як він виглядав, в будь-який момент часу, шляхом додавання всіх латок.

Централізовані системи контролю версій

Наступним важливим питанням, з яким стикаються люди, є необхідність співпрацювати з іншими розробниками. Щоб справитися з цією проблемою, були розроблені централізовані системи контролю версій (далі – ЦСКВ). Такі системи як CVS, Subversion і Perforce, мають єдиний сервер, який містить всі версії файлів, та деяке число клієнтів, які отримують файли з центрального місця. Протягом багатьох років, це було стандартом для систем контролю версій.

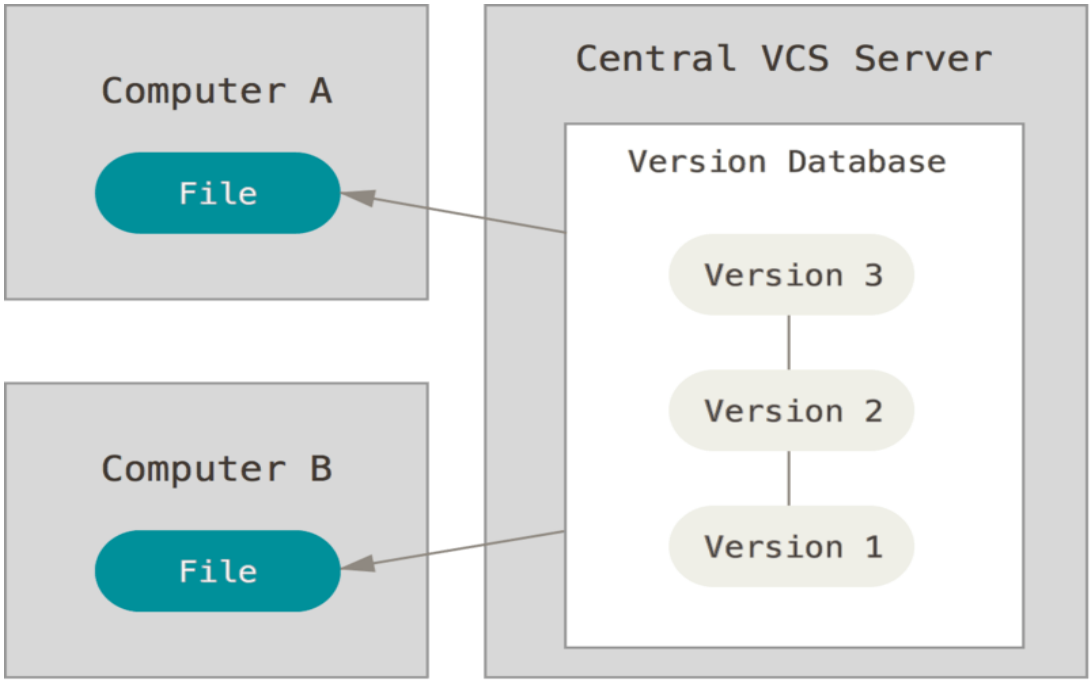


Рис. 4.2. Схема побудови централізованої системи контролю версій

Такий підхід має безліч переваг, особливо над локальними СКВ. Наприклад, кожному учаснику проєкту відомо, певною мірою, чим займаються інші. Адміністратори мають повний контроль над тим, хто і що може робити. Набагато легше адмініструвати ЦСКВ, ніж мати справу з локальними базами даних для кожного клієнта.

Але цей підхід також має деякі серйозні недоліки. Найбільш очевидним є єдина точка відмови, яким є централізований сервер. Якщо сервер виходить з ладу протягом години, то протягом цієї години ніхто не може співпрацювати або зберігати зміни над якими вони працюють під версійним контролем. Якщо жорсткий диск центральної бази даних на

сервері пошкоджено, і своєчасні резервні копії не були зроблені, ви втрачаєте абсолютно все – всю історію проєкту, крім одиночних знімків проекту, що збереглися на локальних машинах людей. Локальні СКВ страждають тією ж проблемою – щоразу, коли вся історія проєкту зберігається в одному місці, ви ризикуєте втратити все.

Децентралізовані системи контролю версій

В децентралізованих системах контролю версій (Git, Mercurial, Bazaar або Darcs), клієнти не просто отримують останній знімок файлів репозиторія: натомість вони є повною копією сховища разом з усією його історією. Таким чином, якщо вмирає який-небудь сервер, через який співпрацюють розробники, будь-який з клієнтських репозиторіїв може бути скопійований назад до серверу, щоб відновити його. Кожна копія дійсно є повною резервною копією всіх даних. Більш того, багато з цих систем дуже добре взаємодіють з декількома віддаленими репозиторіями, так що ви можете співпрацювати з різними групами людей, застосовуючи різні підходи в межах одного проекту одночасно. Це дозволяє налаштувати декілька типів робочих процесів, таких як ієрархічні моделі, які неможливі в централізованих системах.

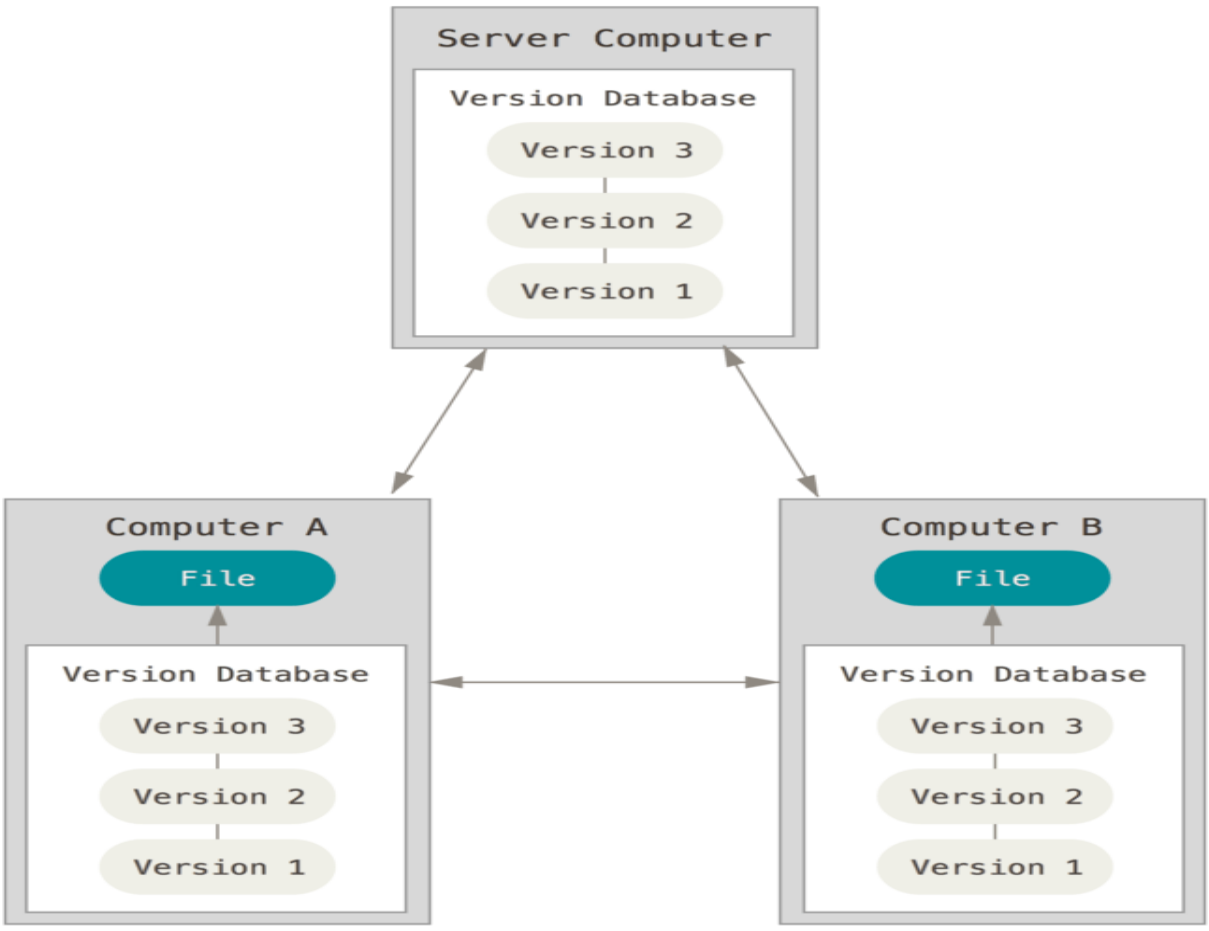


Рис. 4.3. Схема побудови децентралізованої системи контролю версій

Основи Git

Git – розподілена система керування версіями файлів та спільної роботи. Проект створив Лінус Торвальдс для керування розробкою ядра Linux, а сьогодні підтримується Джуніо Хамано. Git є однією з найефективніших, надійних і високопродуктивних систем керування версіями, що надає гнучкі засоби нелінійної розробки, які базуються на відгалуженні і злитті гілок. Для забезпечення цілісності історії та стійкості до змін заднім числом використовуються криптографічні методи, також можлива прив'язка цифрових підписів розробників до тегів і комітів.

Основною відмінністю від інших систем (таких як Subversion та подібних їй) є те, як Git сприймає дані. Концептуально, більшість СКВ зберігають інформацію як список файлових редагувань. Ці, інші системи (CVS, Subversion, Perforce, Bazaar тощо) розглядають інформацію як список файлів та змін кожного з них протягом деякого часу (це зазвичай називають контроль версій оснований на дельтах).

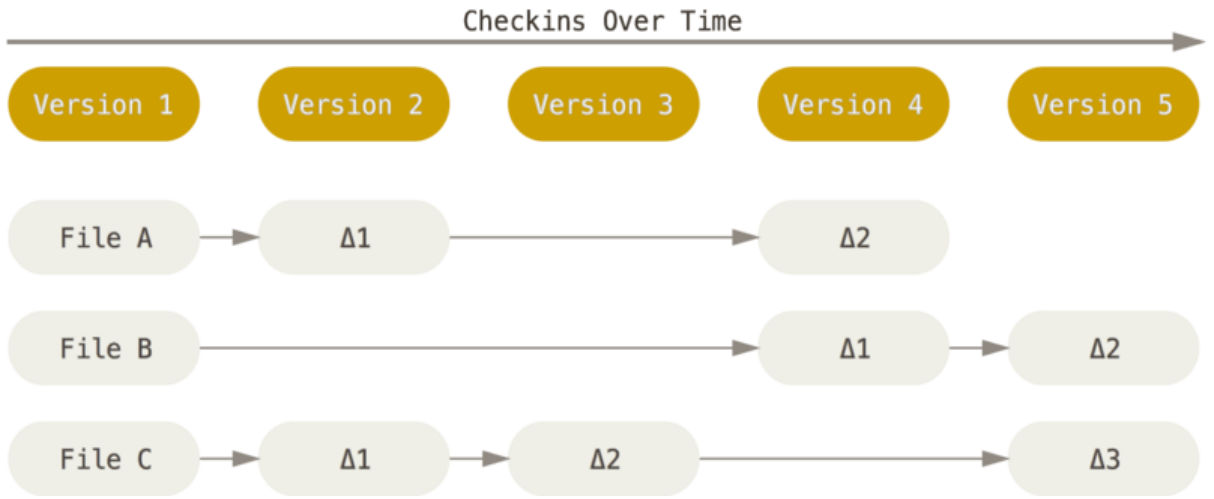


Рис. 4.4. Схема контролю версій на основі дельт

Git не оброблює та не зберігає свої дані таким чином. Замість цього, Git сприймає свої дані радше як низку знімків мініатюрної файлової системи. У Git щоразу, як ви створюєте коміт, тобто зберігаєте стан вашого проєкту, Git запам'ятовує як виглядають всі ваші файли в той момент і зберігає посилання на цей знімок. Для ефективності, якщо файли не змінилися, Git не зберігає файли знову, просто робить посилання на попередній ідентичний файл, котрий вже зберігається. Git вважає свої дані більш як потік знімків.

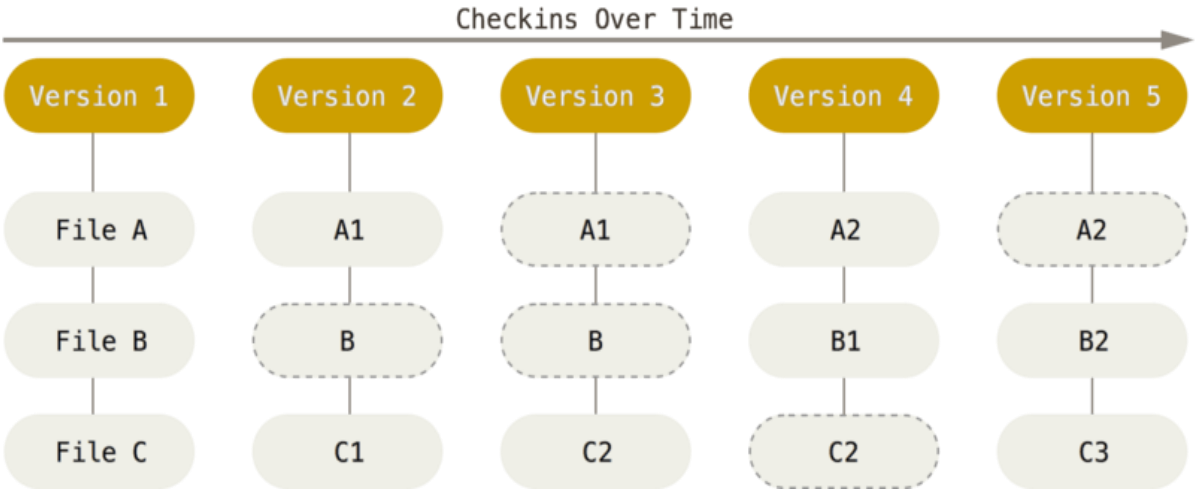


Рис. 4.5. Схема контролю версій на основі знімків файлів

Це дуже важлива різниця між Git та майже всіма іншими СКВ. З цієї причини в Git було заново переосмислено майже кожен аспект контролю версій, які інші системи просто копіювали з попереднього покоління. Це зробило Git – мініатюрну файлову систему з деякими неймовірно потужними вбудованими інструментами, більш схожою на додаток, а не просто СКВ.

Майже кожна операція локальна

Більшість операцій у Git потребують лише локальних файлів та ресурсів для здійснення операцій – немає необхідності в інформації з інших комп'ютерів вашої мережі. Через те, що повна історія проєкту знаходиться на вашому локальному диску, більшість операцій здійснюються майже миттєво.

Наприклад, для перегляду історії проєкту, Git не має потреби брати її з серверу, він просто зчитує її прямо з локальної бази даних. Це означає, що ви отримуєте історію проекту не встигнувши кліпнути оком. Якщо ви бажаєте переглянути відмінності між поточною версією файла та його редакцією місячної давності, Git знайде копію збережену місяць тому і проведе локальне обчислення різниці замість того, щоб звертатись за цим до віддаленого серверу чи спочатку робити запит на отримання старішої версії файла.

Git – цілісний

Будь-що в Git, перед збереженням, отримує контрольну суму, за якою потім і можна на нього посилатися. Таким чином, неможливо змінити файл чи директорію так, щоб Git про це не дізнався. Цей функціонал вбудовано в систему на найнижчих рівнях і є невід'ємною частиною її філософії. Ви не можете втратити інформацію при передачі чи отримати пошкоджений файл без відома Git.

Механізм, який використовується для цього контролю, називається хешування типу SHA-1. Він являє собою 40-символьну послідовність цифр та перших літер латинського алфавіту (a-f) і вираховується на основі вмісту файла чи структури директорії в Git. При роботі з Git ви повсюди зустрічатимете такі хеші, адже Git постійно їх використовує. Фактично, Git зберігає все не за назвою файла, а саме за значенням хешу його змісту.

Три стани

Git має три основних стани, в яких можуть перебувати ваші файли: збережений у коміті (commited), змінений (modified) та індексований (staged):

– збережений у коміті – означає, що дані безпечно збережено в локальній базі даних;

– змінений – означає, що у файл внесено редагування, які ще не збережено в базі даних;

– індексований стан виникає тоді, коли ви позначаєте змінений файл у поточній версії, щоб ці зміни ввійшли до наступного знімку коміту.

З цього випливають три основні частини проекту під управлінням Git: *директорія Git, робоче дерево та індекс*.

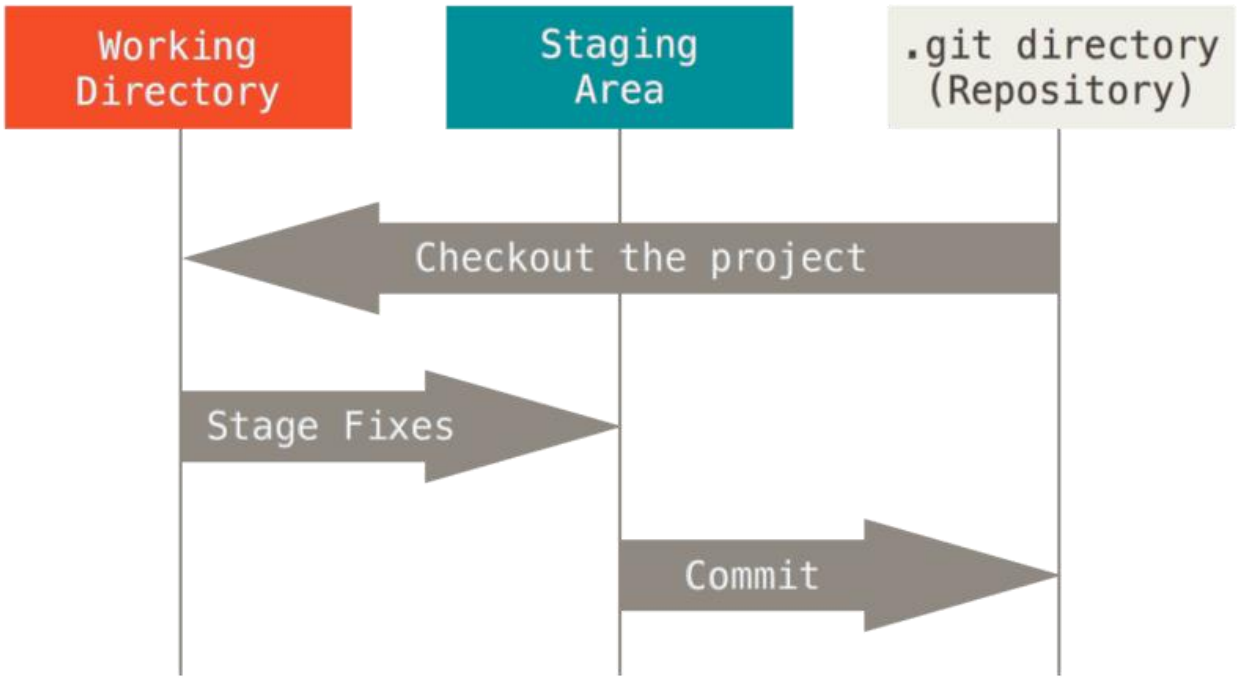


Рис. 4.6. Схема основних частин проекту під керівництвом Git

У **директорії Git** система зберігає метадані та базу даних об’єктів вашого проєкту. Це найважливіша частина Git, саме вона копіюється при клонуванні сховища з іншого комп’ютера.

Робоче дерево – це одна окрема версія проекту, взята зі сховища. Ці файли видобуваються з бази даних у теці Git та розміщуються на диску для подальшого використання та редагування.

Індекс – це файл, що зазвичай знаходиться в директорії Git і містить інформацію про те, що буде збережено у наступному коміті. Також цей файл називають "областю додавання" (staging area), проте ми переважно будемо користуватись технічним терміном Git "індекс".

Найпростіший процес взаємодії з Git виглядає приблизно так:

ви редагуєте файли у своїй робочій директорії;

вибірково надсилаєте до індексу лише ті зміни, що їх ви бажаєте зберегти в наступному коміті, і лише ці зміни буде збережено в індексі;

створюєте коміт – знімок з індексу остаточно зберігається в директорії Git.

У випадку, якщо окрема версія файлу вже є в директорії Git, цей файл вважається збереженим у коміті. Якщо він зазнав змін і перебуває в індексі, то він індексований. Якщо ж його стан відрізняється від того, який був у коміті, і файл не знаходиться в індексі, то він називається зміненим.

4.2. Інструмент контролю версій GitLab

Gitlab – це сервіс, що забезпечує віддалений доступ до репозиторіїв Git. Крім розміщення вашого коду, служби надають додаткові функції, призначені для управління життєвим циклом розробки програмного забезпечення. Ці додаткові функції включають управління спільним використанням коду між людьми, відстеження помилок, Wiki та інші інструменти для спільного кодування.

Історія

GitLab був створений Дмитром Запорожцем і Валерієм Сизовим в жовтні 2011 року. Він був поширений по ліцензії MIT, а стабільна версія GitLab 10.4 випущена 22 січня 2018 року.

Навіщо використовувати GitLab?

GitLab – відмінний спосіб керувати репозиторіями git на централізованому сервері. GitLab дає вам повний контроль над вашими репозиторіями або проектами і дозволяє вам вирішувати, чи є вони публічними або приватними безкоштовно.

GitLab – це платформа для управління Git-репозиторіями.

GitLab пропонує безкоштовні публічні та приватні репозиторії, відстеження проблем і Wiki.

GitLab – це зручний вебінтерфейс поверх Git, який збільшує швидкість роботи з Git.

GitLab надає власну систему безперервної інтеграції (CI) для управління проєктами та надає користувальницький інтерфейс поряд з іншими функціями GitLab.

GitLab надає користувачам версію GitLab “Community Edition” для встановлення та налаштування на власному сервері.

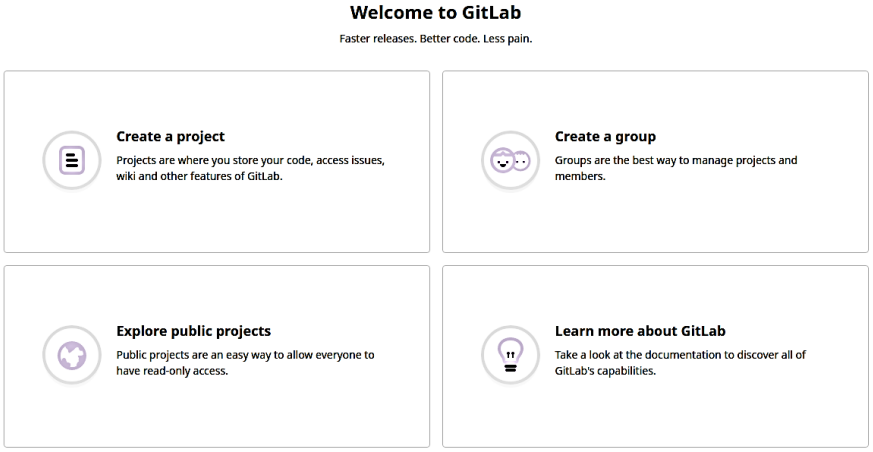


Рис. 4.7. Вигляд робочого вікна GitLab

Основні поняття Gitlab

GitLab – це сервіс, який організації можуть використовувати для забезпечення внутрішнього управління git-репозиторіями. Це самостійна система управління Git-репозиторієм, яка зберігає для користувача код в секреті і може легко впроваджувати зміни коду.

Репозиторій – файли управляємі git.

Коміт – зміна репозиторію.

Гілка – це файл, що містить 50 символів контрольної суми SHA-1 коміту, на який вказує. Створити гілку так же швидко, як записати 41 байт до файла.

Основні команди для роботи з Gitlab

Інсталяція Gitlab на Linux (Ubuntu 16.04 - 20.04)

Крок 1 – Завантажте залежності:

sudo apt-get update;

sudo apt-get install -y curl openssh-server ca-certificates tzdata.

Крок 2 – Додайте пакетний репозиторій Gitlab:

curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh | sudo bash.

Крок 3 – Встановіть пакет Gitlab-ce та перезавантажте:

sudo EXTERNAL_URL="https://[ваш домен]" apt-get install gitlab-ce;

sudo gitlab-ctl reconfigure.

Перейшовши в браузер і ввівши https://[ваш домен] ви перейдете на ваш локальний Gitlab.

Інсталяція Git на Linux (Ubuntu 20.04 LTS) – за допомогою команди sudo apt install git-all.

Інсталяція Git на MacOS – за допомогою бінарного інсталятора з сайту http://git-scm.com/download/mac.

Рис. 4.8. Встановлення Git на MacOS

На Windows – перейдіть до http://git-scm.com/download/win і завантаження почнеться автоматично.

Налаштування git

Додайте ім'я користувача Git та адресу електронної пошти, щоб визначити автора при передачі інформації за допомогою команди:

git config --global user.name "[ім'я користувача]";

git config --global user.email "[електронна пошта]".

Налаштування SSH доступу до Gitlab

Крок 1 – Щоб створити SSH ключ, перейдіть до командного рядка і на своєму ПК та введіть:

ssh-keygen.

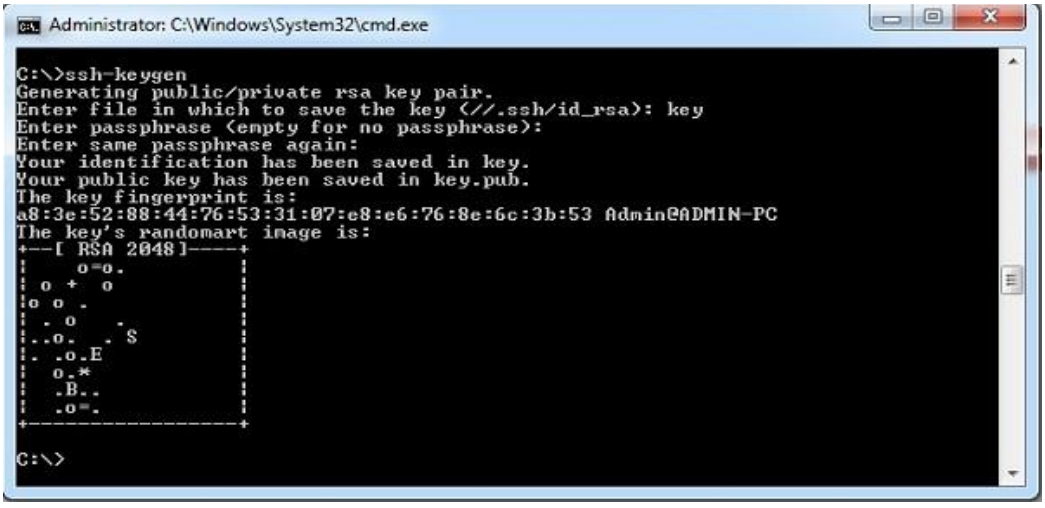


Рис. 4.9. Встановлення SSH-з'єднання

Крок 2 – Увійдіть на свій акаунт в Gitlab та виберіть опцію “Settings”.

Крок 3 – Щоб встановити ключ SSH, натисніть вкладку “SSH Keys” в лівій частині меню.

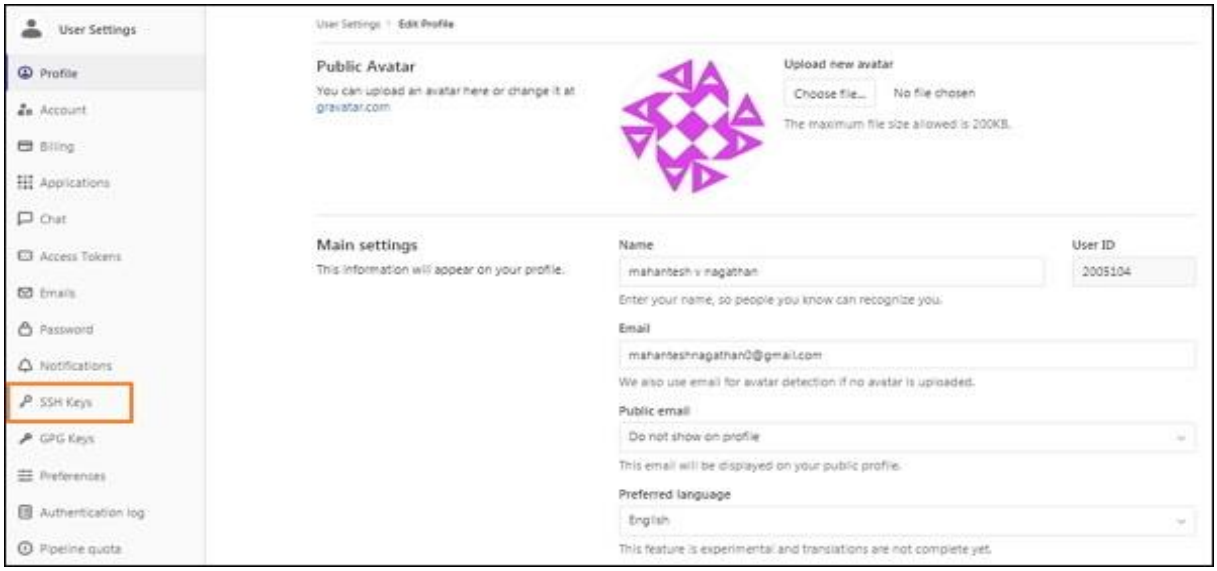


Рис. 4.10. Встановлення SSH-з’єднання

Крок 4 – Перейдіть на диск C, ви побачите файл з розширенням .pub, який був створений на першому кроці.

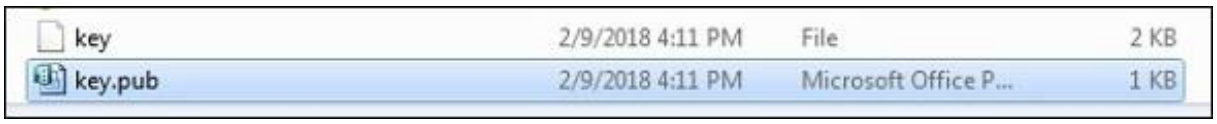


Рис. 4.11. Створений публічний ключ

Крок 5 – Скопіюйте вміст файла та вставте в поле для SSH ключа.

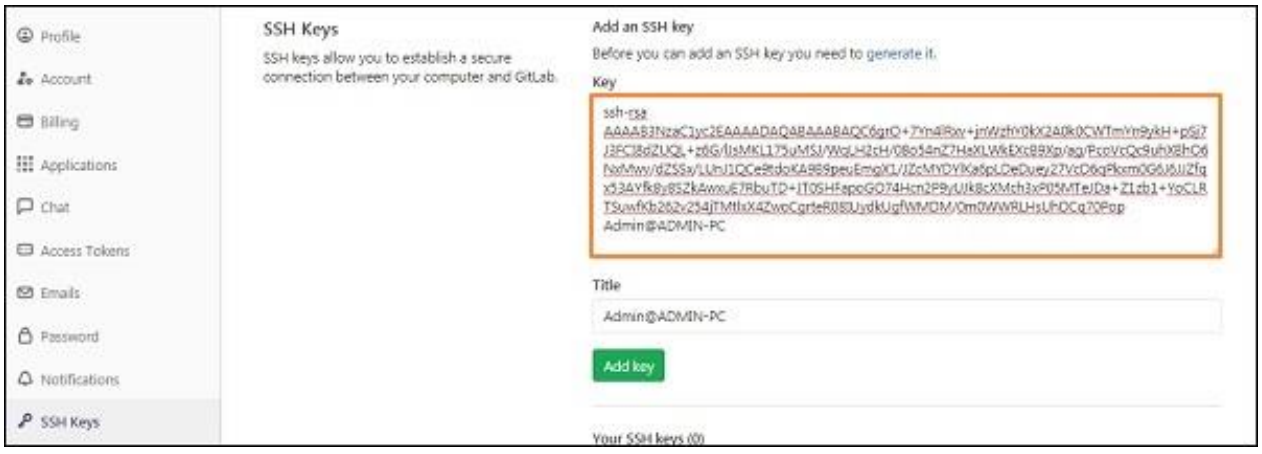


Рис. 4.12. Додавання публічного ключа

Крок 6 – Натисніть клавішу “Add key”, щоб додати ключ SSH в ваш GitLab.

Робота з проєктом

Крок 1 – Щоб створити новий проєкт, перейдіть на свій акаунт GitLab та натисніть “New project” на панелі інструментів.

Рис. 4.13. Панель інструментів у вікні GitLab

Крок 2 – Введіть назву проєкту, опис, рівень доступності та натисніть “Create project”.

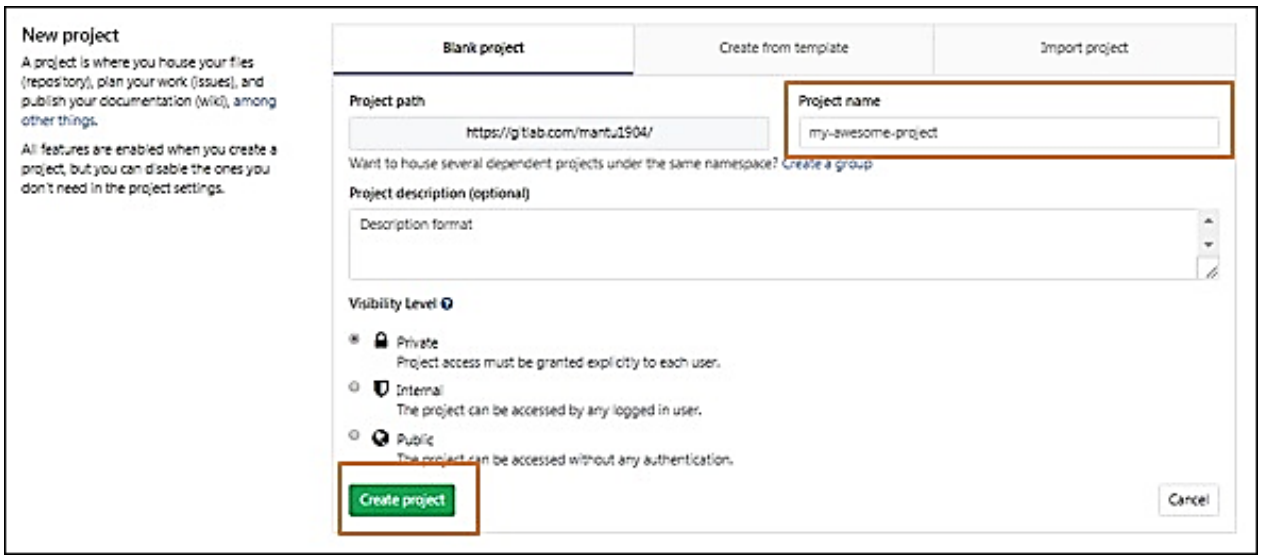


Рис. 4.14. Створення нового проєкту в GitLab

Крок 3 – Щоб працювати з проєктом його потрібно склонувати собі на ПК. Перейдіть до свого проєкту, натисніть клавішу “Clone” та скопіюйте посилання яке знаходиться під “Clone with SSH”.

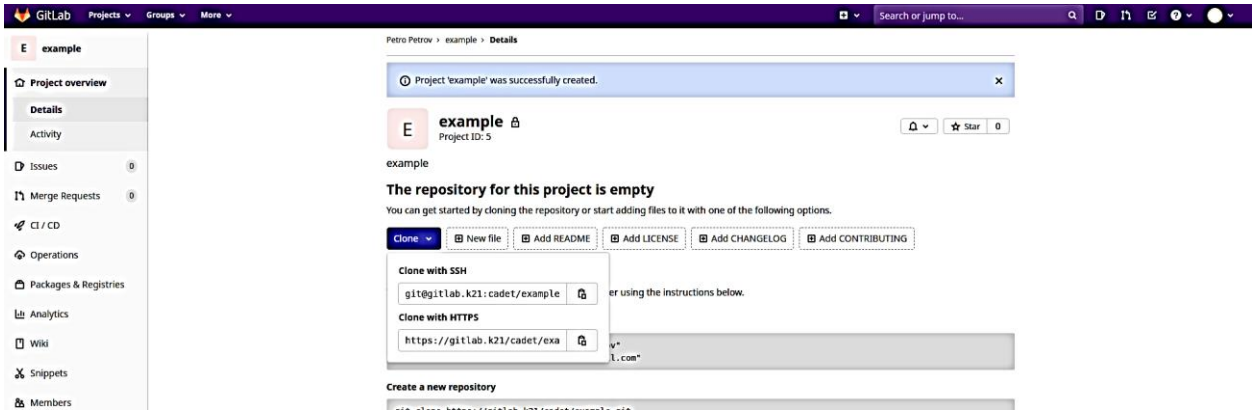
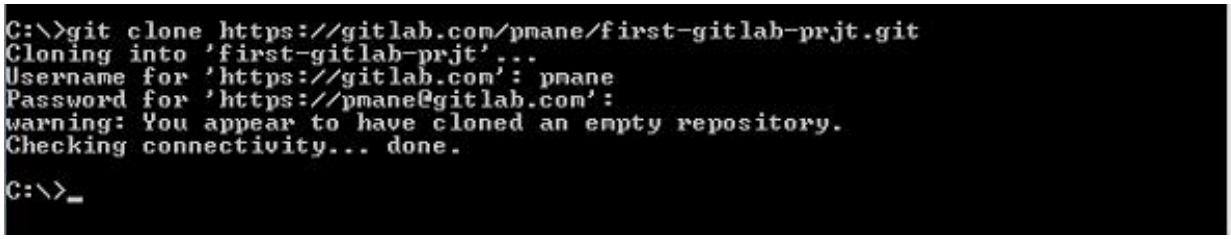


Рис. 4.14. Клонування проєкту GitLab

Крок 4 – Перейдіть в командному рядку в директорію куди хочете клонувати проект та введіть команду:

– git clone [посилання скопійоване в третьому кроці].



Крок 5 – Тепер перейдіть в клонований вами проєкт та створіть файл.

Крок 6 – Додайте створений вами файл у відслідковування git командою:

– git add [ваш файл]

Крок 7 – Збережіть зміни проєкту командою:

– git commit -m “[назва зміни]”.

Крок 8 – Відішліть зміни на GitLab:

– git push -u orign master.

Робота з гілками

Щоб створити нову гілку та перейти на неї, виконайте команду:

– git branch [назва гілки];

– git checkout [назва гілки].

Для злиття гілок виконайте команду:

– git checkout master;

– git merge [назва гілки].

Для видалення гілки виконайте команду:

– git branch -d [назва гілки].

4.3. Поняття постійної інтеграції програмного забезпечення

Багатьом проєктам розробки програмного забезпечення притаманна дивна, але дуже поширена властивість: протягом майже всього процесу розробки додаток знаходитися в непрацездатному стані. Фактично більшість додатків, розроблювальних великими командами, значну частину часу в процесі розробки знаходяться в непридатному для використання стані. Причина цьому очевидна: ніхто не зацікавлений у використанні додатку, розробка якого не завершена. Розробники реєструють зміни і можуть навіть виконувати автоматичні модульні тести, але ніхто не намагається запустити додаток у повному обсязі та використовувати його в працюючому середовищі.

Це справедливо для проєктів, в яких використовуються довгоживучі гілки версій або приймальні тести відкладаються на завершальний етап.

Часто в розклад таких проєктів закладена тривала фаза інтеграції в кінці розробки, щоб дати команді час виконати злиття гілок і підготувати додаток до приймального тестування. Часто виявляється, що, коли проект підходить до фази інтеграції, додаток ще не готовий. Періоди інтеграції можуть зайняти дуже довгий час, і що найгірше, ніхто не може надійно передбачити, наскільки довгий.

З іншого боку, існує чимало проєктів, в яких після внесення чергової зміни додаток знаходиться в непрацездатному стані всього декілька хвилин. Різниця між цими двома проєктами полягає в застосуванні технології неперервної інтеграції. Ця технологія передбачає, що кожен раз, коли хтось фіксує зміну, додаток проходить автоматичне збирання та тестування. Якщо процес збирання або тестування невдалий, то команда розробників негайно відкладає всі свої справи і не повертається до них, поки не усунуть проблему. Головна мета неперервної інтеграції полягає в тому, щоб додаток постійно знаходився в працездатному стані. Головна ідея неперервної інтеграції полягає в наступному: якщо звичайна інтеграція приносить явну користь кодовій базі, чому б не використовувати її неперервно? У контексті інтеграції слово “неперервно” означає, що вона повинна виконуватися щоразу, коли хтось фіксує зміну в системі керування версіями.

Концепція неперервної інтеграції викликає зрушення парадигми програмування. Без неперервної інтеграції додаток вважався непрацездатним, поки хтось не доведе зворотнє, зазвичай на стадії тестування або інтеграції. При використанні неперервної інтеграції додаток вважається працездатним практично відразу ж після кожної зміни. Команди, які використовують неперервну інтеграцію, мають можливість постачати ПЗ набагато частіше і з меншою кількістю помилок. Помилки виявляються на ранніх стадіях процесу постачання, коли усунути їх набагато легше, в результаті чого економиться час.

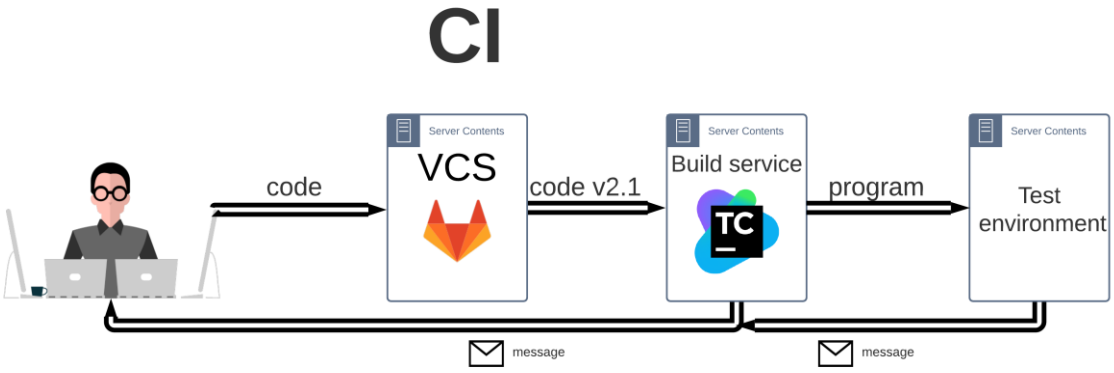


Рис. 4.15. Процес неперервної інтеграції

Початкові вимоги

Система керування версіями. Все, що використовується в проєкті, повинно бути в єдиному сховищі системи керування версіями, включаючи програмний код, тести, сценарії баз даних, сценарії збирання та

розгортання, а також все, що потрібно для створення, інсталяції, виконання та тестування додатку.

Рис. 4.16. Приклад СКВ

Автоматизоване збирання проєкту. Обов'язково потрібно мати можливість запускати збирання додатку з командного рядка. Можна розпочати з утиліти командного рядка, яка запускає сценарій збирання та тестування в середовищі розробки. Також дозволяється використання складної колекції сценаріїв збирання, які складаються з багатьох стадій.

Середовища розробки і засоби неперервної інтеграції стали вдосконаленими, тому є можливість виконувати збирання та тестування без допомоги командного рядка. Але бажано мати сценарії збирання, які запускаються з командного рядка без середовища розробки.

Рис. 4.17. Приклади систем автоматизованого збирання проектів

Згода команди. Неперервна інтеграція – це технологія, а не інструмент. Від команди розробників вона вимагає дисципліни та згоди використати її. Кожен учасник команди повинен часто реєструвати невеликі зміни в системі керування версіями.

Поняття неперервного розгортання ПЗ

Найбільш важлива проблема, яка стоїть перед розробниками при розгортанні ПЗ: як в короткий строк представить користувачам чергову версію додатку після додавання в неї оновлень? Нові корисні ідеї з'являються часто. Окрім того в додатку можуть виявитися помилки, які потрібно швидко виправити.

Неперервне розгортання сприяє швидкому, автоматизованому та повторювальному оновленню додатку та усуненню помилок. Неперервне розгортання є результатом логічного розвитку принципу неперервної інтеграції.

Зворотній зв'язок також важливий для реалізації постійних автоматизованих постачань нових версій.

Кожні зміни повинні ініціювати зворотній зв'язок

Працездатні додатки інколи корисно умовно розділити на чотири компоненти: код, що виконується, конфігурація, робоче середовище і дані.

Зміна будь-якого з цих компонентів може змінити поведінку додатку. Потрібно контролювати всі чотири компоненти та забезпечити можливість верифікації при кожній зміні.

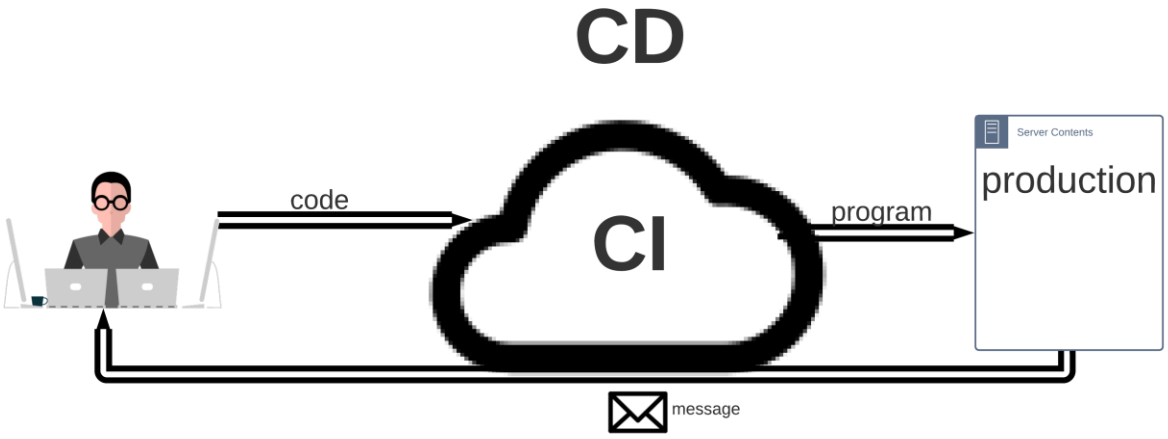


Рис. 4.18. Процес неперервного розгортання програмного забезпечення

Зворотній зв’язок повинен виконуватися якомога швидше

Ключ до швидкого зворотнього зв’язку – автоматизація. В повністю автоматизованому процесі одне обмеження – кількість та параметри обладнання, яке можливо задіяти для вирішення задач.

4.4. Інструменти для системи CI/CD

Системи контролю версій

GitLab – вебдодаток і система управління репозиторіями програмного коду для Git.

GitLab пропонує рішення для зберігання коду та спільної розробки масштабних програмних проєктів. Репозиторій включає в себе систему контролю версій для розміщення різних ланцюжків розробки та гілок, дозволяючи розробникам перевіряти код.

GitLab підтримує як публічні, так і необмежену кількість приватних гілок розробки.

GitLab має безкоштовну версію, яку можна встановити в себе на сервері.

GitHub – сервіс онлайн-хостингу репозиторіїв, що володіє всіма функціями розподіленого контролю версій і функціональністю управління вихідним кодом, тобто все, що підтримує Git і навіть більше. Зазвичай він використовується разом з Git і дає розробникам можливість зберігати їх код онлайн, а потім взаємодіяти з іншими розробниками в різних проєктах.

Рис. 4.19. Емблема GitHub

Також GitHub може похвалитися контролем доступу, управлінням завданнями і wiki для кожного проєкту. Мета GitHub – сприяти взаємодії розробників.

До проєкту, що завантажений на GitHub, можна отримати доступ за допомогою інтерфейсу командного рядка Git і Git-команд. Також є й інші функції, такі як документація, запити на прийняття змін (pull requests), історія коммітів, інтеграція з безліччю популярних сервісів, email-повідомлення, емодзі, графіки, вкладені списки завдань, та ін.

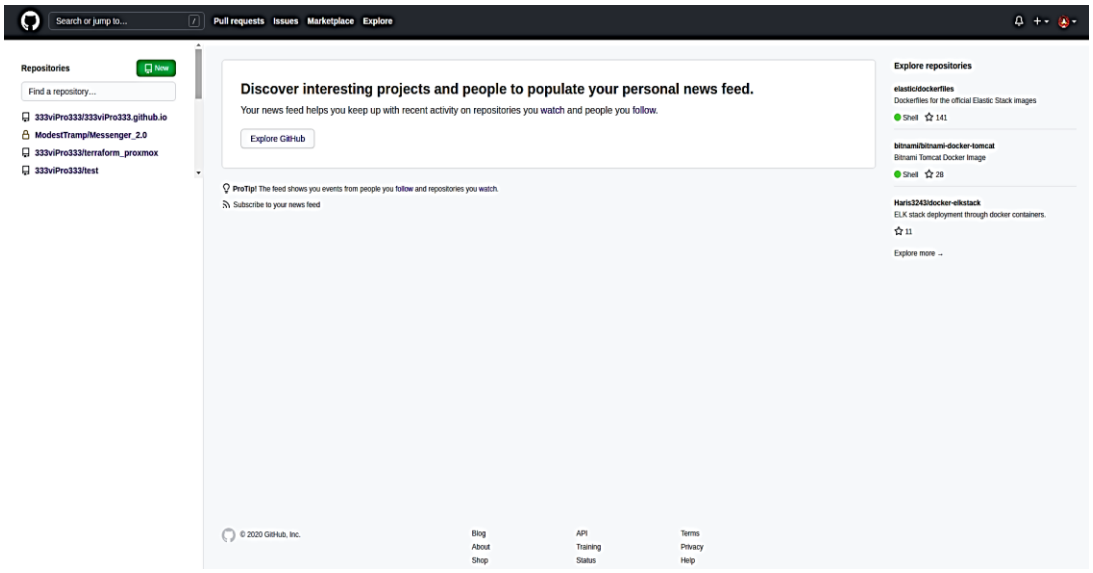


Рис. 4.20. Вигляд графічного інтерфейсу GitHub

Bitbucket – це вебсервіс для хостингу проєктів і їх спільної розробки, на базі системи контролю версій Mercurial і Git. За цілями застосування і функціональності аналогічний GitHub, хоч останній не надає безкоштовні “закриті” репозиторії.

Рис. 4.21. Емблема Bitbucket

В даний час всім користувачам безкоштовно надаються такі можливості:

дисковий простір в 2 ГБ на репозиторій;

необмежена кількість публічних репозиторіїв;

необмежена кількість приватних репозиторіїв для команд до п’яти учасників;

доступ до репозиторіїв за протоколами HTTP і SSH;

можливість прив'язати обліковий запис на сервісі до власного домену;

вікі (окремо для кожного сховища, можна відключити);

система обліку помилок (окремо для кожного сховища, можна відключити);

інтеграція з Google Analytics, Twitter, Basecamp і іншими службами;

RSS-стрічка історії змін;

управління приватних даних окремо для кожного сховища;

для публічних репозиторіїв кількість користувачів не обмежена (BitBucket безкоштовний для проектів відкритого програмного забезпечення);

до приватного (закритого) сховища може мати доступ до п'яти користувачів; більшу кількість записів надається в рамках платного обслуговування (від $ 10 до $ 200 на місяць) або після запрошення нових користувачів.

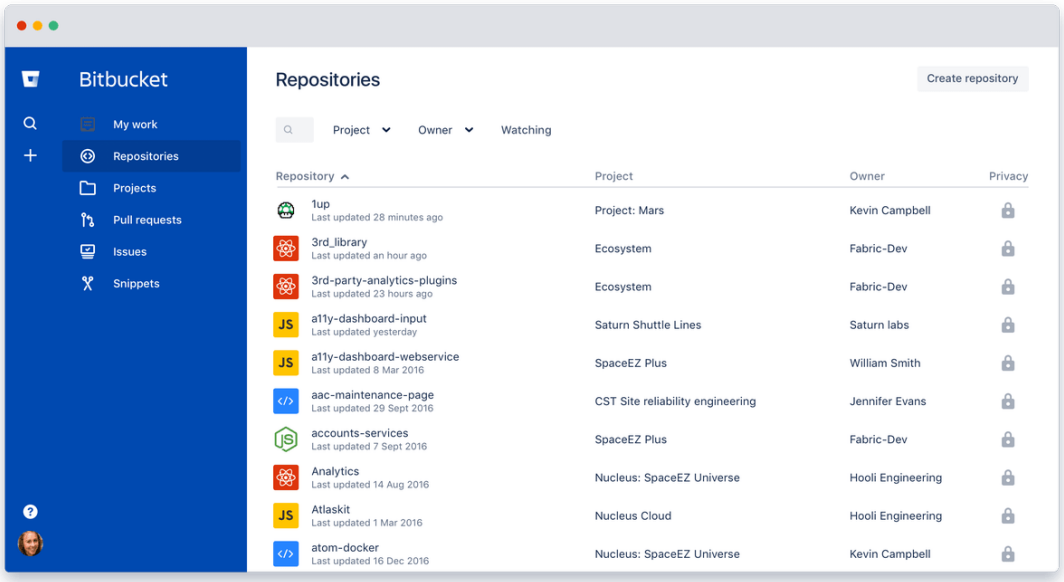


Рис. 4.22. Графічний інтерфейс Bitbucket

Інструменти CI/CD

GitLab CI/CD – це інструмент, вбудований в GitLab для налаштування системи CI/CD.

Рис. 4.23. Емблема GitLab CI/CD

Процес CI/CD в GitLab працює наступним чином:

виконується push змін в репозиторій проєкту;

якщо в корені проекту є файл .gitlab-ci.yml, то GitLab розуміє, що для цього проекту потрібно виконати процес CI/CD;

GitLab шукає запущений runner налаштований для даного проєкту;

GitLab передає файл .gitlab-ci.yml раннеру, який виконує команди, описані в цьому файлі;

після виконання команд раннер повертає в GitLab результати, які показані в вкладці “pipelines”.

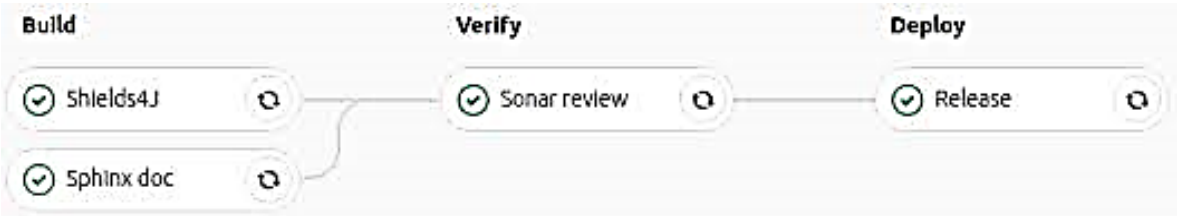


Рис. 4.24. Вигляд пайплайну GitLab CI/CD

```yaml
stages:
  - test
  - deploy

test_job:
  stage: test
  script:
    - ansible-lint playbook.yml
    - ansible-playbook --check playbook.yml

  tags:
    - ansible

deploy_job:
  stage: deploy
  script:
    - ansible-playbook playbook.yml

  tags:
    - ansible
```

Рис. 4.25. Приклад файла gitlab-ci.yml

Jenkins – система з відкритим вихідним кодом, написана на Java, яка забезпечує процес CI/CD.

Рис. 4.26. Емблема Jenkins

Основні переваги Jenkins:
режим роботи відразу в двох і більше середовищах;
підвищена надійність програмного забезпечення, що розгортається;
зменшення помилок, пов'язаних з людським фактором;
зменшення витрат на персонал;
спрощення робочого процесу (немає необхідності наймати дорогу команду досвідчених фахівців, з Jenkins впорається невелика група співробітників без спеціальної кваліфікації).

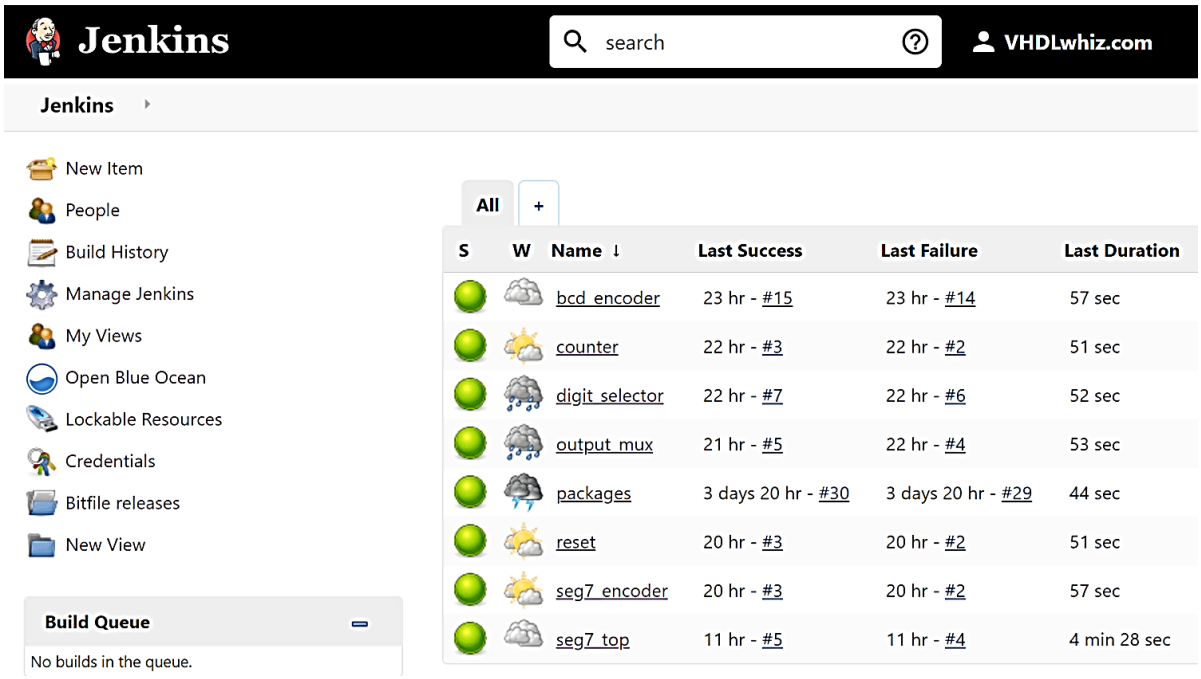


Рис. 4.27. Вигляд графічного інтерфейсу Jenkins

Teamcity – серверне ПЗ від компанії JetBrains, написане на Java, сервер для забезпечення неперервної інтеграції.

Рис. 4.28. Емблема Teamcity

Можливості:

попереднє тестування коду перед коммітом. Запобігає можливість коммітів програмного коду, що містить помилки, порушує нормальну збірку проєкту, шляхом віддаленої збірки змін перед коммітом;

надає можливість здійснювати декілька збирань проєкту одночасно, проводячи тестування на різних платформах та в різному програмному оточенні.

підтримує різні платформи : Java, PHP, .NET, Ruby.

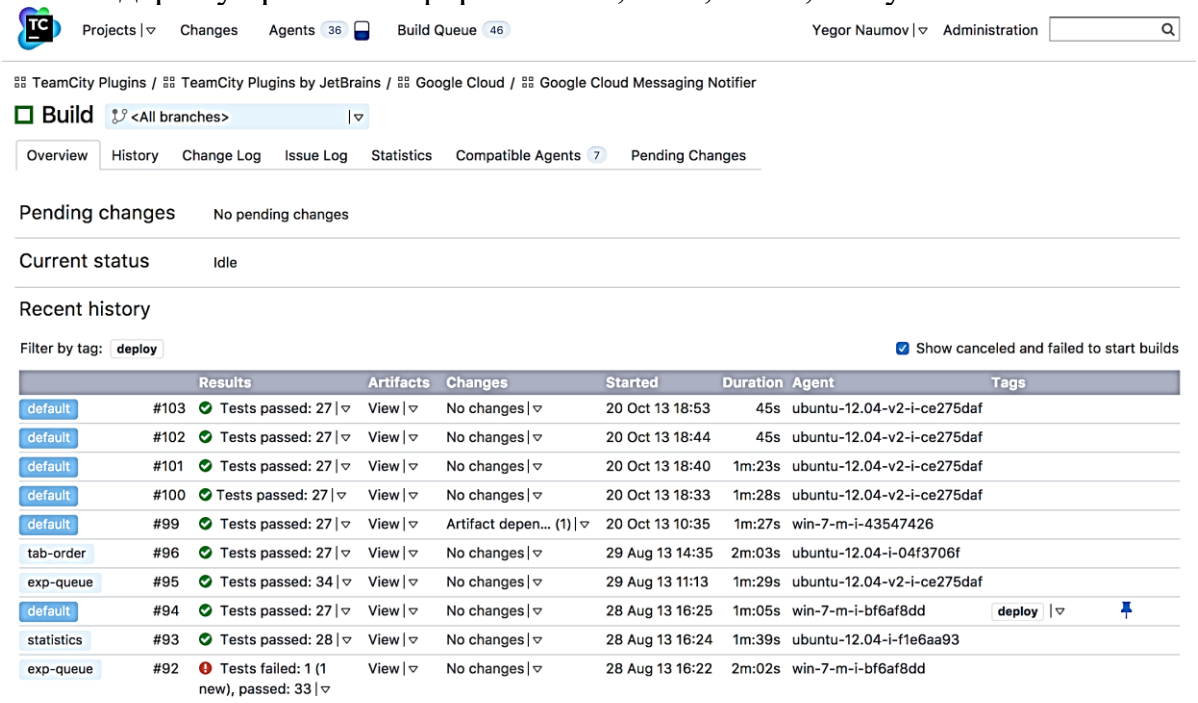


Рис. 4.29. Вигляд графічного інтерфейсу Teamcity

4.5. Робота з Jenkins

Jenkins – це програма з відкритим кодом, написана на Java. Це один з найпопулярніших інструментів безперервної інтеграції (CI), що використовується для побудови та тестування різних видів проєктів.

Вступ до Дженкінса та його особливості

Давайте спочатку зрозуміємо, що таке безперервна інтеграція. CI – одна з найпопулярніших практик розробки додатків останнім часом.

Розробники інтегрують виправлення помилок, розробку нових функцій або інноваційну функціональність у сховище коду. Інструмент CI перевіряє процес інтеграції за допомогою автоматизованої збірки та автоматизованого виконання тесту, щоб виявити проблеми з поточним джерелом програми та забезпечити швидкий зворотний зв'язок.

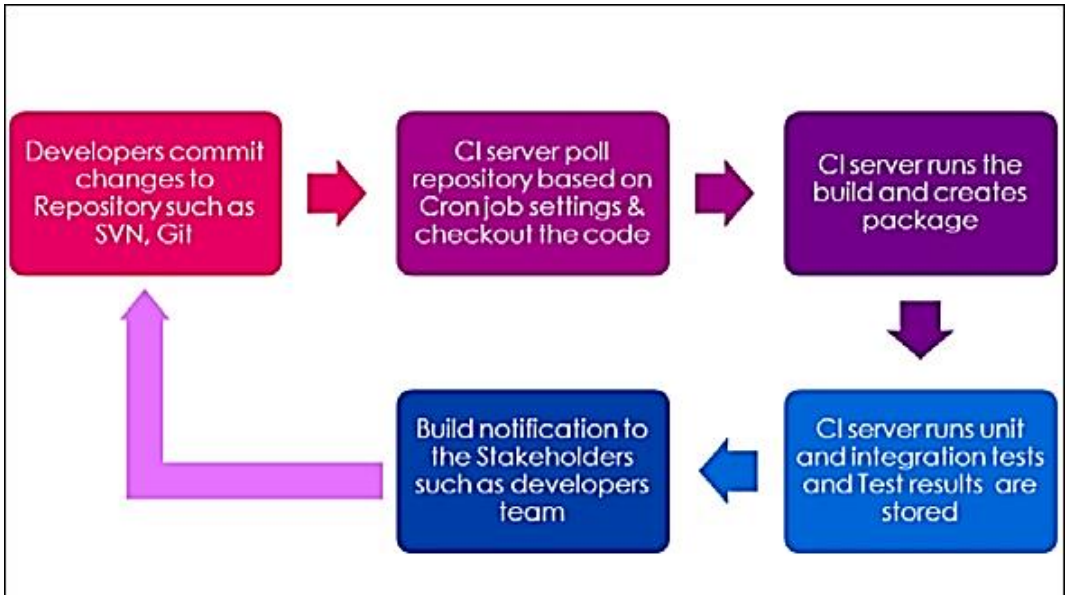


Рис. 4.30. Схема безперервної інтеграції

Jenkins – це простий та зручний інструмент з відкритим кодом, який надає послуги CI для розробки додатків. Дженкінс підтримує такі інструменти SCM, як StarTeam, Subversion, CVS, Git, AccuRev тощо. Дженкінс може будувати проєкти на базі Freestyle, Apache Ant та Apache Maven.

Концепція плагінів робить Jenkins більш привабливим, простим у вивченні та зручним у використанні. Доступні різні категорії плагінів, такі як управління вихідним кодом, пускові пристрої та контролери Slave, тригери побудови, інструменти побудови, сповіщення Build, звіти побудови, інші дії після збірки, інтеграція зовнішніх сайтів/інструментів, плагіни інтерфейсу користувача, автентифікація та управління користувачами, розробка Android, розробка iOS, розробка .NET, розробка Ruby, плагіни бібліотек тощо.

Jenkins визначає інтерфейси або абстрактні класи, які моделюють аспект системи побудови.

Інтерфейси або абстрактні класи визначають домовленість про те, що потрібно реалізувати, Jenkins використовує плагіни для розширення цих реалізацій.

Особливості

Jenkins – один з найпопулярніших серверів CI на ринку. Причини його популярності такі:

проста установка на різні операційні системи;

легкі оновлення – Jenkins має дуже швидкі цикли випуску;

простий та зручний користувальницький інтерфейс;

легко розширюється за допомогою сторонніх плагінів – понад 400 плагінів;

простота налаштування середовища в інтерфейсі користувача. Також можливо налаштувати інтерфейс користувача на основі уподобань;

архітектура ведучого-підлеглого підтримує розподілені збірки для зменшення навантаження на сервер CI;

Jenkins доступний з тестовим вузлом, побудованим навколо JUnit; результати тестів доступні у графічній та табличній формах;

побудуйте планування на основі виразу cron (щоб дізнатися більше про cron, відвідайте http://en.wikipedia.org/wiki/Cron);

виконання команд оболонки та Windows у кроках попередньої збірки;

підтримка сповіщень, пов'язана зі статусом складання.

Підготовка до встановлення Jenkins на Windows та CentOS

1. Зайдіть на https://jenkins-ci.org/. Знайдіть розділ "Завантажити Jenkins" на домашній сторінці веб-сайту Jenkins.

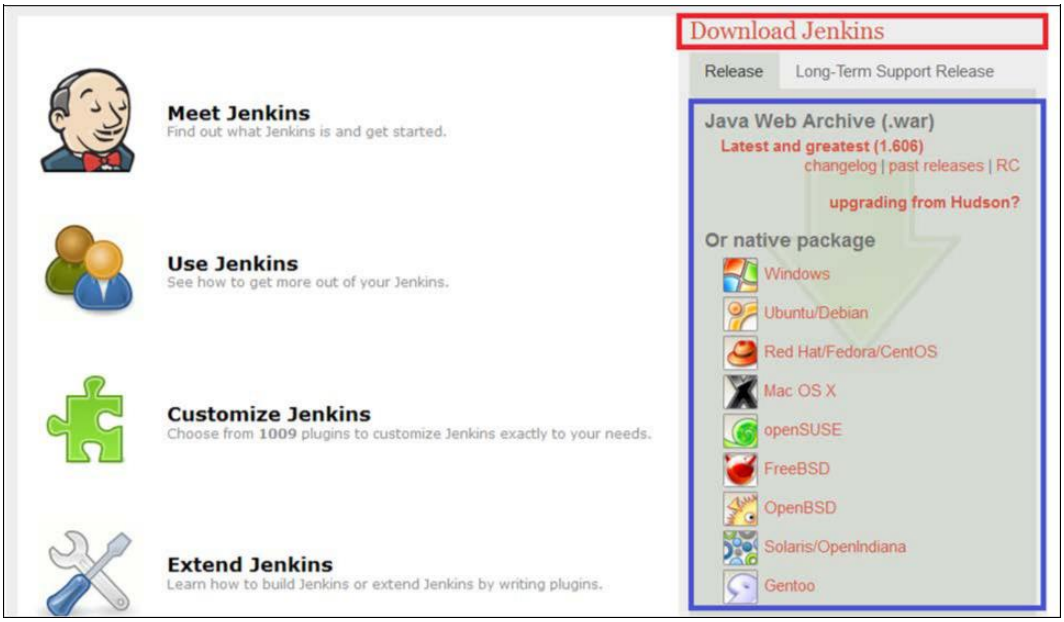


Рис. 4.31. Домашня сторінка вебсайту Jenkins

2. Завантажте інсталяційний файл або власні пакунки на основі вашої операційної системи.

Для запуску Дженкінса потрібна інсталяція Java.

3. Встановіть Java на основі вашої операційної системи та встановіть відповідну змінну середовища JAVA_HOME.

Встановлення Дженкінса на Windows

1. Виберіть власний пакет, доступний для Windows. Він завантажить jenkins-1.xxx.zip. У нашому випадку він завантажить jenkins-

1.606.zip. Розпакуйте його, і ви отримаєте файли setup.exe та jenkins-1.606.msi.

2. Клацніть setup.exe і виконайте наступні кроки послідовно. У вікні привітання натисніть Далі:

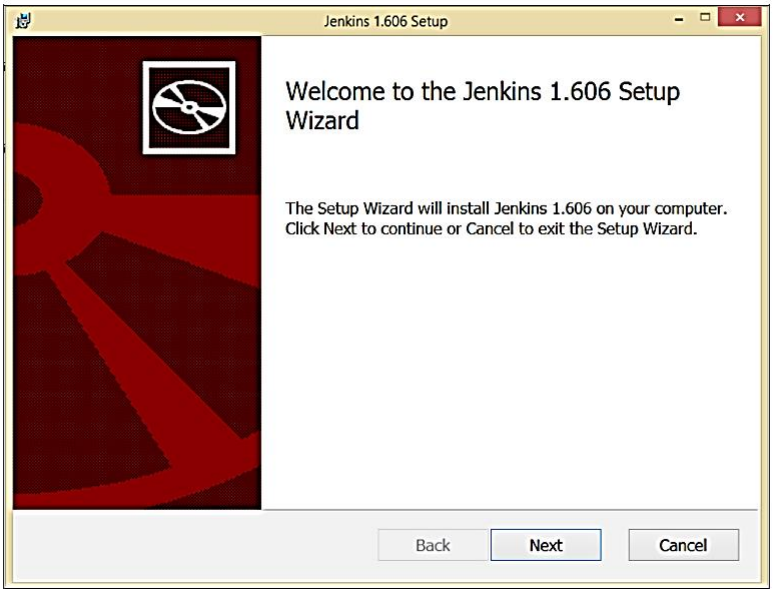


Рис. 4.32. Вікно привітання майстра встановлення Jenkins

2. Виберіть папку призначення та натисніть Далі.

3. Клацніть на Встановити, щоб розпочати встановлення. Зачекайте, поки Майстер встановлення встановить Jenkins.

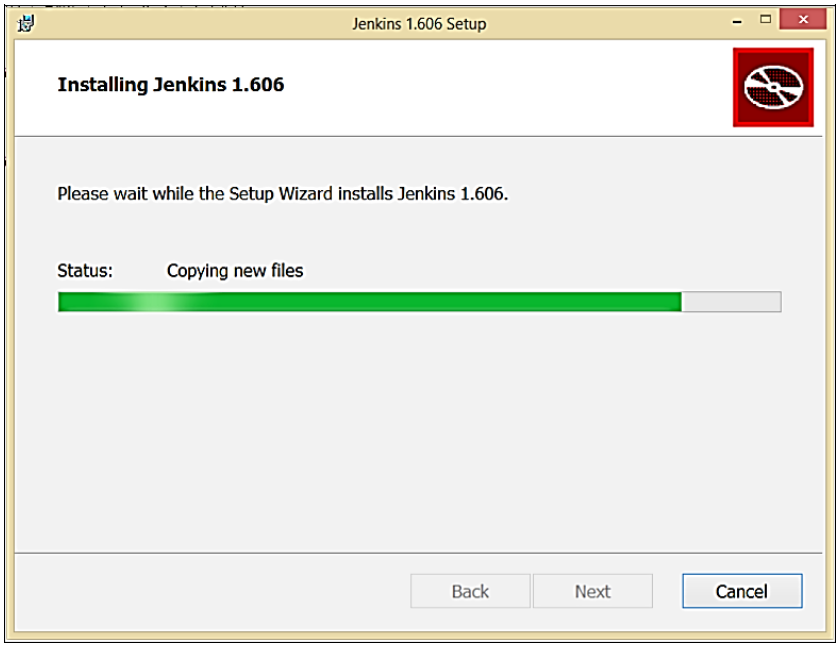


Рис. 4.33. Робота майстра встановлення Jenkins

5. Після завершення інсталяції Jenkins натисніть кнопку Готово (Finish).

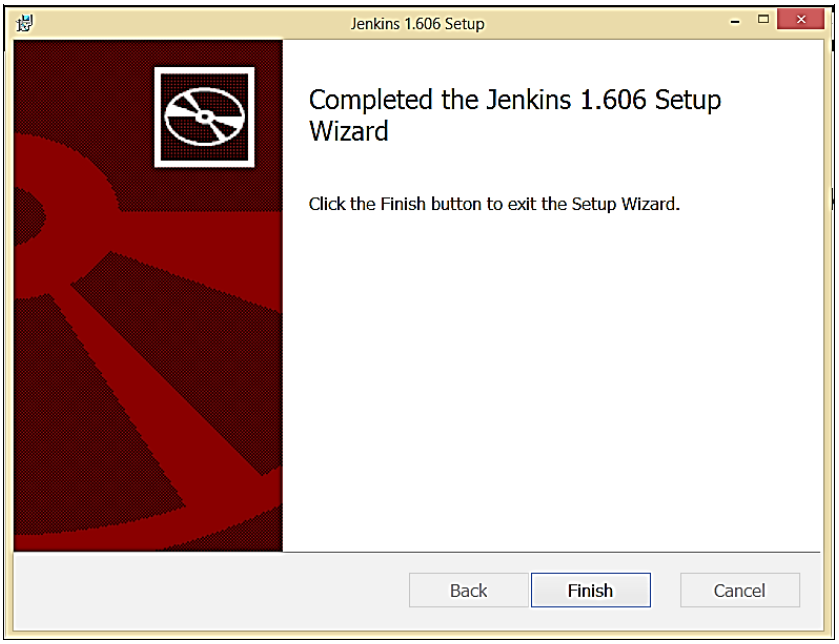


Рис. 4.34. Завершення роботи майстра встановлення Jenkins

6. Перевірте установку Jenkins на EOM з Windows, відкривши URL http: // <ip_address>: 8080 у системі, де ви встановили Jenkins.

Рис. 4.35. Графічний інтерфейс Jenkins

Встановлення Jenkins на CentOS

1. Щоб встановити Jenkins на CentOS, завантажте репозиторій Jenkins у свою локальну систему за адресою /etc/yum.repos.d/ та імпортуйте ключ.

2. Використовуйте wget -O /etc/yum.repos.d/jenkins.repo http://pkg. команда jenkins-ci.org/redhat/jenkins.repo для завантаження репозиторія.

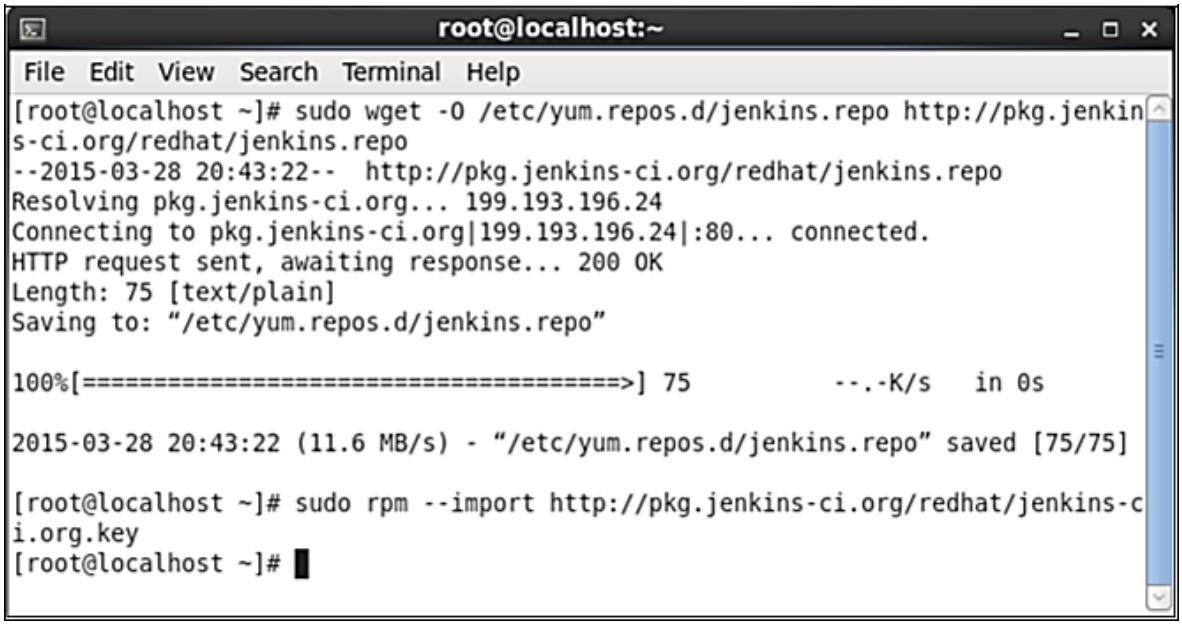


Рис. 4.36. Імпорт репозиторію Jenkins

3. Тепер запустіть yum install Jenkins; він визначить залежності та запропонує встановити ПЗ.

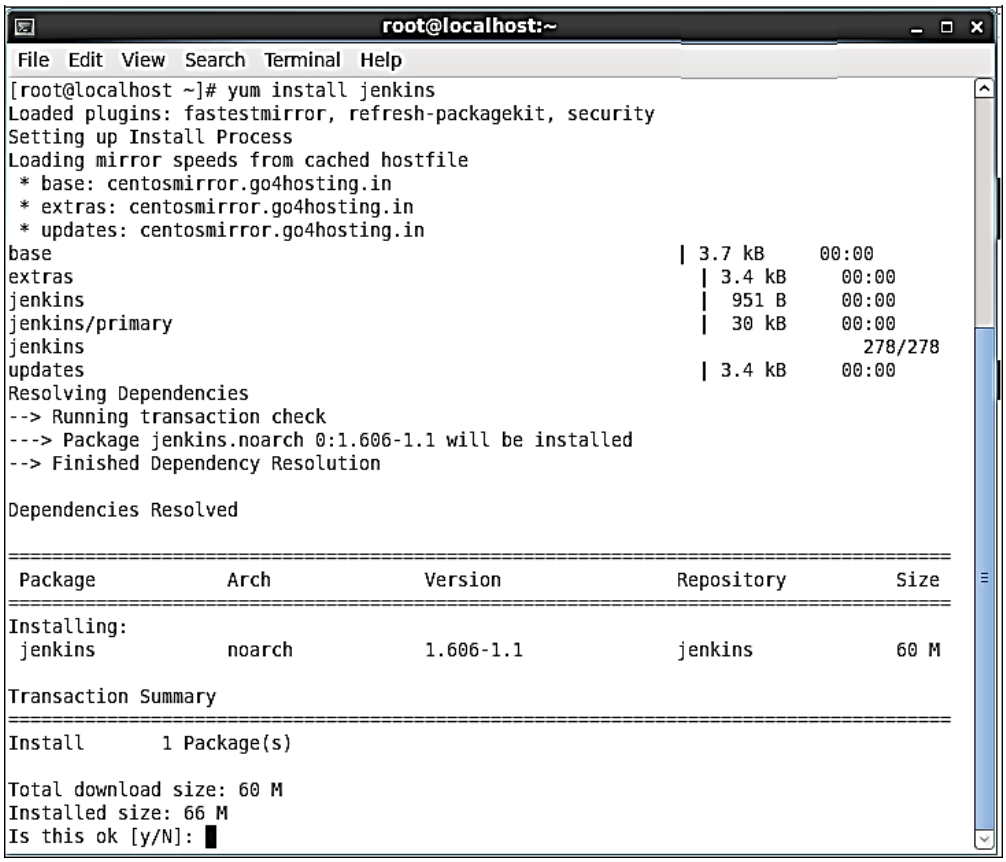


Рис. 4.37. Запуск інсталяції Jenkins

4. Відповідайте y, і він завантажить необхідний пакет для встановлення Jenkins на CentOS. Перевірте статус Jenkins, ввівши команду служби jenkins status. Спочатку це він буде зупинений. Запустіть Jenkins, виконавши команду service jenkins start в терміналі.

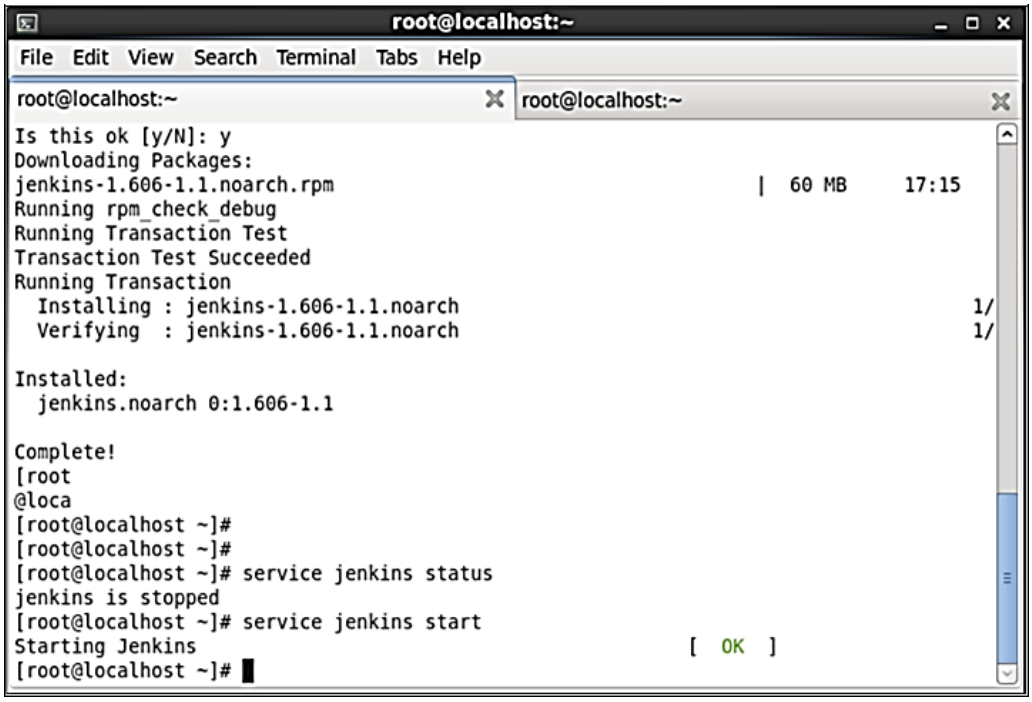


Рис. 4.38. Перевірка статусу Jenkins

5. Перевірте установку Jenkins на машині CentOS, відкривши URL-адресу http: // <ip_address>: 8080 у системі, де ви встановили Jenkins.

Рис. 4.39. Графічний інтерфейс Jenkins

Встановлення Jenkins як вебдодатку

1. Завантажте вебархів Java (.war) (останній і найвищий (1.606)) з вебсайту http://jenkins-ci.org/.

2. Скопіюйте jenkins.war у свою віртуальну або фізичну машину. Відкрийте командний рядок або термінал на основі операційної системи. У нашому випадку ми скопіюємо його в каталог віртуальної машини CentOS.

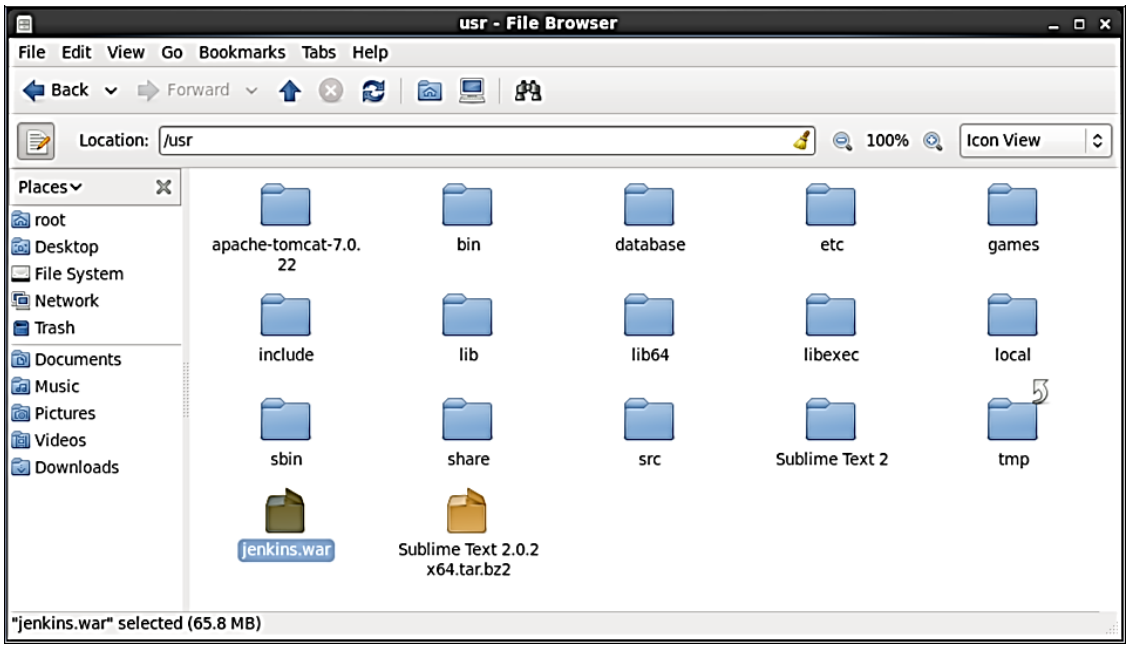


Рис. 4.40. Завантаження архіву Jenkins в CentOS

3. Відкрийте командний рядок і виконайте команду java –jar Jenkins.war. Перевірте встановлення Jenkins у системі, відкривши URL-адресу http: // <ip_address>: 8080 у системі, де ви встановили Jenkins.

Ознайомлення з панеллю приладів Jenkins

1. На інформаційній панелі Jenkins натисніть Створити Нові робочі місця або Новий елемент, щоб створити проєкти, засновані на Freestyle або Maven для CI.

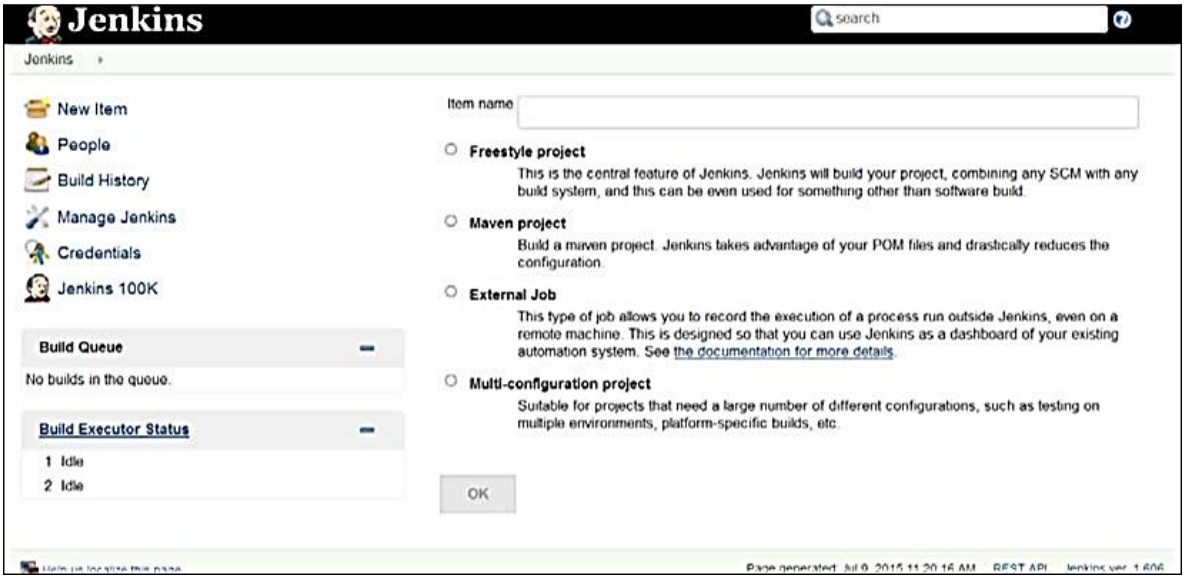


Рис. 4.41. Панель приладів Jenkins

2. Щоб перевірити властивості системи, відвідайте http://<ip_address>: 8080/systeminfo або клацніть на Manage Jenkins, а потім клацніть на System Information (Інформація про систему), щоб отримати інформацію про навколишнє середовище для усунення несправностей.

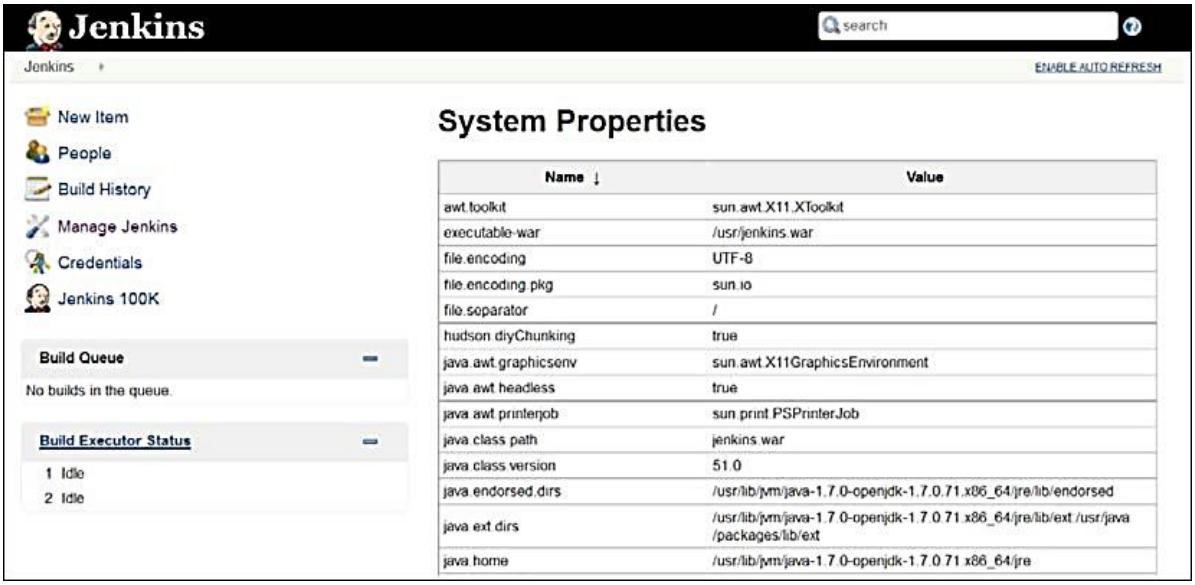


Рис. 4.42. Властивості системи Jenkins

Як змінити налаштування конфігурації в Jenkins

1. Клацніть посилання Manage Jenkins на інформаційній панелі, щоб налаштувати систему, безпеку, керувати плагінами, підлеглими вузлами, обліковими даними тощо.

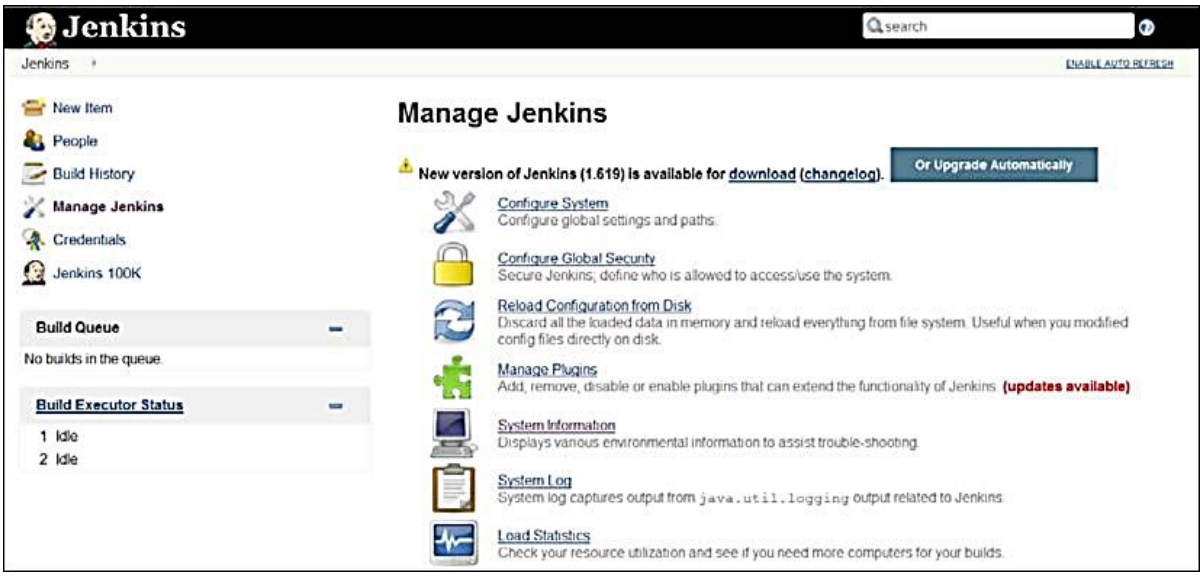


Рис. 4.43. Налаштування конфігурації Jenkins

2. Клацніть на посилання Configure System, щоб налаштувати інформацію про Java, Ant, Maven та інші сторонні продукти.

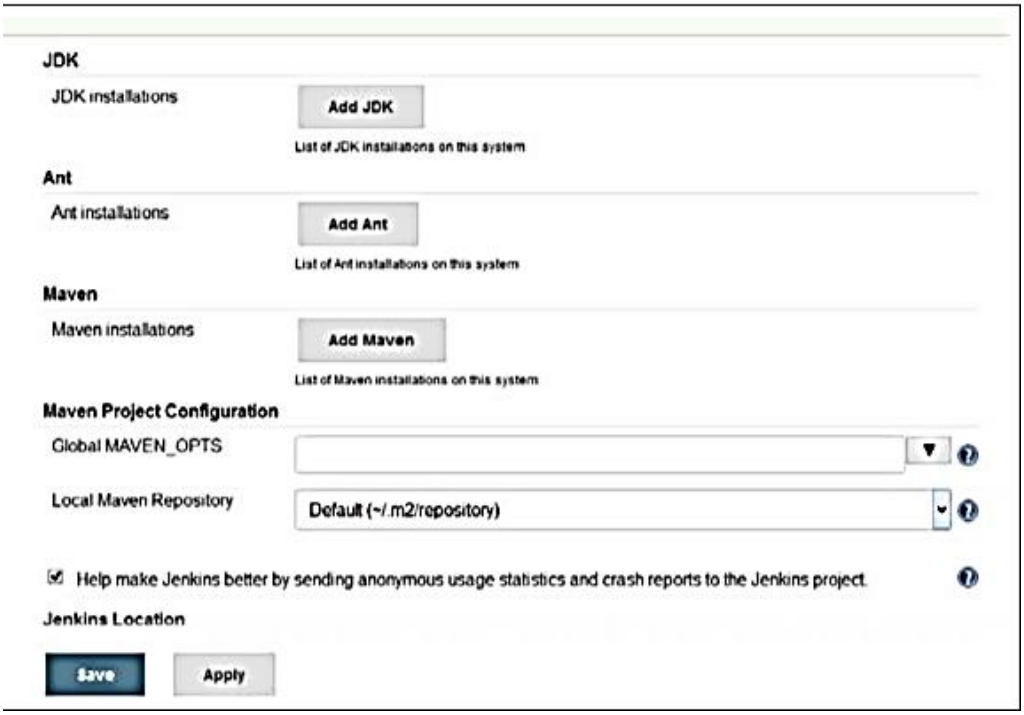


Рис. 4.44. Налаштування конфігурації Jenkins

3. Jenkins використовує Groovy як мову сценаріїв. Щоб виконати довільний сценарій для адміністрування/усунення несправностей/діагностики на інформаційній панелі Jenkins, перейдіть до посилання Manage Jenkins на інформаційній панелі, натисніть Script Console і запустіть println (Jenkins.instance.pluginManager.plugins).

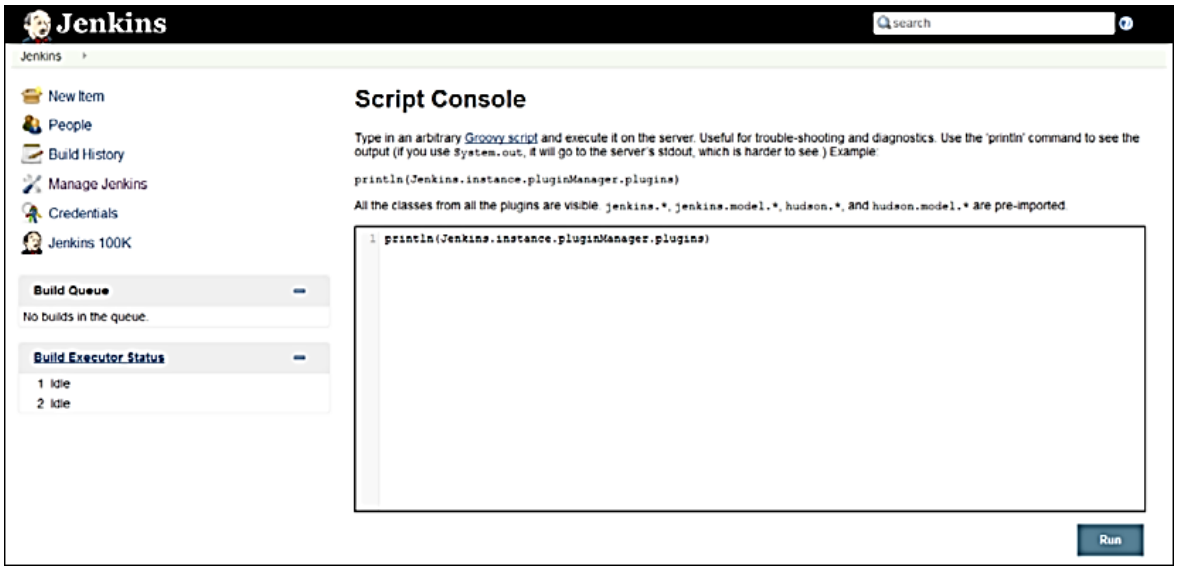


Рис. 4.45. Виконання сценарію в Jenkins

4. Щоб перевірити системний журнал, перейдіть за посиланням Manage Jenkins на інформаційній панелі та натисніть посилання System Log або відвідайте http: //localhost:8080/log/all.

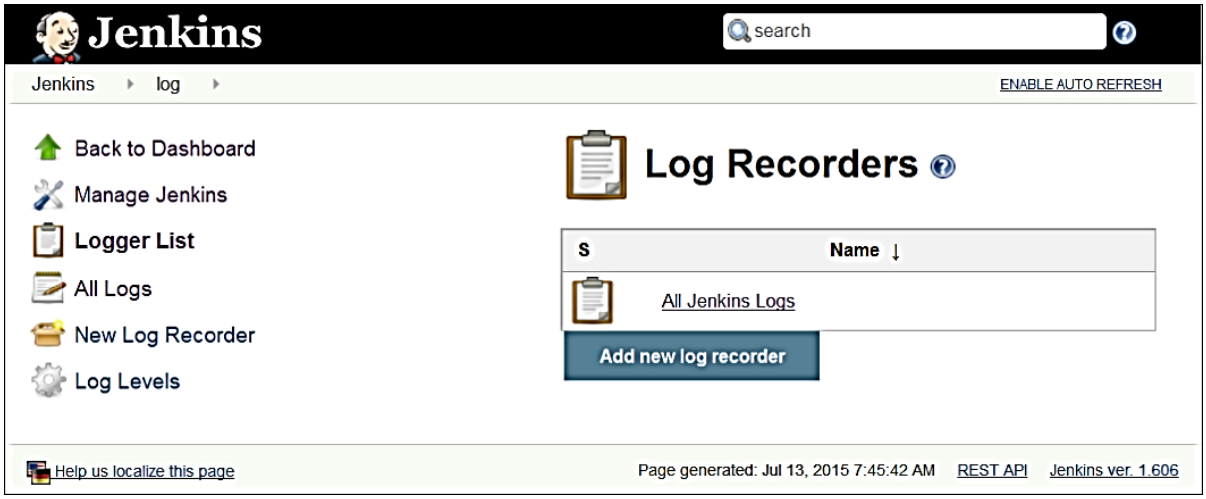


Рис. 4.43. Перевірка системного журналу в Jenkins

5. Щоб отримати додаткову інформацію про сторонні бібліотеки – інформацію про версію та ліцензію в Jenkins, перейдіть за посиланням Manage Jenkins на інформаційній панелі та натисніть на посилання About Jenkins.

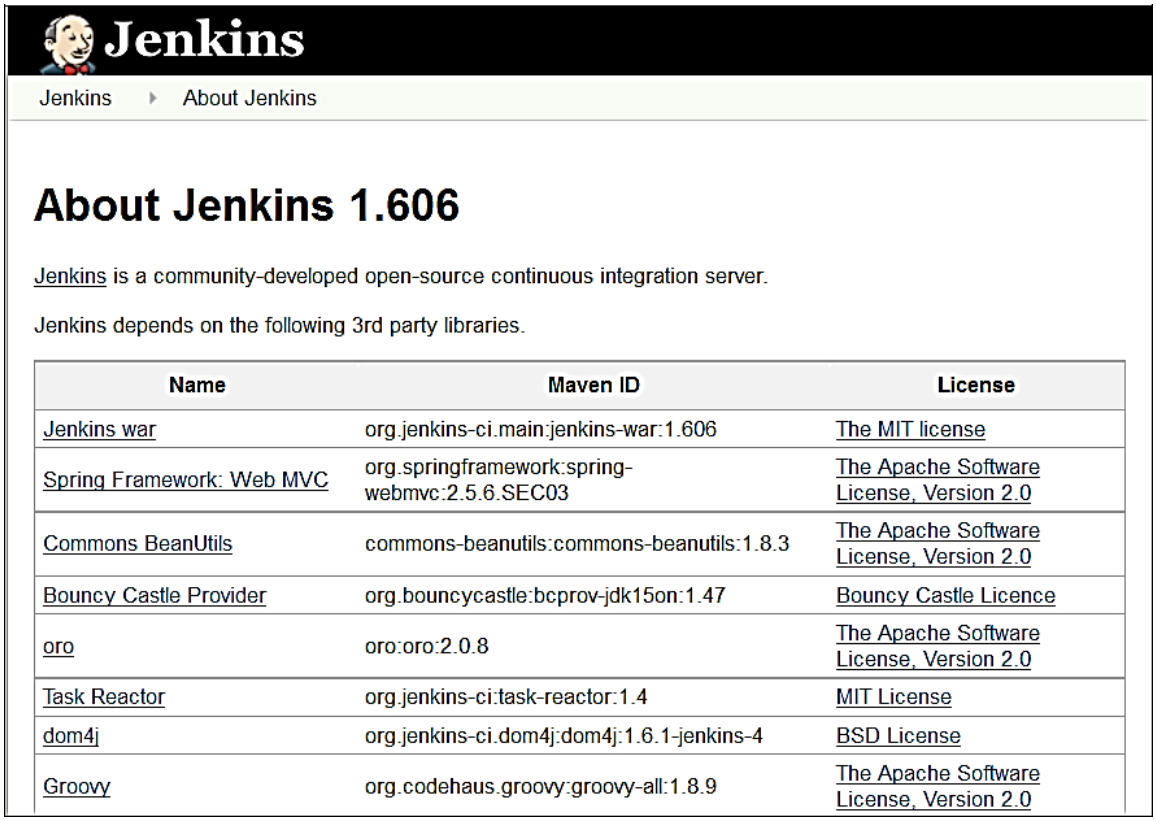


Рис. 4.47. Отримання додаткової інформації про Jenkins

Що таке конвеєр розгортання?

Життєвий цикл розробки додатків є традиційно тривалим і конфігурованим процесом. Крім того, це вимагає ефективної співпраці між командами розробників та операцій. **Конвеєр розгортання** (deployment pipeline) – це демонстрація автоматизації, що бере участь у життєвому циклі розробки додатків, яка містить автоматичне виконання збірки та виконання тесту, повідомлення зацікавленій стороні та розгортання в різних середовищах виконання. Фактично конвеєр розгортання являє собою поєднання CI та безперервної доставки, а отже, є частиною практики DevOps. На наступній схемі зображено процес конвеєру розгортання:

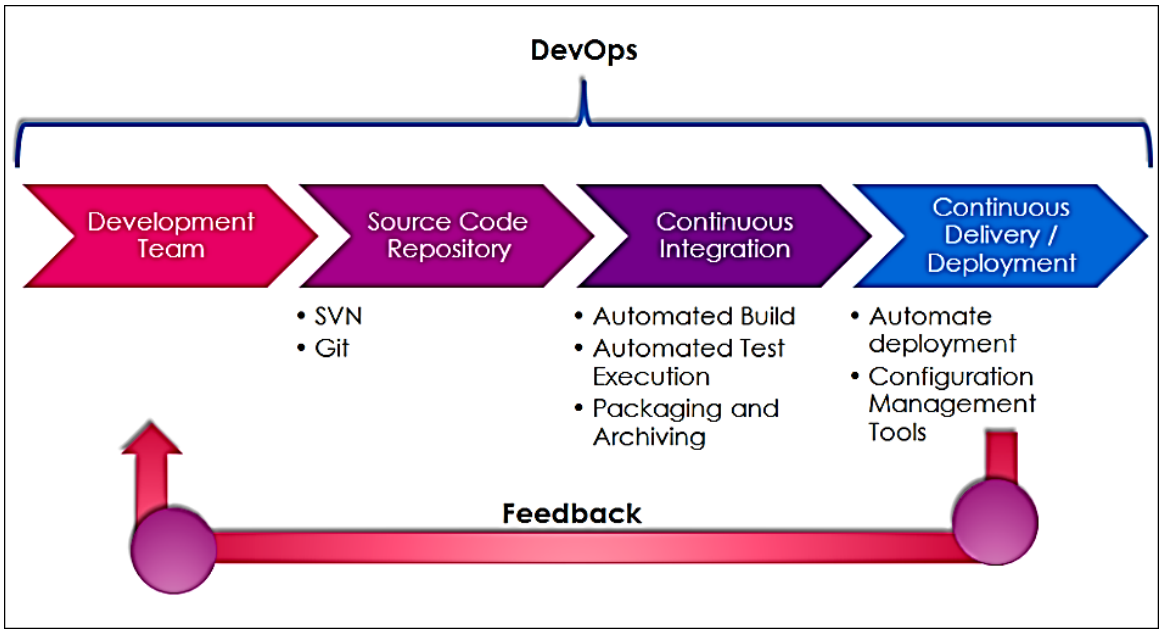


Рис. 4.48. Процес конвеєру розгортання

Члени команди розробників перевіряють код у репозиторію вихідного коду. Продукти CI, такі як Jenkins, налаштовані на опитування змін із сховища коду. Зміни в сховищі завантажуються в локальну робочу область, і Jenkins запускає автоматизований процес збірки, який проходить за допомогою Ant або Maven. Автоматизоване виконання тестів або модульне тестування, статичний аналіз коду, звітування та повідомлення про успішний або невдалий процес побудови також є частиною процесу CI.

Як тільки збірка буде успішною, її можна буде розгорнути в різних середовищах виконання, таких як тестування, попереднє виготовлення, виготовлення тощо. Розгортання war-файла з точки зору програми JEE, як правило, є завершальним етапом у конвеєрі розгортання.

Однією з найбільших переваг конвеєра розгортання є швидший цикл зворотного зв'язку. Виявлення проблем у програмному забезпеченні на

ранніх стадіях та відсутність залежності від ручної збірки роблять весь цей наскрізний процес більш ефективним.

Встановлення та налаштування репозиторію коду та інструментів побудови. Огляд збірки в Jenkins та вимоги до неї

Для пояснення безперервної інтеграції ми будемо використовувати репозиторій коду, встановлений на фізичній машині або ноутбуці, тоді як Дженкінс встановлюється на віртуальній машині, як це пропонується різними способами в попередньому матеріалі. На наступному рисунку зображено налаштування середовища виконання:

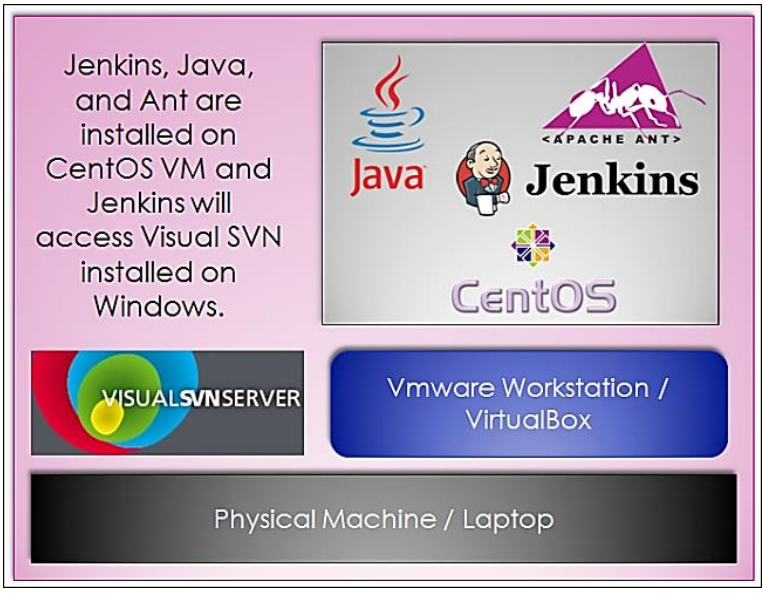


Рис. 4.49. Налаштування середовища виконання

У попередньому матеріалі ми побачили, що посилання “Manage Jenkins” на інформаційній панелі використовується для налаштування системи. Клацніть на посилання Налаштувати систему, щоб налаштувати Java, Ant, Maven та іншу інформацію, пов’язану із сторонніми продуктами. Ми можемо створити віртуальну машину за допомогою Virtual box або робочої станції VMware. Нам потрібно встановити все необхідне програмне забезпечення, щоб забезпечити середовище виконання для постійної інтеграції. Ми вважаємо, що Java вже встановлена в системі.

Встановлення Java та налаштування змінних середовища

Якщо Java ще не встановлено в системі, ви можете встановити її наступним чином, знайдіть пакунки, пов’язані з Java, доступні у сховищі CentOS, і знайдіть відповідний пакет для встановлення:

```
[root@localhost ~]# yum search java
Loaded plugins: fastestmirror, refresh-packagekit, security
.
```

.
ant-javamail.x86_64 : Optional javamail tasks for ant

eclipse-mylyn-java.x86_64 : Mylyn Bridge: Java Development
.
.
java-1.5.0-gcj.x86_64 : JPackage runtime compatibility layer for GCJ
java-1.5.0-gcj-devel.x86_64 : JPackage development compatibility layer for GCJ
java-1.5.0-gcj-javadoc.x86_64 : API documentation for libgcj
java-1.6.0-openjdk.x86_64 : OpenJDK Runtime Environment
java-1.6.0-openjdk-devel.x86_64 : OpenJDK Development Environment
java-1.6.0-openjdk-javadoc.x86_64 : OpenJDK API Documentation
java-1.7.0-openjdk.x86_64 : OpenJDK Runtime Environment
jcommon-serializer.x86_64 : JFree Java General Serialization Framework
.
.
Install the identified package java-1.7.0-openjdk.x86_64
[root@localhost ~]# yum install java-1.7.0-openjdk.x86_64
Loaded plugins: fastestmirror, refresh-packagekit, security
No such command: in. Please use /usr/bin/yum –help

Тепер встановіть пакет Java, доступний у локальних сховищах, виконавши команду yum install наступним чином:

[root@localhost ~]# yum install java-1.7.0-openjdk.x86_64
Loaded plugins: fastestmirror, refresh-packagekit,
security Loading mirror speeds from cached hostfile
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package java-1.7.0-openjdk.x86_64 1:1.7.0.3-2.1.el6.7
will be installed
--> Finished Dependency Resolution
Dependencies Resolved
.
Install 1 Package(s)
Total download size: 25 M
Installed size: 89 M
Is this ok [y/N]: y
Downloading Packages:
java-1.7.0-openjdk-1.7.0.3-2.1.el6.7.x86_64.rpm
| 25MB 00:00
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction

Installing : 1:java-1.7.0-openjdk-1.7.0.3-2.1.el6.7.x86_64 1/1
Verifying : 1:java-1.7.0-openjdk-1.7.0.3-2.1.el6.7.x86_64 1/1
Installed:
java-1.7.0-openjdk.x86_64 1:1.7.0.3-2.1.el6.7
Java успішно встановлена з локального сховища.

Налаштування змінних середовища

Нижче наведено кроки для налаштування змінних середовища:

1. Встановіть змінні JAVA_HOME і JRE_HOME;
2. Перейдіть до / root;
3. Натисніть Ctrl + H, щоб переглянути список прихованих файлів;
4. Знайдіть .bash_profile та відредагуйте його, додавши шлях до Java, як показано на наступному скріншоті:

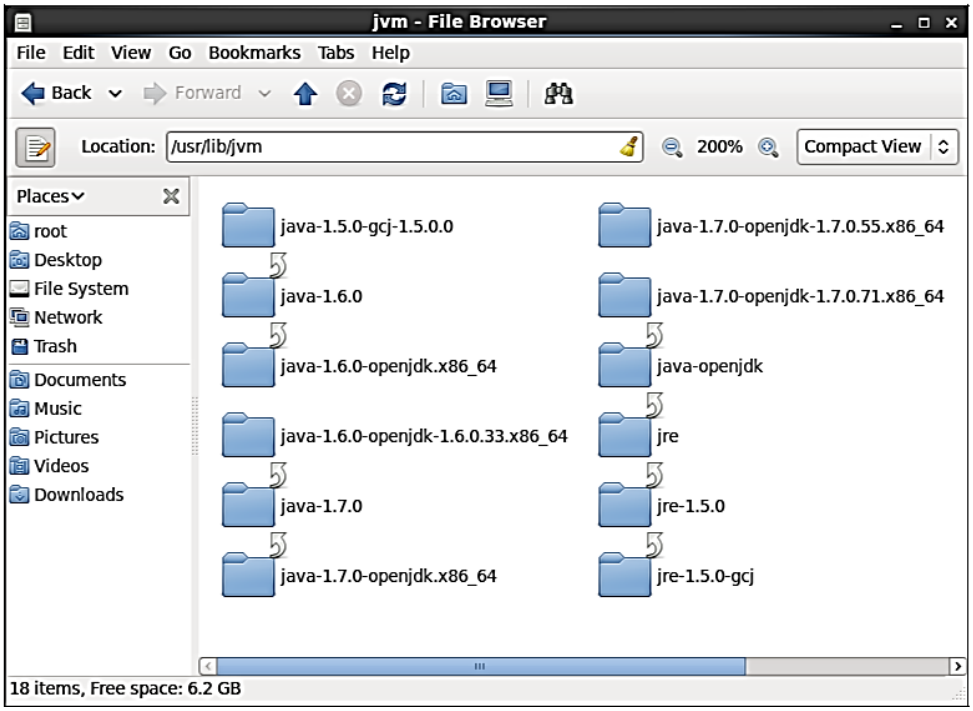


Рис. 4.50. Шлях до Java

Встановлення та налаштування Git

Git – це безкоштовна система розподіленого контролю версій з відкритим кодом. Ми спробуємо встановити та налаштувати Git:

1. Відкрийте термінал в системі на базі CentOS і виконайте в терміналі команду yum install git;
2. Після успішного встановлення перевірте версію за допомогою команди git --version;
3. Надайте інформацію про користувача за допомогою команди git config, щоб повідомлення комітів були сформовані з доданою правильною інформацією;

4. Вкажіть ім’я та адресу електронної пошти для вбудовування у коміти;

5. Щоб створити середовище робочої області, створіть каталог з назвою git у домашньому каталозі, а потім створіть підкаталог усередині того, що називається розробкою. Використовуйте mkdir -p ~ / git / development; cd ~ / git / розробка в терміналі;

6. Скопіюйте каталог AntExample1 у папку розробника;

7. Перетворіть існуючий проєкт у середовище робочої області за допомогою команди git init;

8. Після ініціалізації сховища додайте файли та папки;

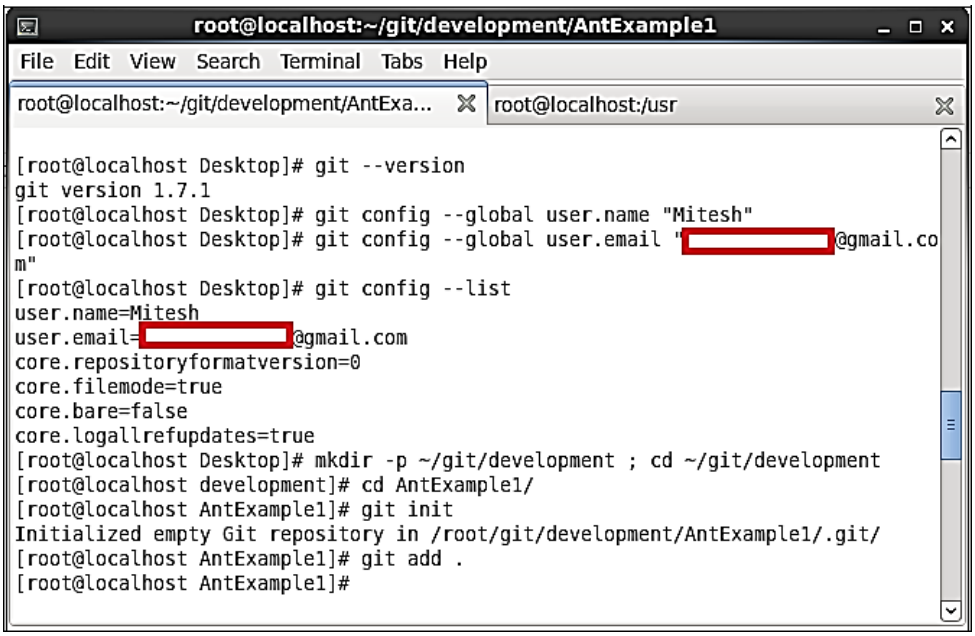


Рис. 4.51. Встановлення Git

9. Комітувати, виконавши git commit -m “Initial Commit” –a;

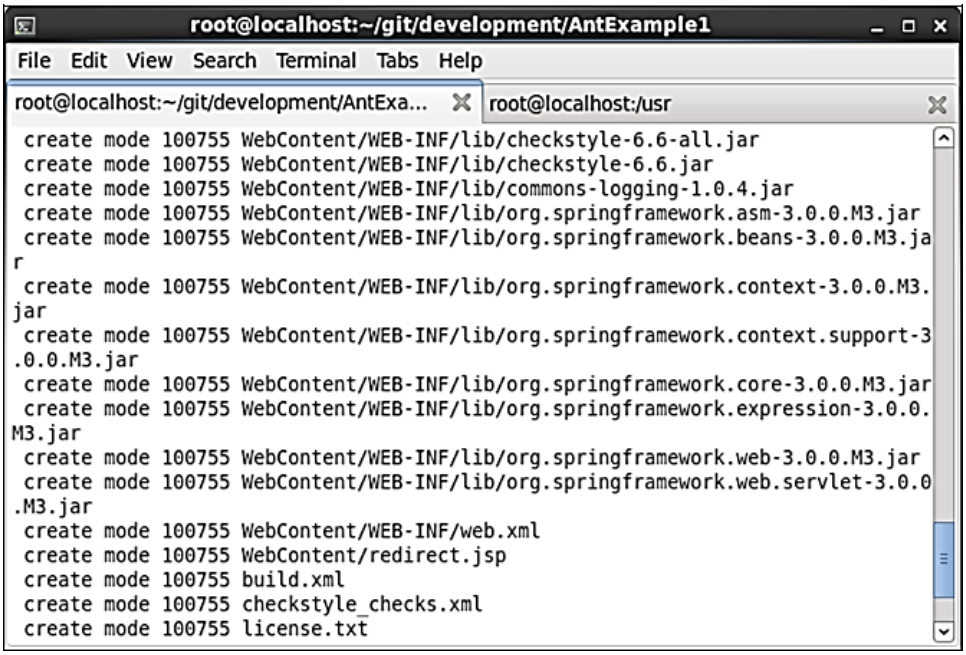


Рис. 4.52. Комітування в Git

10. Перевірте репозиторій Git;

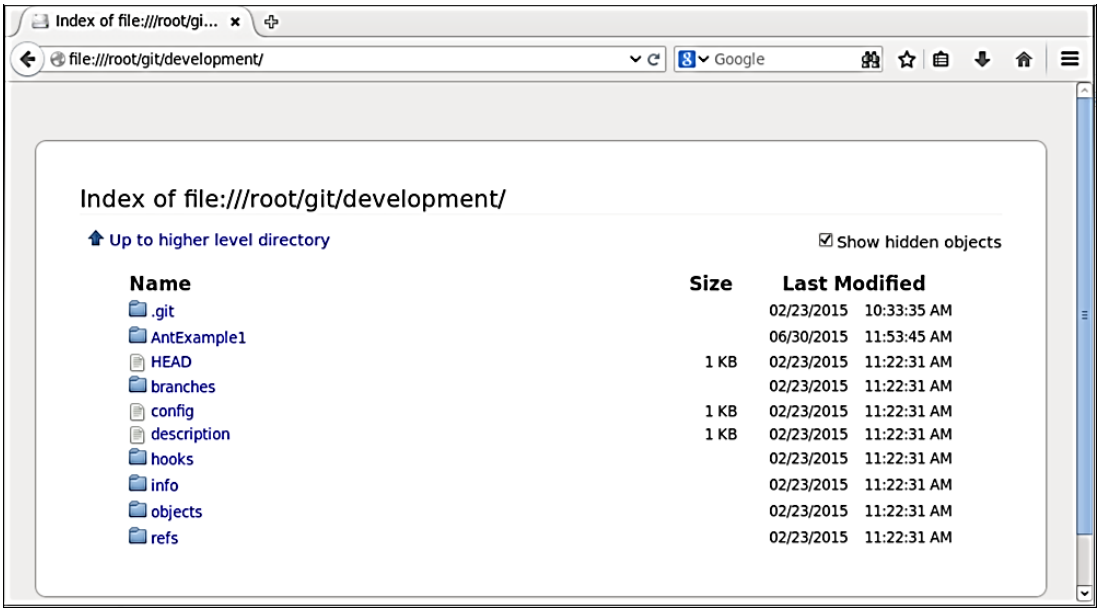


Рис. 4.53. Перевірка репозиторію в Git

11. Перевірте проєкт у репозиторію Git.

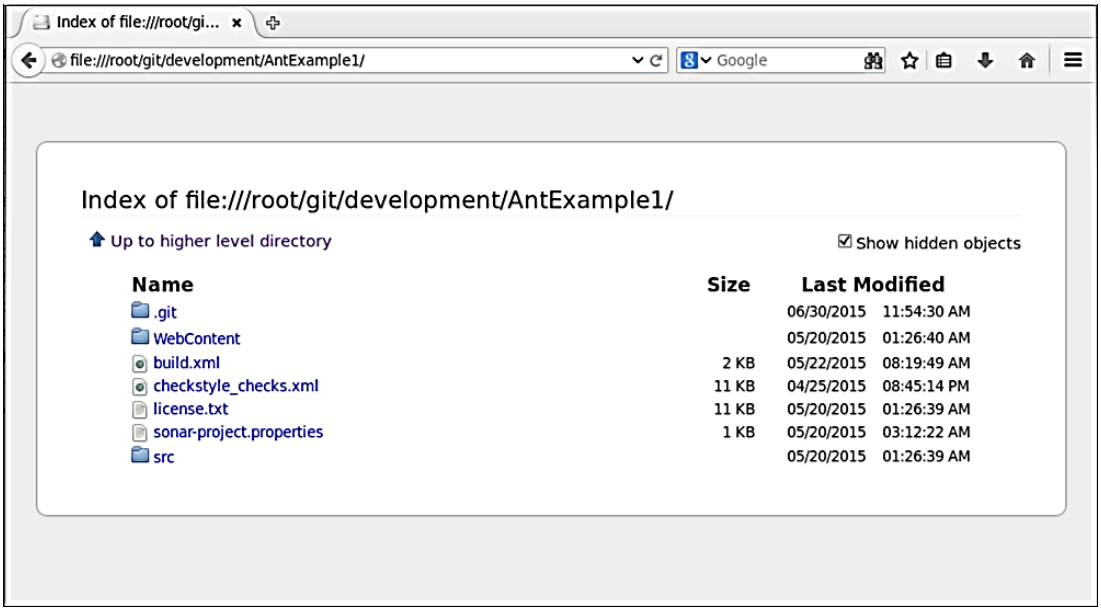


Рис. 4.54. Перевірка проєкту в репозиторію Git

Створення нового робочого місця в Jenkins за допомогою Git

1. На інформаційній панелі Jenkins натисніть Manage Jenkins і виберіть Manage Plugins. Клацніть на вкладку Available та напишіть плагін github у вікні пошуку.
2. Поставте прапорець і натисніть кнопку **Download now and install after restart**.
3. Перезапустіть Jenkins.

Рис. 4.55. Додавання GitHub в Jenkins

4. Створіть новий проєкт Freestyle. Введіть назву проєкта та натисніть кнопку OK.

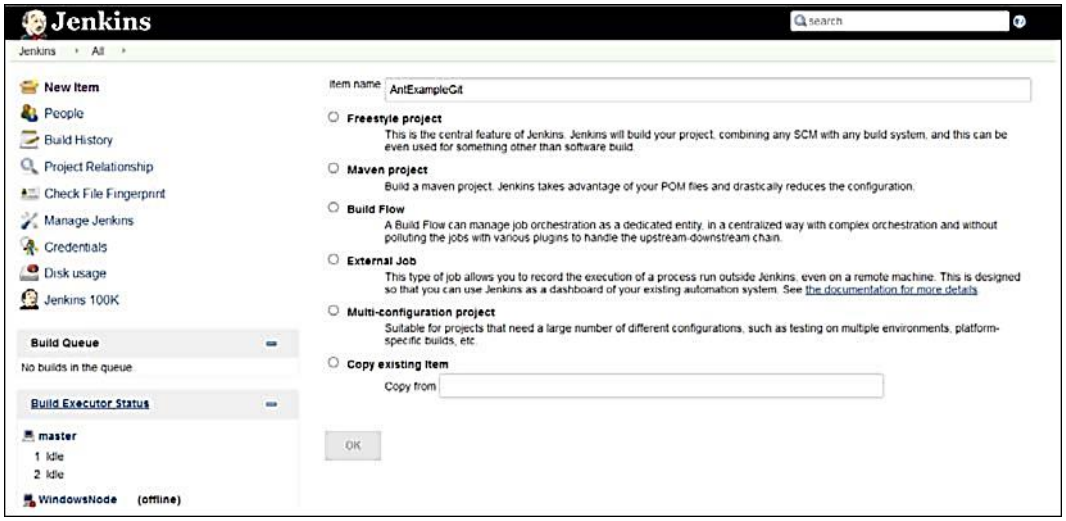


Рис. 4.56. Створення нового Freestyle проекту в Jenkins

5. Налаштуйте Git у розділі **Source Code Management**.

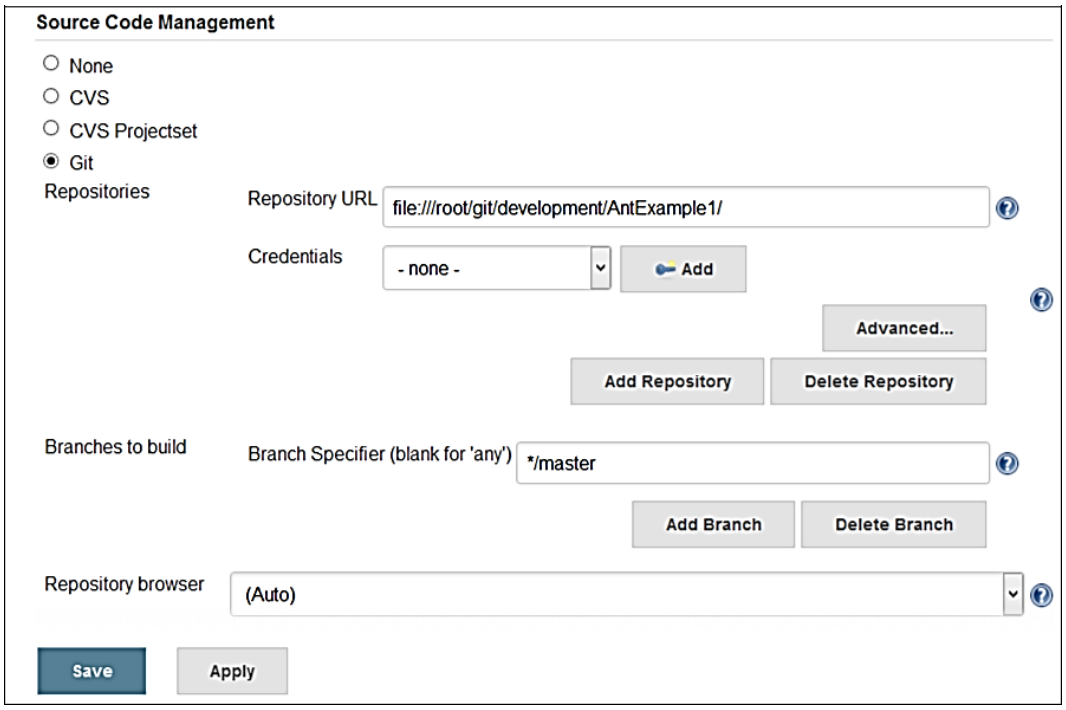


Рис. 4.57. Налаштування Git у розділі Source Code Management

6. Додайте крок побудови Invoke Ant, натиснувши **Add build step**.

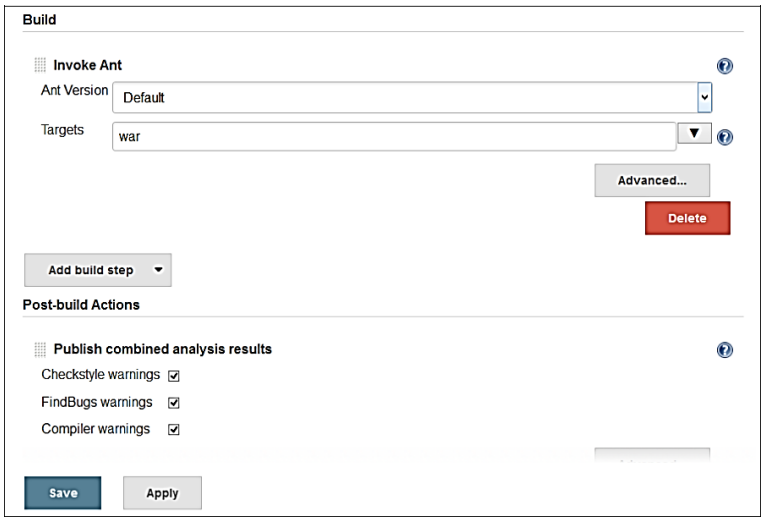


Рис. 4.58. Додавання Invoke Ant

7. Виконайте побудову.

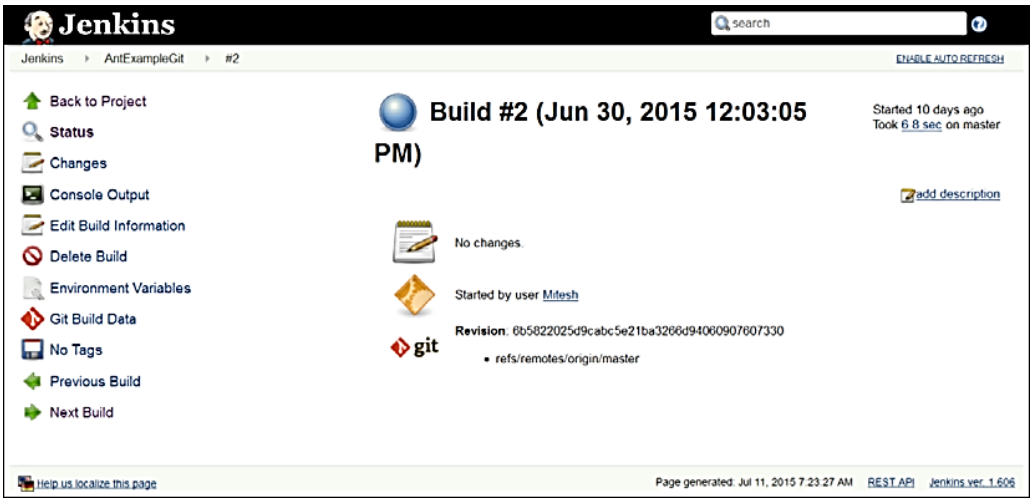


Рис. 4.59. Виконана побудова проекту

8. Клацніть на Console Output, щоб побачити хід збірки.

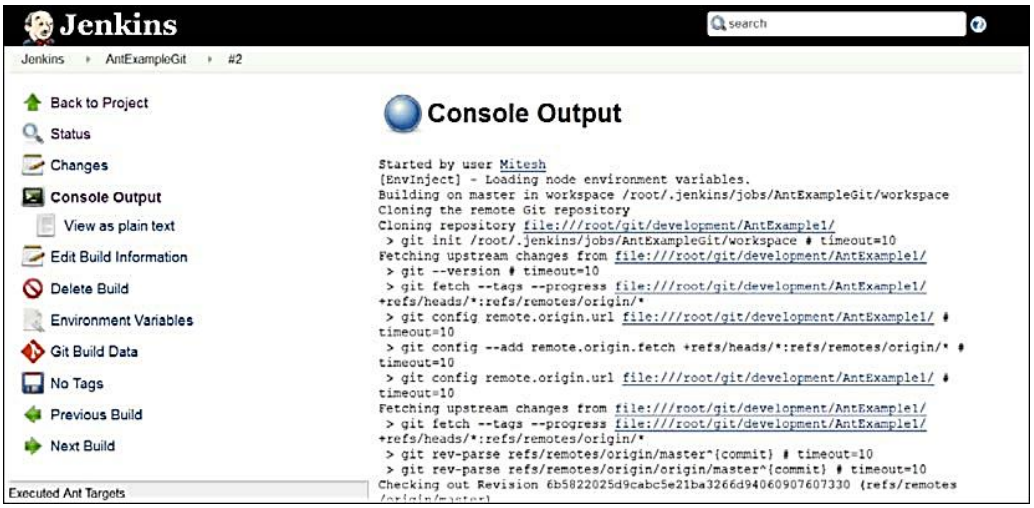


Рис. 4.60. Перегляд ходу збірки

9. Після того, як збірка була успішною, перевірте робочий простір у завданні збірки.

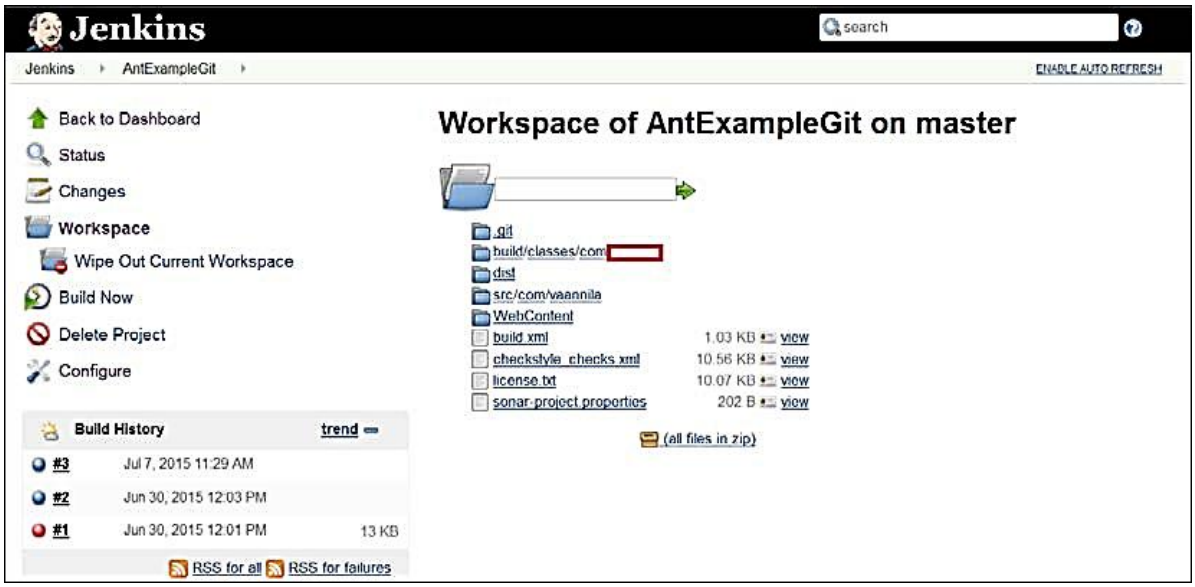


Рис. 4.61. Створений робочий простір

Контрольні запитання

1. Що таке система контролю версій?
2. Які типи систем контролю версій ви знаєте?
3. Охарактеризуйте локальну систему контролю версій.
4. Охарактеризуйте централізовану систему контролю версій.
5. Охарактеризуйте децентралізовану систему контролю версій.
6. Що таке Git?
7. Що таке дельта файла в системі контролю версій?
8. Що таке snapshot файла в системі контролю версій?
9. В яких станах може бути файл в Git?
10. Що таке GitLab?
11. Основні команди для роботи з GitLab.
12. Поясніть поняття постійної інтеграції ПЗ.
13. Поясніть поняття неперервного розгортання ПЗ.
14. Які ви знаєте інструменти для реалізації CI/CD?
15. Що таке Jenkins?
16. Як встановити Jenkins на Windows?
17. Як встановити Jenkins на Linux?
18. Як встановити Jenkins як вебдодаток?
19. Що таке конвеєр розгортання в Jenkins?

СПИСОК ЛІТЕРАТУРИ

1. Бейер Б., Джоунс К., Петофф Д., Мёрфи Н.Р. Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2019.

2. Брент Ластер, “Jenkins 2. Приступаем к работе”, ДМК Пресс, 2019 год, 652 с.

3. Дженифер Девис, Кетрин Дениелс, Философия DevOps. К.: Питер Пресс, 2019 год, 416 с.

4. Джин Ким, Патрик Дебуа, Джон Уиллис, Джез Хамбл, Руководство по DevOps, К.: 2018 год, 512 с.

5. Дэвис Д., Дэниелс К. Философия DevOps. Искусство управления IT. – СПб.: Питер, 2017 год.

6. Евгений Брикман “Terraform. Инфраструктура на уровне кода”, Питер, 2020 год, 368 с.

7. Э. Моуэт “Использование Docker” ДМК Пресс, 2017 год, 354 с.

8. Колиснеченко Д.Н., Аллен П.В., LINUX Полное руководство. К.: Наука и Техника, 2014 год, 781 с.

9. Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2019.

10. Лукша М. Kubernetes в действии Action. — М.: ДМК-Пресс, 2019.

11. Михаель Кофлер Linux. Установка, настройка администрирование. К. Питер, 2014 год, 768 с.

12. Нейгард М. Release it! Проектирование и дизайн ПО для тех, кому не все равно. — СПб.: Питер, 2016.

13. Рене Мозер, Лорин Хоштейн “Запускаем Ansimble”, ДМК Пресс, 2018 год, 382 стр.

14. Хамбл Д., Уиллис Д., Дебуа П., Ким Д. Руководство по DevOps. Как добиться гибкости, надежности и безопасности мирового уровня в технологических компаниях. — М.: Манн, Иванов и Фербер, 2018.

15. Чакон С., Штрауб Б., Git для профессионального программиста. К.:Питер Пресс, 2016 год, 496 с.

16. Behr K., Kim G., SpaffordG. Visible Ops Handbook. — Information Technology Process Institute, 2004.

17. Brikman Y. Hello, Startup: A Programmer’s Guide to Building Products, Techno logies and Teams. — O’Reilly, 2015.

18. Gruver G., Mouser T. Leading the Transformation: Applying Agile and DevOps Principles at Scale. — IT Revolution Press, 2015.

19. Hashimoto M. Vagrant: Up and Running. O'Reilly Media, 2013.

20. Humble J., Farley D. Continuous Delivery: Reliable Software Releases

through Build, Test and Deployment Automation. – Addison-Wesley Professional, 2010.

21. Humble J., Molesky J., O'Reilly B. Lean Enterprise. — O'Reilly, 2014.

22. Hunt A., Thomas D. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 1991.

23. Jaynes M. Taste Test: Puppet, Chef, Salt, AnsiЬIe. PuЬlisher, 2014.

24. Клерртапп M. Designing Data-Intensive Applications. O'Reilly Media, 2015.

25. Kurniawan У. AnsiЬIe for AWS. Leanpub, 2016.

26. Limoncelli T. A., Hogan C. J., Chalup S. R. The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems. Addison-Wesley Professional, 2014.

27. Mell P., Grance T. The NIST Definition of Cloud Computing. NIST Special Pblication 800-145, 2011.

28. Morris K. Infrastructure as Code: Managing Servers in the Cloud. – O'Reilly, 2016.

29. OpenSSH/Cookbook/Multiplexing, Wikibooks. URL:http://blt.ly/1bpeV0y. October 28, 2014.

30. Shafer A. C. Agile Infrastructure in Web Operations: Keeping the Data on Time. O'Reilly Media, 2010.

31. Volodymyr Melymuka, Teamcity 7 Continous Integration, K.: Packt publishing, 2012 year – 128 p.

БЛОГИ

1. Code as Craft (codeascraft.com).
2. dev2ops (dev2ops.org).
3. High Scalability (highscalability.com).
4. Kitchen Soap (www.kitchensoap.com).
5. Блог AWS (aws.amazon.com/blogs/aws).
6. Блог Пола Хамманта (paulhammant.com).
7. Блог Мартіна Фаулера (martinfowler.com/bliki).
8. Блог Gruntwork (blog.gruntwork.io).
9. Блог Євгенія Брікмана (www.ybrikman.com/writing)


13406