Instalacja narzędzi

Ansible z pierwszym inwentarzem, Semaphore w Dockerze, Terraform i OpenTofu.

- Instalacja Ansible i pierwszy inwentarz Zielonej
- Semaphore UI w Dockerze: Ansible z przeglądarki
- Instalacja Terraform albo OpenTofu i stan w bezpiecznym miejscu

Instalacja Ansible i pierwszy inwentarz Zielonej

Poziom: podstawy

Dotyczy: Ansible Core 2.16+, instalacja przez pipx i przez EPEL na Rocky Linux 9

Czas: ~30 min

Efekt: zainstalowany Ansible na stacji administratora albo na `app01`, klucz SSH rozniesiony na serwery Zielonej, pierwszy inwentarz YAML z trzema grupami i pierwszy playbook uruchomiony najpierw na sucho.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
EPEL	Extra Packages for Enterprise Linux: dodatkowe repozytorium pakietów dla RHEL i klonów
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

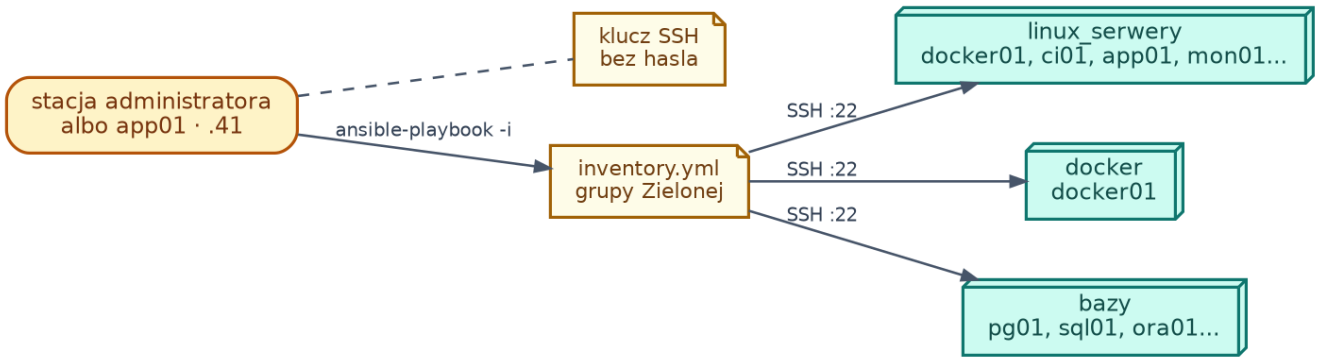
Zanim zaczniemy

> **Z ŻYCIA WZIĘTE**
>
> Tydzień po tym, jak Tomek ręcznie łatał piętnaście serwerów po kolei, siada z drugim administratorem i instaluje Ansible na `app01`. Dwie godziny później ten sam playbook, który wcześniej byłby piętnastoma osobnymi sesjami SSH, przechodzi przez cały inwentarz jedną komendą, sam po sobie, jeden serwer na raz.

Komendy wykonujemy z CLI bash jako user z dostępem SSH do maszyn zarządzanych. Python 3 jest wymagany tylko na maszynach zarządzanych (moduły Ansible go potrzebują), nie na maszynie, z której odpalamy Ansible.

Wymaganie	Uwaga
Python 3 na maszynie sterującej	Rocky Linux 9 ma go domyślnie, sprawdzamy `python3 --version`
Klucz SSH bez hasła	generujemy, jeśli go jeszcze nie ma: `ssh-keygen -t ed25519`
Konto z `sudo` na maszynach docelowych	Ansible łączy się zwykłym kontem i podnosi uprawnienia przez `become`, opisane na stronie Ansible: podstawy operacyjne
Osiągalność maszyn docelowych po SSH na porcie 22	sprawdzamy `ssh nazwa_usera@NazwaSerwera` przed pierwszym uruchomieniem Ansible

Instalujemy Ansible na jednej z dwóch maszyn z tabeli w 08-swiat-zielonej: stacji administratora do pracy ręcznej albo na `app01` (10.20.10.41), skąd później uruchamia go też Semaphore, opisany na stronie Semaphore UI w Dockerze: Ansible z przeglądarki. Poniżej budujemy inwentarz obejmujący część floty Zielonej: grupę `linux_serwery`, podgrupę `docker` i grupę `bazy`.



Stacja administratora czyta inwentarz i łączy się po SSH z grupami linux_serwery, docker i bazy (kliknięcie otwiera pełny rozmiar)

Dwa sposoby instalacji

Sposób	Skąd	Jaką wersję dostajemy	Kiedy wybieramy
`pipx`	pakiet Pythona z PyPI, zainstalowany we własnym, odizolowanym środowisku	zawsze najnowsza stabilna wersja Ansible	stacja administratora, gdy zależy nam na najnowszych modułach, niezależnie od tego, co pakuje dystrybucja

Sposób	Skąd	Jaką wersję dostajemy	Kiedy wybieramy
pakiet z EPEL	`dnf install ansible`, repozytorium opisane na stronie Repozytoria EPEL	wersja spakowana przez EPEL dla danej wersji Rocky, aktualizowana razem z resztą systemu	`app01`, gdy wolimy jedną komendę `dnf update` obejmującą wszystko na serwerze, kosztem nieco starszej wersji

> Obie metody da się mieć na tej samej maszynie naraz, ale w praktyce wybieramy jedną, żeby `ansible --version` nie budził wątpliwości, która instalacja właśnie odpowiada.

Przebieg

Instalacja przez pipx, na stacji administratora

Bootstrapujemy samo `pipx`, jeśli go jeszcze nie ma:

```
python3 -m pip install --user pipx
python3 -m pipx ensurepath
```

Otwieramy nową sesję terminala (albo `source ~/.bashrc`), żeby `~/.local/bin` trafiło do `PATH`, i instalujemy Ansible razem z jego zależnościami we własnym środowisku:

```
pipx install --include-deps ansible
```

Sprawdzenie wersji:

```
ansible --version
```

Przykład:

```
ansible [core 2.17.4]
  config file = None
  configured module search path = ['/home/nazwa_usera/.ansible/plugins/modules']
  ansible python module location =
/home/nazwa_usera/.local/pipx/venvs/ansible/lib/python3.12/site-packages/ansible
  executable location = /home/nazwa_usera/.local/bin/ansible
```

```
  python version = 3.12.3
```

Instalacja z EPEL, na app01

Włączamy EPEL i instalujemy pakiet, tak jak każdy inny pakiet z tego repozytorium:

```
sudo dnf install epel-release -y
sudo dnf install ansible -y
```

Sprawdzenie wersji tym samym poleceniem co wyżej, tylko z inną ścieżką do binarki (`/usr/bin/ansible` zamiast katalogu `pipx`).

Klucz SSH i pierwsze połączenie

Zakładamy parę kluczy, jeśli jeszcze jej nie mamy, i kopiujemy klucz publiczny na każdą maszynę docelową:

```
ssh-keygen -t ed25519 -C "ansible-zielona"
ssh-copy-id nazwa_usera@NazwaSerwera
```

Powtarzamy `ssh-copy-id` dla każdej maszyny z przyszłego inwentarza. Bez tego kroku każde połączenie Ansible pyta o hasło, co blokuje uruchomienie na wielu maszynach naraz.

Katalog projektu i ansible.cfg

Zakładamy katalog na inwentarz i playbooki, najlepiej w repozytorium Gitea, tak jak pierwsze repozytorium zakłada strona Pierwsze repozytorium: od pustego projektu do pierwszego pusha:

```
mkdir -p ~/projekty/automatyzacja-zielona
cd ~/projekty/automatyzacja-zielona
```

Plik `ansible.cfg` w katalogu projektu, czytany automatycznie, gdy uruchamiamy Ansible z tego katalogu. Pełny opis parametrów stoi na stronie Ansible: podstawy operacyjne:

```
[defaults]
inventory = ./inventory.yml
remote_user = nazwa_usera
forks = 10
```

```
host_key_checking = False
```

Pierwszy inwentarz Zielonej

Plik `inventory.yml`, trzy grupy: `linux_serwery` (wszystkie linuksy pod naszą opieką), `docker` (maszyna hostująca kontenery) i `bazy` (silniki baz danych na Linuksie):

```
all:
  children:
    linux_serwery:
      hosts:
        docker01:
          ansible_host: 10.20.10.42
        ci01:
          ansible_host: 10.20.10.43
        app01:
          ansible_host: 10.20.10.41
        mon01:
          ansible_host: 10.20.10.51
        ipa01:
          ansible_host: 10.20.10.53
        bkp01:
          ansible_host: 10.20.10.61
    docker:
      hosts:
        docker01:
    bazy:
      hosts:
        pg01:
          ansible_host: 10.20.10.33
        ora01:
          ansible_host: 10.20.10.32
        my01:
          ansible_host: 10.20.10.34
  vars:
    ansible_user: nazwa_usera
```

`docker01` celowo należy do dwóch grup naraz: do `linux_serwery`, bo jest linuksową maszyną jak każda inna, i do `docker`, bo to na niej stoją kontenery Zielonej. Playbook kierowany na `docker` dotyczy tylko tej jednej maszyny, playbook na `linux_serwery` obejmuje ją razem z resztą floty.

Sprawdzenie łączności ze wszystkimi maszynami z inwentarza naraz:

```
ansible all -i inventory.yml -m ping
```

Przykład:

```
docker01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
pg01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Pierwszy playbook: aktualizacja pakietów, jeden serwer na raz

Plik `aktualizacja.yml`:

```
- name: Aktualizacja pakietow na serwerach linuksowych
  hosts: linux_serwery
  become: true
  serial: 1
  tasks:
    - name: Aktualizacja wszystkich pakietow przez dnf
      ansible.builtin.dnf:
        name: "*"
        state: latest
```

> **Pojęcie: `serial`.** Liczba maszyn, na których playbook wykonuje się naraz, zanim przejdzie do kolejnej partii. `serial: 1` oznacza, że Ansible kończy cały playbook na jednym serwerze, zanim zacznie na drugim, dokładnie odwrotnie niż ręczne łatanie piętnastu serwerów naraz w piętnastu oknach terminala, ale bez czekania między nimi na nas.

Najpierw sprawdzamy, co playbook by zmienił, bez zmieniania czegokolwiek:

```
ansible-playbook -i inventory.yml aktualizacja.yml --check
```

Dopiero gdy wynik trybu `--check` wygląda tak, jak się spodziewamy, uruchamiamy playbook naprawdę, bez tej flagi.

Weryfikacja

Sprawdzamy, że Ansible widzi wszystkie trzy grupy z inwentarza:

```
ansible-inventory -i inventory.yml --graph
```

Przykład:

```
@all:
  |--@linux_serwery:
  |  |--docker01
  |  |--ci01
  |  |--app01
  |  |--mon01
  |  |--ipa01
  |  |--bkp01
  |--@docker:
  |  |--docker01
  |--@bazy:
  |  |--pg01
  |  |--ora01
  |  |--my01
```

Sprawdzamy, że playbook w trybie `--check` przechodzi bez błędów `UNREACHABLE` ani `FAILED` na żadnej maszynie z grupy `linux_serwery`.

Wycofanie zmian

Odinstalowanie Ansible zainstalowanego przez pipx:

```
pipx uninstall ansible
```

Odinstalowanie pakietu z EPEL:

```
sudo dnf remove ansible -y
```

Usunięcie klucza publicznego z maszyny docelowej wymaga zalogowania się na nią i ręcznego wykasowania odpowiedniej linii z `~/.ssh/authorized_keys` danego konta.

Gdy coś nie zadziała

Objaw	Przyczyna	Co robimy
`UNREACHABLE! ... Permission denied (publickey)`	klucz publiczny nie trafił jeszcze na maszynę docelową	`ssh-copy-id nazwa_usera@NazwaSerwera`, potem ponawiamy `ansible all -m ping`
`/usr/bin/python3: not found` na maszynie docelowej	interpreter Pythona brakuje na starszym albo minimalnym systemie	instalujemy `python3` ręcznie jedną komendą SSH albo ustawiamy `ansible_python_interpreter` w inwentarzu
`pipx: command not found` zaraz po instalacji	`~/.local/bin` jeszcze nie trafiło do `PATH` bieżącej sesji	otwieramy nową sesję terminala po `pipx ensurepath` albo robimy `source ~/.bashrc`
Playbook przechodzi z `--check`, ale prawdziwe uruchomienie się wywraca	zadanie używa modułu, który nie wspiera w pełni trybu `--check` (np. `ansible.builtin.command`)	sprawdzamy w dokumentacji modułu pole „Support: check_mode”

□□ Ćwiczenia

> Zadania sprawdzające.

1. Piszemy inwentarz YAML z jedną grupą zawierającą trzy własne serwery i sprawdzamy go komendą `ansible all -i plik.yml -m ping`.

Odpowiedź

```
all:
  children:
    moja_grupa:
      hosts:
        serwer1:
          ansible_host: 10.20.10.41
        serwer2:
          ansible_host: 10.20.10.42
```

```
        serwer3:
          ansible_host: 10.20.10.43
```

`ansible all -i plik.yml -m ping` łączy się z każdym z trzech hostów i zwraca `pong`, jeśli klucz SSH jest już rozniesiony, albo błąd `UNREACHABLE`, jeśli nie.

1. Wyjaśniamy różnicę między instalacją Ansible przez pipx a przez pakiet z EPEL i mówimy, kiedy wybralibyśmy tę drugą.

Odpowiedź

pipx instaluje zawsze najnowszą stabilną wersję Ansible we własnym, odizolowanym środowisku Pythona, niezależnie od tego, co pakuje dystrybucja. Pakiet z EPEL jest zwykle nieco starszy, ale aktualizuje się razem z resztą systemu jedną komendą `dnf update`. Na serwerze produkcyjnym typu `app01`, gdzie zależy nam na jednym spójnym cyklu aktualizacji całej maszyny, wybieramy EPEL.

1. Piszemy playbook aktualizujący pakiety jeden serwer po drugim i uruchamiamy go najpierw w trybie sprawdzenia, bez wprowadzania zmian.

Odpowiedź

Playbook z `hosts: linux_serwery`, `become: true` i `serial: 1`, zadanie modułem `ansible.builtin.dnf` z `name: "*"` i `state: latest`. Uruchomienie: `ansible-playbook -i inventory.yml aktualizacja.yml --check`.

Powiązane: Automatyzacja: jak przestaliśmy klikać te same rzeczy na dwudziestu serwerach · Ansible: podstawy operacyjne · Ansible, AWX, Semaphore, Rudder, Terraform: co do czego · Semaphore UI w Dockerze: Ansible z przeglądarki · Repozytoria EPEL · Pierwsze repozytorium: od pustego projektu do pierwszego pusha

Źródła: docs.ansible.com, rozdział „Installing Ansible" (instalacja przez pipx i przez pakiety dystrybucji) · docs.ansible.com, rozdział „Installing Ansible on specific operating systems" (EPEL dla RHEL i pochodnych)

Semaphore UI w Dockerze: Ansible z przeglądarki

Poziom: operacyjny
Dotyczy: Semaphore UI 2.19, obraz semaphoreui/semaphore:v2.19.12 **Czas: ~35 min**

Efekt: działający kontener Semaphore z własną bazą PostgreSQL, projekt podłączony do repozytorium playbooków w Gitea i pierwszy szablon zadania, który Ola potrafi uruchomić sama, bez konta SSH na żadnym serwerze.

Skróty użyte na tej stronie (8)

Skrót	Co znaczy
AD	Active Directory: katalog Microsoftu z kontami użytkowników, komputerów i grup w firmie
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
IT	Information Technology: dział albo praca związana z komputerami i infrastrukturą firmy
LDAP	Lightweight Directory Access Protocol: protokół do odpytywania katalogu użytkowników (AD, FreeIPA)
LDAPS	LDAP szyfrowany TLS, zwykle port 636
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera
StartTLS	Start Transport Layer Security: polecenie, które zamienia zwykłe, nieszyfrowane połączenie (np. LDAP na porcie 389) w szyfrowane już po jego nawiązaniu
WWW	World Wide Web: strony i aplikacje otwierane w przeglądarce po HTTP albo HTTPS

Zanim zaczniemy

Z ŻYCIA WZIĘTE

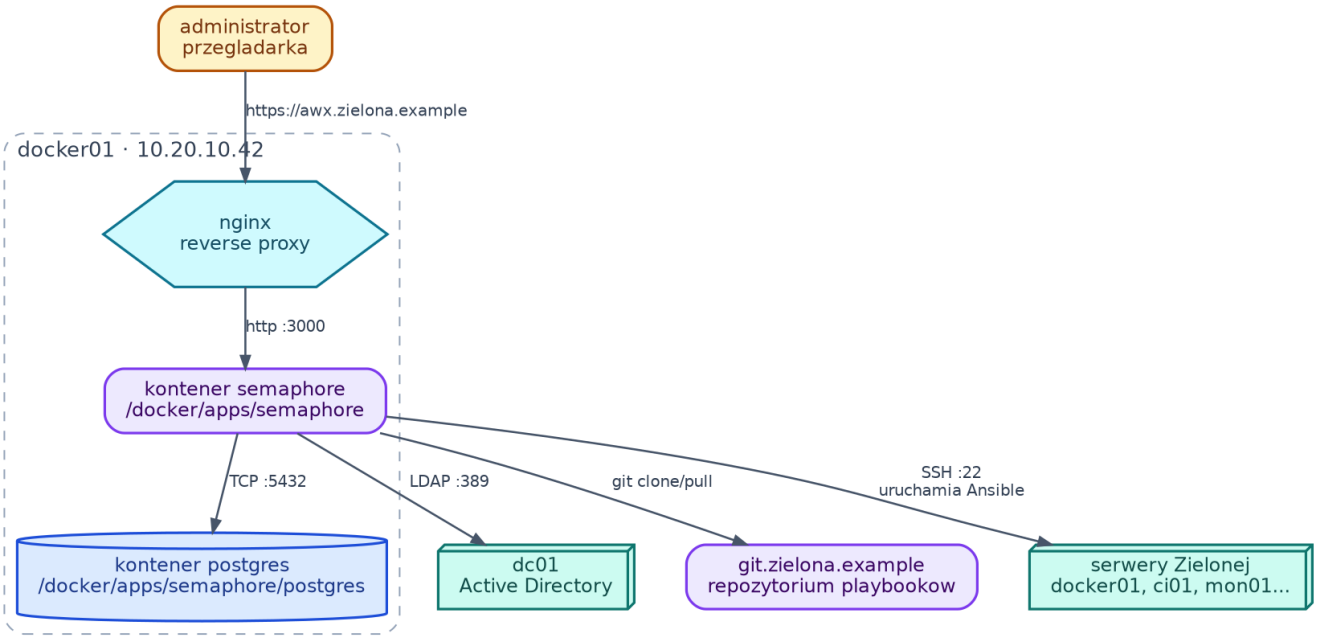
Ola od miesiąca prosi o dostęp „tylko na chwilę” do serwerów, na których stoją jej kontenery, żeby samodzielnie odpalić playbook aktualizujący zależności. Zespół IT nie chce rozdawać kluczy SSH do produkcji trzeciej osobie, ale też nie chce być jedynym, kto potrafi wpisać `ansible-playbook`. Semaphore rozwiązuje oba problemy naraz: Ola dostaje przycisk w

> przeglądarce, klucz SSH zostaje tam, gdzie był.

Komendy wykonujemy z CLI bash jako user z uprawnieniami sudo albo root, na `docker01`, z zainstalowanym Dockerem i wtyczką Compose. Zakładamy, że Ansible i pierwszy inwentarz już istnieją, zgodnie ze stroną Instalacja Ansible i pierwszy inwentarz Zielonej, i że playbooki leżą w repozytorium Gitea.

> W naszym środowisku dane aplikacji trzymamy w `/docker/apps/<nazwa>`, a plik Compose w `/docker/compose/<nazwa>/compose.yaml`. To konwencja, nie wymóg Dockera: można ją zmienić, byle w całej firmie była jedna. Katalog `/docker` to osobny wolumen `vgDocker/lvDocker`, żeby pełny dysk aplikacji nie zatrzymał systemu.

Wymaganie	Uwaga
Docker Engine i wtyczka Compose	sprawdzamy `docker compose version`
Wolny port `3000` na `docker01`	reverse proxy nginx stawiamy przed nim tak samo jak przed innymi aplikacjami Zielonej
Repozytorium z playbookami w Gitea	patrz Pierwsze repozytorium: od pustego projektu do pierwszego pusha
Klucz SSH z poprzedniej strony	prywatna połowa trafia do magazynu poświadczeń Semaphore, nie do repozytorium



Architektura Semaphore na docker01: nginx, kontener semaphore i postgres, połączenia do AD, Gitea i serwerów Zielonej (kliknięcie otwiera pełny rozmiar)

Przebieg

Compose z Semaphore i własną bazą

Zakładamy katalogi zgodnie z konwencją ścieżek:

```
sudo mkdir -p /docker/apps/semaphore/postgres /docker/apps/semaphore/config
sudo mkdir -p /docker/compose/semaphore
```

Plik `/docker/compose/semaphore/.env`, obok `compose.yaml`, nie w nim samym. Klucz szyfrowania kluczy generujemy sami, nie kopiujemy żadnego przykładu z dokumentacji:

```
openssl rand -base64 32
```

```
SEMAPHORE_DB_PASSWORD=HasloDoBazy
SEMAPHORE_ADMIN_PASSWORD=HasloAdministratora
SEMAPHORE_ACCESS_KEY_ENCRYPTION=WygenerowanyKluczBase64
```

Plik `/docker/compose/semaphore/compose.yaml`:

```
services:
  postgres:
    image: postgres:16
    container_name: semaphore-postgres
    restart: always
    environment:
      - POSTGRES_USER=semaphore
      - POSTGRES_PASSWORD=${SEMAPHORE_DB_PASSWORD}
      - POSTGRES_DB=semaphore
    volumes:
      - /docker/apps/semaphore/postgres:/var/lib/postgresql/data

  semaphore:
    image: semaphoreui/semaphore:v2.19.12
    container_name: semaphore
```

```
    restart: always
    ports:
      - "3000:3000"
    environment:
      - SEMAPHORE_DB_DIALECT=postgres
      - SEMAPHORE_DB_HOST=postgres
      - SEMAPHORE_DB_PORT=5432
      - SEMAPHORE_DB_USER=semaphore
      - SEMAPHORE_DB_PASS=${SEMAPHORE_DB_PASSWORD}
      - SEMAPHORE_DB=semaphore
      - SEMAPHORE_ADMIN=admin
      - SEMAPHORE_ADMIN_NAME=admin
      - SEMAPHORE_ADMIN_EMAIL=admin@zielona.example
      - SEMAPHORE_ADMIN_PASSWORD=${SEMAPHORE_ADMIN_PASSWORD}
      - SEMAPHORE_ACCESS_KEY_ENCRYPTION=${SEMAPHORE_ACCESS_KEY_ENCRYPTION}
      - TZ=Europe/Warsaw
    volumes:
      - /docker/apps/semaphore/config:/etc/semaphore
    depends_on:
      - postgres
```

Pierwsze uruchomienie:

```
cd /docker/compose/semaphore
docker compose up -d
```

◫ ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: pierwszy ekran logowania pod `http://docker01:3000`, formularz z polami „Username” i „Password”.

Projekt podłączony do Gitea

W interfejsie zakładamy nowy projekt i wskazujemy repozytorium z playbookami: adres `git.zielona.example/zespol-it/automatyzacja-zielona.git`, gałąź `main`. Semaphore klonuje repozytorium przy każdym uruchomieniu zadania, więc treść playbooka zawsze pochodzi z ostatniego commita, nie z tego, co ktoś wkleił kiedyś do formularza.

◫ ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: ekran „New Project” z wypełnionym polem repozytorium Git i wybraną gałęzią `main`.

Klucze w magazynie Semaphore

W sekcji poświadczeń dodajemy klucz SSH wygenerowany na stronie Instalacja Ansible i pierwszy inwentarz Zielonej, typ „SSH Key”, wklejając prywatną część klucza. Playbook odwołuje się do tego poświadczenia przez nazwę, a osoba uruchamiająca zadanie, na przykład Ola, nie widzi ani nie potrzebuje samego klucza.

Inwentarz dodajemy tym samym plikiem `inventory.yml` co na poprzedniej stronie, wskazując go z repozytorium zamiast wklejać ręcznie, żeby obie strony (Ansible z linii poleceń i Ansible z Semaphore) korzystały z jednego źródła prawdy.

Szablon zadania

Nowy szablon zadania łączy trzy elementy: projekt, inwentarz i klucz SSH z kroków wyżej, plus konkretny playbook z repozytorium, na przykład `aktualizacja.yml`. Zapisany szablon staje się przyciskiem „Run”, który Ola klika bez znajomości linii poleceń.

◫ ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: ekran „New Task Template” z wypełnionymi polami Playbook, Inventory i SSH Key, wskazującymi na `aktualizacja.yml`.

Harmonogram

Pole harmonogramu przyjmuje standardowy pięciopolowy zapis cron. Przykład: uruchomienie playbooka porządkującego stare pliki tymczasowe codziennie o drugiej w nocy:

```
0 2 * * *
```

◫ ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: ekran „Schedules” z dodanym wpisem `0 2 * * *` przypisanym do szablonu zadania.

Logowanie LDAP z AD

Semaphore umie uwierzytelniać przez LDAP, w tym przez Active Directory, ale konfiguruje się to plikiem `config.json`, nie samym interfejsem. Po pierwszym uruchomieniu plik leży w `/docker/apps/semaphore/config/config.json` (zamontowanym jako `/etc/semaphore` w kontenerze) i dopisujemy do niego sekcję LDAP:

```
{
  "ldap_enable": true,
  "ldap_needtls": true,
  "ldap_server": "dc01.ad.zielona.example:389",
  "ldap_binddn": "CN=svc-semaphore,OU=Serwisowe,DC=ad,DC=zielona,DC=example",
  "ldap_bindpassword": "HasloKontaSerwisowego",
  "ldap_searchdn": "OU=Uzytkownicy,DC=ad,DC=zielona,DC=example",
  "ldap_searchfilter": "(sAMAccountName=%s)",
  "ldap_mappings": {
    "dn": "dn",
    "mail": "mail",
    "uid": "sAMAccountName",
    "cn": "displayName"
  }
}
```

Po zapisaniu pliku restartujemy kontener, żeby wczytał nową konfigurację:

```
docker compose restart semaphore
```

`ldap_binddn` wskazuje konto serwisowe w AD, którym Semaphore przeszukuje katalog, nie konto osoby logującej się. Dokładne wymaganie co do szyfrowania połączenia (`ldap_needtls`, port 389 ze StartTLS czy 636 z LDAPS) sprawdzamy w dokumentacji Semaphore dla naszej wersji, bo bywa zależne od konfiguracji samego `dc01`.

Dlaczego Semaphore, a nie AWX

Pełne porównanie obu narzędzi, niezależnie od Zielonej, stoi na stronie AWX i Semaphore. Tutaj tylko powód konkretnej decyzji:

Kryterium w Zielonej	Semaphore	AWX

Gdzie by dziś stanął	jeden dodatkowy kontener na już istniejącym `docker01`	potrzebuje Kubernetesa albo wielu współpracujących usług; jedyny kandydat to laboratoryjny `k3s01`
Kto go utrzyma	dwuosobowy zespół IT, bez dedykowanego administratora platformy	zwykle wymaga kogoś, kto na stałe pilnuje samej platformy, nie tylko playbooków
Kiedy Zielona po niego sięgnie	już teraz, do codziennego uruchamiania playbooków	gdy `k3s01` przejdzie z laboratorium do produkcji

Weryfikacja

Sprawdzamy, że oba kontenery działają:

```
docker compose ps
```

Przykład:

```
NAME                IMAGE                              STATUS
semaphore           semaphoreui/semaphore:v2.19.12     Up 3 minutes
semaphore-postgres  postgres:16                        Up 3 minutes
```

Sprawdzamy, że interfejs WWW odpowiada:

```
curl -I http://localhost:3000
```

Przykład:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

Uruchamiamy zapisany szablon zadania z przeglądarki i sprawdzamy, że log zadania kończy się statusem „Success”, tak samo jak wcześniej kończyło się uruchomienie tego samego playbooka z linii poleceń.

Wycofanie zmian

Zatrzymanie i usunięcie kontenerów, bez kasowania danych:

```
docker compose down
```

Pełne usunięcie razem z danymi i historią zadań:

```
docker compose down -v
sudo rm -rf /docker/apps/semaphore
```

Gdy coś nie zadziała

Objaw	Przyczyna	Naprawa
Kontener `semaphore` restartuje się w pętli	baza `semaphore-postgres` jeszcze nie wystartowała	`docker compose logs postgres`, czekamy albo sprawdzamy hasło w `.env`
Zadanie kończy się `Permission denied (publickey)`	do szablonu podpięto złe poświadczenie albo klucz publiczny nie trafił na maszynę docelową	sprawdzamy przypisany klucz SSH w szablonie i powtarzamy `ssh-copy-id` z poprzedniej strony
Logowanie LDAP zwraca błąd połączenia	`dc01` nieosiągalny z kontenera albo zła wartość `ldap_needtls`	`docker compose exec semaphore sh` i próba połączenia do `dc01.ad.zielona.example:389` z wnętrza kontenera
Harmonogram nie uruchamia zadania o właściwej porze	zmienna `TZ` kontenera nie ustawiona albo ustawiona na inną strefę niż oczekiwana	ustawiamy `TZ=Europe/Warsaw` w `compose.yaml` i restartujemy kontener

□□ Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy, dlaczego prywatny klucz SSH trafia do magazynu poświadczeń Semaphore, a nie do repozytorium Gitea razem z playbookiem.

Odpowiedź

Repozytorium widzi cały zespół, a klucz prywatny daje pełny dostęp do maszyn docelowych. Magazyn poświadczeń Semaphore pozwala playbookowi odwołać się do klucza przez nazwę, bez ujawniania jego treści osobie uruchamiającej zadanie, nawet jeśli ta osoba ma prawo kliknąć „Run”.

1. Zespół chce, żeby Ola mogła uruchamiać wyłącznie jeden konkretny playbook, bez dostępu do reszty projektów w Semaphore. Opisujemy, jakim mechanizmem to

ograniczyć.

Odpowiedź

Osobny projekt w Semaphore ze swoim własnym zestawem poświadczeń, inwentarzy i szablonów zadań, do którego dodajemy Olę jako jedynego użytkownika spoza zespołu IT, zamiast dawać jej dostęp do głównego projektu z całą resztą automatyzacji.

1. Piszemy wpis harmonogramu, który uruchamia szablon zadania w każdą sobotę o 3:00.

Odpowiedź

`0 3 * * 6` (minuta 0, godzina 3, każdy dzień miesiąca, każdy miesiąc, dzień tygodnia 6, czyli sobota).

Powiązane: Instalacja Ansible i pierwszy inwentarz Zielonej · AWX i Semaphore · Ansible, AWX, Semaphore, Rudder, Terraform: co do czego · Pierwsze repozytorium: od pustego projektu do pierwszego pusha · Konwencja Dockera w Zielonej

Źródła: semaphoreui.com/docs, rozdział „Docker" (zmienne środowiskowe i ścieżka `/etc/semaphore`) · semaphoreui.com/docs, rozdział „LDAP and Active Directory” · semaphoreui.com/docs, rozdział „Schedules” (format cron)

Instalacja Terraform albo OpenTofu i stan w bezpiecznym miejscu

Poziom: podstawy **Dotyczy: Terraform 1.16.x, OpenTofu 1.12.x jako alternatywa**

Czas: ~30 min

Efekt: zainstalowany `terraform` albo `tofu` na stacji administratora i na `app01`, pierwszy `terraform init` przechodzący bez błędu i stan zapisywany w backendzie zdalnym, nie na lokalnym dysku ani w repozytorium Git.

Skróty użyte na tej stronie (6)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
AWS	Amazon Web Services: chmura Amazonu
BSL	Business Source License: licencja, na ktora HashiCorp zmienil w 2023 roku Terraform, ograniczajaca prawo do oferowania narzedzia jako konkurencyjnej platnej uslugi
CI	Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie; w ITIL także configuration item, czyli element konfiguracji: każdy składnik usługi zapisany w CMDB
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
MPL	Mozilla Public License: otwarta licencja, na której HashiCorp wydawał swoje programy do sierpnia 2023 roku i na której jest dziś Talos Linux

Zanim zaczniemy

> **Z ŻYCIA WZIĘTE**
>
> Zespół Oli dostaje zielone światło na pilotaż nowej usługi wewnętrznej i potrzebuje trzech maszyn wirtualnych na klastrze `hv01`/`hv02`, identycznie skonfigurowanych, do piątku. Ręczne klikanie przez konsolę Hyper-V Managera trzy razy z rzędu, z tymi samymi ustawieniami sieci

> i dysków, to dokładnie ten rodzaj powtarzalnej pracy, po który sięgamy po Terraform, nie po Ansible: te maszyny jeszcze nie istnieją.

Komendy wykonujemy z CLI bash (na Rocky Linux) albo z PowerShell (na Windows), z uprawnieniami wystarczającymi do instalacji pakietów. Instalujemy narzędzie na dwóch maszynach z tabeli w 08-swiat-zielonej: stacji administratora do pracy ręcznej i na `app01` (10.20.10.41), skąd później uruchamia je też Semaphore albo pipeline CI.

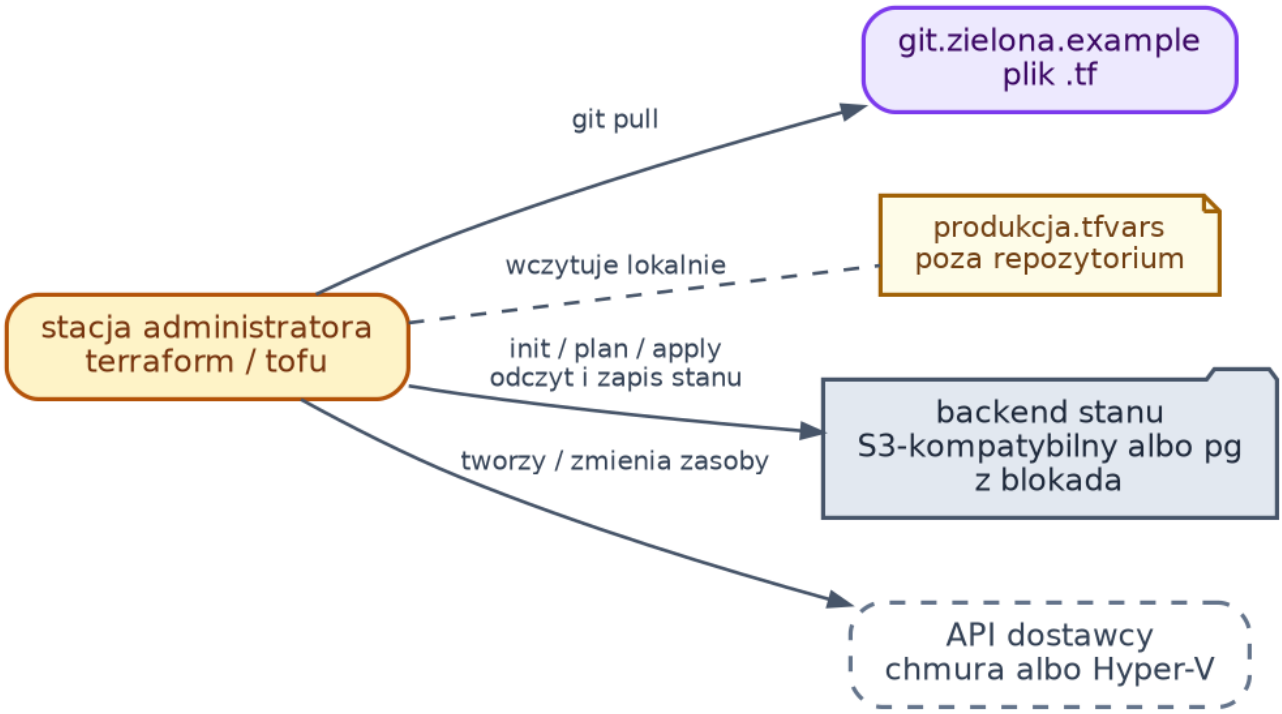
Wymaganie	Uwaga
Uprawnienia administracyjne na maszynie instalującej	`sudo` na Linuksie, konto administratora na Windows
Dostęp do internetu albo do wewnętrznego lustra pakietów	pobieramy binarkę z repozytorium producenta
Miejsce na backend stanu	Postgres na `pg01` albo magazyn S3-kompatybilny, patrz niżej

Terraform czy OpenTofu

Terraform i OpenTofu czytają te same pliki `.tf` i obsługują te same komendy, bo OpenTofu powstał jako fork Terraform sprzed zmiany licencji, opisanej na stronie Automatyzacja: jak przestaliśmy klikać te same rzeczy na dwudziestu serwerach. Wybór między nimi to dziś głównie kwestia licencji i tego, kto ma nadzór nad projektem, nie różnic w codziennej pracy.

	Terraform	OpenTofu
Licencja	Business Source License, ograniczająca oferowanie go jako konkurencyjnej usługi	Mozilla Public License 2.0, w pełni otwarta
Kto nadzoruje	HashiCorp (IBM)	fundacja Linux Foundation
Zgodność plików `.tf`	punkt odniesienia	wsteczna zgodność z Terraform sprzed forka, nowe funkcje mogą się rozjeżdżać z czasem
Aktualny stan	sprawdzamy w dokumentacji, licencja i warunki użycia bywają zmieniane	sprawdzamy w dokumentacji projektu na opentofu.org

Zielona nie ma dziś powodu, żeby świadczyć komercyjną usługę konkurencyjną wobec HashiCorp, więc licencja BSL sama w sobie jej nie przeszkadza. Poniżej pokazujemy instalację obu, wybieramy jedno.



Stacja administratora uruchamia terraform albo tofu, wczytuje pliki z Gitea i sekrety z tfvars, zapisuje stan w backendzie z blokadą, woła API dostawcy (kliknięcie otwiera pełny rozmiar)

Przebieg

Instalacja Terraform na Rocky Linux z repozytorium HashiCorp

```
sudo yum install -y yum-utils shadow-utils
sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
sudo yum -y install terraform
```

Sprawdzenie wersji:

```
terraform -version
```

Przykład:

```
Terraform v1.16.4
on linux_amd64
```

Instalacja OpenTofu na Rocky Linux, oficjalnym skryptem

```
curl --proto '=https' --tlsv1.2 -fsSL https://get.opentofu.org/install-opentofu.sh -o install-opentofu.sh
chmod +x install-opentofu.sh
./install-opentofu.sh --install-method rpm
rm -f install-opentofu.sh
```

> Skrypt instalacyjny warto przejrzeć przed uruchomieniem, tak jak podpowiada sama dokumentacja OpenTofu: `less install-opentofu.sh` zanim damy mu prawo wykonania.

Binarka instaluje się pod nazwą `tofu`, nie `terraform`:

```
tofu -version
```

Przykład:

```
OpenTofu v1.12.3
on linux_amd64
```

Instalacja na Windows, przez winget

Na stacji administratora z Windows 11:

```
winget install Hashicorp.Terraform
```

Weryfikacja tym samym poleceniem `terraform -version` co na Linuksie, z terminala PowerShell.

Pierwsze polecenia

Pełny opis `init`, `plan`, `apply` i pracy ze stanem stoi na stronie Terraform: podstawy, identycznie dla `terraform` i dla `tofu`, bo różni je tylko nazwa binarki. Tutaj tylko pierwsze sprawdzenie, że instalacja działa, w katalogu z jednym prostym plikiem `.tf`:

```
terraform init
terraform validate
```

Dlaczego stan nie leży w Git

Plik stanu **Pojęcie: state** opisuje strona Terraform: podstawy. W skrócie: zawiera surowe dane o utworzonych zasobach, często z wartościami uznawanymi za wrażliwe w postaci jawnej, i wymaga blokady przy pracy zespołowej, której Git w ogóle nie oferuje. Dwie osoby z zespołu Zielonej, każda z osobnym lokalnym stanem tego samego projektu, prędzej czy później nadpiszą sobie nawzajem zmiany.

Backend: Postgres na pg01

Zielona ma już działający serwer PostgreSQL na `pg01`, więc backend `pg` nie wymaga stawiania niczego nowego, tylko jednej bazy:

```
CREATE DATABASE terraform_state;
CREATE USER terraform WITH ENCRYPTED PASSWORD 'HasloDoBazy';
GRANT ALL PRIVILEGES ON DATABASE terraform_state TO terraform;
```

```
terraform {
  backend "pg" {
    conn_str = "postgres://terraform:HasloDoBazy@pg01:5432/terraform_state"
  }
}
```

Blokada w backendzie `pg` działa od razu, przez mechanizm blokad samego PostgreSQL, bez dodatkowej konfiguracji.

Backend: magazyn S3-kompatybilny

Tam, gdzie Zielona wolałaby magazyn obiektowy zamiast bazy (na przykład na `bkp01`, obok reszty kopii zapasowych), backend `s3` obsługuje też magazyny inne niż Amazon, przez wskazanie własnego adresu:

```
terraform {
  backend "s3" {
    bucket         = "NazwaBucketa-terraform-state"
    key            = "NazwaProjektu/terraform.tfstate"
```

```
    region         = "eu-central-1"
    use_path_style = true
    use_lockfile   = true
    skip_region_validation      = true
    skip_credentials_validation = true
    skip_requesting_account_id  = true

    endpoints = {
      s3 = "https://bkp01.zielona.example:9000"
    }
  }
}
```

`use_lockfile` włącza blokadę stanu wprost w magazynie S3-kompatybilnym, bez osobnej usługi do samych blokad, którą starsze poradniki opisują jako wymaganą. Wartość `skip_requesting_account_id` i pozostałe flagi `skip_*` wyłączają sprawdzenia specyficzne dla prawdziwego AWS, których magazyn inny niż Amazon i tak by nie obsłużył.

Sekrety poza repozytorium

Plik `.tfvars` z hasłami i kluczami dostawcy nie trafia do repozytorium Gitea razem z resztą kodu `.tf`:

```
produkcja.tfvars
```

```
echo "*.tfvars" >> .gitignore
```

Zastosowanie wartości z takiego pliku przy `apply` opisuje strona Terraform: podstawy (`-var-file`); tutaj pilnujemy tylko, żeby sam plik nigdy nie trafił do `git add`.

Weryfikacja

Sprawdzamy, że backend faktycznie przyjął połączenie i zablokował stan podczas operacji:

```
terraform init
```

Przykład:

```
Initializing the backend...

Successfully configured the backend "pg"! Terraform will automatically
use this backend unless the backend configuration changes.

Terraform has been successfully initialized!
```

Sprawdzamy, że na dysku lokalnym nie ma pliku `terraform.tfstate` z prawdziwymi danymi: powinien tam zostać co najwyżej pusty wpis wskazujący na backend, nie sam stan.

```
ls -la terraform.tfstate 2>/dev/null || echo "brak lokalnego pliku stanu, tak jak powinno byc"
```

Wycofanie zmian

Odinstalowanie Terraform:

```
sudo dnf remove terraform -y
sudo rm -f /etc/yum.repos.d/hashicorp.repo
```

Odinstalowanie OpenTofu (metoda rpm):

```
sudo dnf remove tofu -y
```

Na Windows:

```
winget uninstall Hashicorp.Terraform
```

Sam backend (baza `terraform_state` na `pg01` albo bucket na `bkp01`) usuwamy osobno, dopiero gdy mamy pewność, że żaden projekt już go nie używa.

Gdy coś nie zadziała

Objaw	Przyczyna	Co robimy
`terraform init` zgłasza błąd połączenia z backendem `pg`	`pg01` nieosiągalny z maszyny albo złe hasło w `conn_str`	sprawdzamy `psql -h pg01 -U terraform -d terraform_state`, poprawiamy `conn_str`

Objaw	Przyczyna	Co robimy
`Error: Backend configuration changed` przy kolejnym `init`	zmieniono blok `backend` bez migracji istniejącego stanu	`terraform init -migrate-state`, po dokładnym przeczytaniu, co migracja przeniesie
Dwie osoby uruchamiają `apply` naraz i druga dostaje błąd blokady	tak ma działać: backend odrzuca drugą operację, dopóki pierwsza się nie skończy	czekamy na zakończenie pierwszego `apply`, nie próbujemy wymuszać zwolnienia blokady bez pewności, że pierwsza operacja faktycznie padła
`winget install Hashicorp.Terraform` nie znajduje pakietu	winget nie ma jeszcze zaktualizowanych źródeł pakietów	`winget source update`, ponawiamy instalację

Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy, dlaczego plik stanu Terraform nie powinien trafić do tego samego repozytorium Git co pliki `.tf`.

Odpowiedź

Plik stanu zawiera surowe dane o utworzonych zasobach, często z wartościami wrażliwymi w postaci jawnej, i wymaga blokady przy pracy zespołowej, której Git nie zapewnia. Dwie osoby pracujące na osobnych kopiach lokalnego stanu nieuchronnie się rozjadą.

1. Zespół ma już działający serwer PostgreSQL i nie chce stawiać dodatkowej usługi tylko pod stan Terraform. Wskazujemy backend, który to rozwiąże, i piszemy jego blok konfiguracji.

Odpowiedź

Backend `pg`, wskazujący istniejący serwer, na przykład `pg01`:

```
terraform {
  backend "pg" {
    conn_str = "postgres://terraform:HasloDoBazy@pg01:5432/terraform_state"
  }
}
```

1. Opisujemy różnicę między Terraform a OpenTofu, jaką napotkalibyśmy w codziennej pracy z plikami `.tf`.

Odpowiedź

Praktycznie żadnej na starcie: OpenTofu powstał jako fork Terraform sprzed zmiany licencji i czyta te same pliki `.tf` tymi samymi komendami, tylko pod nazwą binarki `tofu`. Różnica dotyczy licencji i tego, kto nadzoruje projekt, a z czasem może dotyczyć też nowych funkcji, które w jednym z narzędzi pojawią się wcześniej niż w drugim.

Powiązane: Automatyzacja: jak przestaliśmy klikać te same rzeczy na dwudziestu serwerach · Terraform: podstawy · Ansible, AWX, Semaphore, Rudder, Terraform: co do czego

Źródła: developer.hashicorp.com/terraform/install · opentofu.org/docs/intro/install/rpm · opentofu.org, historia projektu i licencja MPL 2.0 · developer.hashicorp.com/terraform/language/backend/pg · developer.hashicorp.com/terraform/language/backend/s3