Mariusz Dworniczak

16 gotowych scenariuszy do uruchomienia w każdym systemie

Terraform

W PRAKTYCE

Buduj i automatyzuj infrastrukturę chmurową oraz zarządzaj nią z wykorzystaniem **Dockera**

Helion

Redaktor prowadzący: Natalia Hermansa

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą AdobeStock.com.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/terraf
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
https://ftp.helion.pl/przyklady/terraf.zip

ISBN: 978-83-289-3838-0

Printed in Poland.

Spis treści

Część III. Skalowanie i zaawansowane koncepcje (advanced core)

Wstęp: dlaczego Infrastructure as Code?

Witaj w świecie, w którym serwery, sieci i bazy danych nie są już fizycznymi urządzeniami, których dotykasz, ani zestawem pól do żmudnego wyklikania w panelu administracyjnym. Witaj w świecie, w którym Twoja infrastruktura jest **kodem**.

Jeśli kiedykolwiek spędziłeś godziny na ręcznym konfigurowaniu środowiska tylko po to, by na końcu dowiedzieć się, że „u mnie działa, a na produkcji nie", lub jeśli z drżeniem rąk wprowadzałeś zmiany na żywym organizmie systemu — ta książka jest dla Ciebie.

Co zyskasz dzięki lekturze tej książki?

To nie jest kolejny teoretyczny wykład. Ta książka to **kompletny zestaw sprawdzonych „przepisów-gotowców"**. Otrzymasz dostęp do **16 praktycznych laboratoriów**, które zostały zaprojektowane tak, abyś mógł je uruchomić natychmiast na swoim komputerze.

- **Pełna niezależność sprzętowa:** nieważne, czy pracujesz na **Windowsie**, **Linuksie** czy **Macu**. Dzięki wykorzystaniu Dockera Twoje środowisko pracy będzie identyczne jak u profesjonalistów, bez konieczności zakładania płatnych kont w chmurze na starcie.
- **Uniwersalne fundamenty:** skupiamy się na narzędziu Terraform, ale przede wszystkim na zasadach. *Best practices* to *best practices* — zasady, których nauczysz się tutaj, przeniesiesz 1:1 do swoich przyszłych projektów w AWS, Azure czy Google Cloud.
- **Inwestycja w karierę:** wiedza zawarta w tych rozdziałach to fundament, który pozwoli Ci ewoluować z administratora w stronę **Cloud Engineer**, **DevOps**, a w przyszłości nawet **Cloud Architect**.

Czego nauczysz się w praktyce?

Moim celem było napisanie podręcznika, który da Ci realny warsztat do ręki. Razem przejdziemy drogę od instalacji prostego kontenera, przez zaawansowaną logikę i pętle, aż po profesjonalne testowanie jakości kodu.

Dzięki tej lekturze:

- **przestaniesz klikać, zaczniesz automatyzować:** zapiszesz skomplikowaną architekturę w czytelnych plikach tekstowych;
- **opanujesz workflow Terraforma:** poznasz standardy pracy (`init`, `plan`, `apply`, `destroy`), które są normą w Dolinie Krzemowej;

- **zrozumiesz, czym jest idempotentność:** dowiesz się, dlaczego to najważniejsze słowo w słowniku nowoczesnego inżyniera i jak gwarantuje ono święty spokój w pracy.

Dla kogo jest ta książka?

Niezależnie od tego, czy jesteś administratorem systemów, programistą chcącym kontrolować swoje środowisko, czy studentem marzącym o wejściu do świata IT — ten podręcznik poprowadzi Cię za rękę. Nie potrzebujesz doktoratu z Linuksa. Potrzebujesz ciekawości i zainstalowanego Dockera.

Jak korzystać z tej lektury?

To nie jest powieść — to instrukcja budowy. Zachęcam Cię, abyś każdą linię kodu z moich „gotowców" przeanalizował i uruchomił samodzielnie. Popełniaj błędy, psuj konfigurację i naprawiaj ją razem ze mną. To właśnie w tych momentach, gdy terminal wyrzuci błąd, nauczysz się najwięcej.

Infrastruktura jako kod to zmiana myślenia o stabilności i Twoim wolnym czasie.

Zapnij pasy. Zaczynamy automatyzację Twojego sukcesu!

Kody źródłowe wybranych przykładów dostępne są pod adresem:
https://ftp.helion.pl/przyklady/terraf.zip

ROZDZIAŁ 3.

Pierwszy projekt i cykl życia (workflow)

Zanim wydasz pierwsze polecenie w terminalu, musimy przyjrzeć się narzędziu komunikacji. Terraform nie korzysta ze skomplikowanych skryptów programistycznych; posługuje się autorskim językiem **HCL** (***HashiCorp Configuration Language***). Jest on kluczem do sukcesu Terraforma — został zaprojektowany tak, aby był maksymalnie czytelny dla człowieka, a jednocześnie rygorystyczny strukturalnie dla maszyny.

3.1. Anatomia pliku .tf — język HCL w pigułce

Plik z rozszerzeniem *.tf* to dokument tekstowy, w którym opisujesz swoją infrastrukturę. Zamiast sporządzać listę kroków (zrób to, potem tamto), opisujesz w nim **stan docelowy**. Terraform porównuje ten opis z rzeczywistością i podejmuje niezbędne kroki, aby oba te światy się zgadzały.

Spójrzmy na trzy fundamenty, na których opiera się niemal każdy plik konfiguracyjny:

1. **Blok `terraform {}` — konfiguracja systemowa.**

   To „centrum dowodzenia" Twoim projektem. Tutaj określasz parametry techniczne, które nie są częścią samej infrastruktury, ale są niezbędne do jej zbudowania.

   - **Wymagani dostawcy (`required_providers`):** wskazujesz tu konkretne wtyczki, z których Terraform musi skorzystać (np. `local`, `docker` czy `aws`). Każdy dostawca posiada swoje źródło (`source`) oraz wersję (`version`).
   - **Wersja Terraforma:** możesz tu zastrzec, że Twój kod działa tylko w określonej wersji narzędzia, co zapobiega problemom w pracy zespołowej.

2. **Blok `provider {}` — sterowniki i autoryzacja.**

   Provider (dostawca) to „tłumacz" między Terraformem a zewnętrznym API.

   - Jeśli blok `terraform {}` mówi: „Będę potrzebował wtyczki do Dockera", to blok `provider {}` mówi: „Oto jak masz się z tym Dockerem połączyć".
   - W przypadku dostawców chmurowych (AWS, Azure) to właśnie tutaj definiuje się region (np. `us-east-1`) lub klucze dostępowe.

3. **Blok `resource {}` — serce infrastruktury.**

   To tutaj dzieje się magia. Blok zasobu definiuje, **co** ma zostać stworzone. Składnia zawsze wygląda tak samo: `resource "typ_zasobu" "nazwa_logiczna" { ... }`.

   - **Typ zasobu:** (np. `local_file`) musi być rozpoznawany przez danego dostawcę.
   - **Nazwa logiczna:** (np. `"hello"`) to nazwa, której używasz tylko wewnątrz kodu Terraforma, aby odnosić się do tego elementu.

- **Argumenty:** wewnątrz klamer {} wpisujesz parametry, np. ścieżkę do pliku (`filename`) czy jego zawartość (`content`).

Pamiętaj o **zasadzie deklaratywności**. W pliku *.tf* nie wydajesz rozkazów („stwórz mi plik"). Ty składasz deklarację: „Deklaruję, że w moim systemie ma istnieć plik o nazwie *hello.txt* i zawartości 'Hello Terraform'". To Terraform bierze na siebie ciężar sprawdzenia, czy ten plik już tam jest, a jeśli go tam nie ma — jak go utworzyć.

Teoria jest przedstawiona tutaj krótko, ponieważ języka HCL nauczysz się najlepiej, posługując się nim w dalszej części tej książki, gdy będziesz wykonywać ćwiczenia i analizować przykłady. **To jak z nauką pływania — nie nauczysz się pływać, nie wchodząc do wody.** Traktuj ten opis jako mapę, która pomoże Ci nie utonąć podczas pierwszego kontaktu z kodem.

3.2. Sprawdź swoją wiedzę: czy już widzisz strukturę HCL?

Zanim wejdziemy do laboratorium, sprawdźmy, czy Twoje oko wyłapuje detale składniowe. Pamiętaj: w kodzie infrastruktury jeden brakujący nawias może zatrzymać wdrożenie całego centrum danych.

Zadanie 3.1. Detektyw składni

Spójrz na poniższy fragment kodu. Zawiera on **trzy krytyczne błędy**. Spróbuj je znaleźć, a następnie sprawdź wyjaśnienia poniżej.

LISTING 3.1. Przykładowy kod do odnajdywania błędów

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "~> 2.0"
    }
  }

provider "local" {
}

resource local_file "example"
  filename = "test.txt"
  content  = "Uczę się HCL"
}
```

Wyjaśnienie błędów (Twoja nawigacja po HCL):

1. **Brak klamry zamykającej w bloku `terraform`:** blok `terraform { ... }` musi być domknięty przed rozpoczęciem bloku `provider`. W powyższym przykładzie klamra po `version = "~> 2.0"` zamyka tylko `local`, a brakuje klamry zamykającej cały blok `terraform`.

2. **Brak cudzysłowu w typie zasobu:** prawidłowy zapis to `resource "local_file" "example"`. W HCL zarówno **typ zasobu**, jak i jego **nazwa logiczna** muszą być ujęte w cudzysłowy.
3. **Brak klamry otwierającej blok zasobu:** po definicji `resource "local_file" "example"` musi pojawić się klamra {, która otwiera listę argumentów zasobu.

Jeśli nie udało Ci się wyłapać wszystkich błędów, nie przejmuj się! Nowoczesne edytory kodu, takie jak Visual Studio Code, oferują ogromne wsparcie — automatycznie kolorują pasujące do siebie klamry, formatują tekst, a nawet podpowiadają brakujące elementy składni w czasie rzeczywistym. Dodatkowo, w dalszych częściach książki omówimy zaawansowane techniki testowania kodu oraz dedykowane narzędzia do automatycznego wyłapywania błędów składniowych.

Zadanie 3.2. Pytania kontrolne (z odpowiedziami)

1. **Czym różni się nazwa typu zasobu od nazwy logicznej?**
   - **Odpowiedź:** typ zasobu (np. `local_file`) to stała nazwa zdefiniowana przez dostawcę, mówiąca Terraformowi, co ma stworzyć. Nazwa logiczna (np. `hello`) jest wymyślona przez Ciebie i służy do odwoływania się do tego konkretnego zasobu wewnątrz Twojego kodu.
2. **Dlaczego Terraform jest nazywany językiem deklaratywnym?**
   - **Odpowiedź:** ponieważ w pliku *.tf* opisujemy **stan docelowy** (np. „ten plik ma istnieć"), a nie sekwencję kroków technicznych do jego wykonania. To Terraform decyduje, czy musi plik stworzyć od zera, czy jedynie go zaktualizować.
3. **Co się stanie, jeśli usunę blok `provider`, ale zostawię blok `resource`?**
   - **Odpowiedź:** Terraform zgłosi błąd podczas inicjalizacji (`init`), ponieważ bez konfiguracji dostawcy nie będzie wiedział, jaką wtyczkę pobrać z rejestru, aby obsłużyć dany typ zasobu.

Gotowy na praktykę? Skoro wiesz już, na co uważać w składni i jak wspiera Cię technologia, czas na podrozdział 3.3, gdzie uruchomisz swój pierwszy realny projekt demonstrujący cały *life cycle*: `init`, `plan`, `apply`, `destroy`. Przygotuj terminal — za chwilę przetrenujesz Terraform Lifecycle.

3.3. Lab 01: podstawy cyklu życia Terraform (The Terraform Core Lifecycle)

Opanowanie pracy z Terraformem polega przede wszystkim na zrozumieniu powtarzalnej i przewidywalnej sekwencji komend. **Terraform Core Lifecycle** to proces, który będziesz powtarzać przy każdym projekcie — od tworzenia prostych plików tekstowych po zarządzanie ogromnymi klastrami w chmurze.

Fundament stanowią cztery polecenia:

- `init`: przygotowuje katalog roboczy i pobiera niezbędne wtyczki dostawców (providerów);

- `plan`: Twój „bezpiecznik” — pokazuje dokładnie, jakie operacje Terraform zamierza wykonać, zanim zostaną wprowadzone jakiekolwiek realne zmiany;
- `apply`: właściwe wykonanie zmian i powołanie infrastruktury do życia;
- `destroy`: bezpieczne i całkowite usunięcie wszystkiego, co zostało wcześniej stworzone.

Cel laboratorium

Nauczysz się fundamentów cyklu życia poprzez utworzenie prostego pliku tekstowego na Twoim komputerze za pomocą Terraforma uruchomionego w kontenerze Dockera.

Krok 1. Przygotowanie struktury projektu

Zakładamy, że folder główny `terraform-course-lab` został już utworzony podczas konfiguracji środowiska. Teraz przygotujemy w nim dedykowaną przestrzeń na to ćwiczenie:

Dla macOS/Linux:

```
cd ~
cd terraform-course-lab
mkdir terraform-demo
cd terraform-demo
```

Dla systemu Windows (wiersz poleceń):

```
cd /d %USERPROFILE%
cd terraform-course-lab
mkdir terraform-demo
cd terraform-demo
```

Używając edytora kodu (np. VS Code lub Notepad++), utwórz w tym folderze plik *main.tf* (rysunek 3.1) o treści pokazanej w listingu 3.2.

RYSUNEK 3.1. Plik kodu do Lab 01 w Visual Code

LISTING 3.2. Kod do Lab 01 — tworzenie pliku lokalnego

```
# main.tf
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "~> 2.0"
    }
  }
```

```
}

provider "local" {}

resource "local_file" "hello" {
  filename = "hello.txt"
  content  = "Hello Terraform!"
}
```

Krok 2. Wykonanie cyklu życia

Zanim zaczniesz, upewnij się, że **Docker jest uruchomiony** (najłatwiej sprawdzisz to, szukając ikony wieloryba w pasku systemowym). Będziemy korzystać z parametru `-v $(pwd):/app` (lub `%cd%` w systemie Windows), aby kontener „widział" Twój kod.

1. **Inicjalizacja (`init`):** to polecenie pobierze wtyczkę `local` z oficjalnego rejestru.

   Dla macOS/Linux:

   ```
   docker run -it --rm -v $(pwd):/app -w /app hashicorp/terraform:latest init
   ```

   Dla Windows (wiersz poleceń):

   ```
   docker run -it --rm -v %cd%:/app -w /app hashicorp/terraform:latest init
   ```

2. **Planowanie zmian (`plan`):** użyj tej samej komendy, zamieniając jedynie końcówkę `init` na `plan`.

   Oczekiwany wynik: Terraform powinien wyświetlić komunikat:

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

3. **Wdrażanie (`apply`):** aby Terraform nie pytał nas o ręczne potwierdzenie, dodajemy flagę `-auto-approve`.

   Polecenie: zamień `plan` na `apply -auto-approve`.

   Dla macOS/Linux:

   ```
   docker run -it --rm -v $(pwd):/app -w /app hashicorp/terraform:latest apply
   ↪-auto-approve
   ```

   Dla Windows (wiersz poleceń):

   ```
   docker run -it --rm -v %cd%:/app -w /app hashicorp/terraform:latest apply
   ↪-auto-approve
   ```

Krok 3. Weryfikacja i dowód idempotentności

1. **Weryfikacja:** sprawdź swój folder lokalny (komenda `ls` w macOS/Linux lub `dir` w Windows). Plik `hello.txt` powinien tam być.

   Wyświetl jego treść (komenda `cat hello.txt` w macOS/Linux lub `type hello.txt` w Windows). Zobaczysz napis: `Hello Terraform!`.

2. **Zasada idempotentności:** to jedna z najważniejszych cech Terraforma. Uruchom jeszcze raz komendę `apply -auto-approve`.

   Co się stało? Terraform rozpoznał, że plik o żądanej treści już istnieje. Dzięki temu nie wykonał żadnej akcji (`0 added, 0 changed, 0 destroyed`). To dowód na to, że Terraform dba o stan docelowy, a nie ślepo wykonuje skrypty.

Krok 4. Czyszczenie (destroy)

Dobra praktyka DevOps mówi: „sprzątaj po sobie". Gdy zakończysz naukę, usuń stworzony zasób.

Polecenie: zamień końcówkę komendy na `destroy -auto-approve`.

Dla macOS/Linux:

```
docker run -it --rm -v $(pwd):/app -w /app hashicorp/terraform:latest destroy
↪-auto-approve
```

Dla Windows (wiersz poleceń):

```
docker run -it --rm -v %cd%:/app -w /app hashicorp/terraform:latest destroy
↪-auto-approve
```

Efekt: plik *hello.txt* natychmiast zniknie z Twojego dysku.

Możesz to sprawdzić za pomocą komendy `ls` w macOS/Linux lub `dir` w Windows.

Kluczowa nauka: właśnie przeszedłeś pełny cykl zarządzania infrastrukturą (workflow). Ten sam prosty mechanizm, który stworzył plik tekstowy, posłuży nam w kolejnych rozdziałach do budowy złożonych sieci, baz danych i serwerów w chmurze.

3.4. Lab 02: zarządzanie wieloma zasobami i zależnościami

W rzeczywistych projektach infrastruktura rzadko składa się z jednego elementu. Zazwyczaj zarządzasz dziesiątkami zasobów, które muszą powstawać w ściśle określonej kolejności. Celem tego laboratorium jest zrozumienie, jak Terraform radzi sobie z wieloma zasobami jednocześnie oraz jak tworzyć zależności między nimi.

Zależności w Terraformie (*dependencies*)

Terraform jest inteligentny — zazwyczaj automatycznie buduje tzw. **graf zależności** (*dependency graph*) i wie, co stworzyć najpierw. Jeśli jednak zasoby nie są ze sobą bezpośrednio powiązane w kodzie, a muszą powstać po sobie, stosujemy instrukcję `depends_on`.

Przykłady z życia (AWS): oto jak wyglądałby kod dla scenariuszy, o których wspomnieliśmy wcześniej:

- **VPC i Serwer:** nie możesz uruchomić serwera, dopóki nie istnieje sieć.

LISTING 3.3. Zależności w AWS — VPC i serwery

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  # Terraform sam wykryje zależność, bo używamy ID z aws_vpc
  subnet_id     = aws_subnet.main.id
}
```

- **Baza danych (RDS) i serwer aplikacji:** upewniamy się, że aplikacja nie wystartuje przed bazą.

LISTING 3.4. Zależności w AWS — baza danych i serwer aplikacji

```
resource "aws_db_instance" "database" {
  # konfiguracja bazy...
}

resource "aws_instance" "app_server" {
  ami           = "ami-xxx"
  instance_type = "t2.micro"

  # Jawna zależność: czekaj, aż baza będzie gotowa
  depends_on = [aws_db_instance.database]
}
```

- **Security group i instancja:** grupa bezpieczeństwa (*firewall*) musi istnieć przed maszyną.

LISTING 3.5. Zależności w AWS — security group i EC2

```
resource "aws_security_group" "allow_web" {
  name = "allow_web_traffic"
}

resource "aws_instance" "web" {
  ami             = "ami-xxx"
  instance_type   = "t2.micro"
  # Użycie depends_on zapewnia poprawną kolejność budowania i niszczenia
  depends_on      = [aws_security_group.allow_web]
}
```

W naszym laboratorium dodamy drugi plik: *goodbye.txt*. Drugi plik może powstać dopiero wtedy, gdy pierwszy zostanie pomyślnie utworzony.

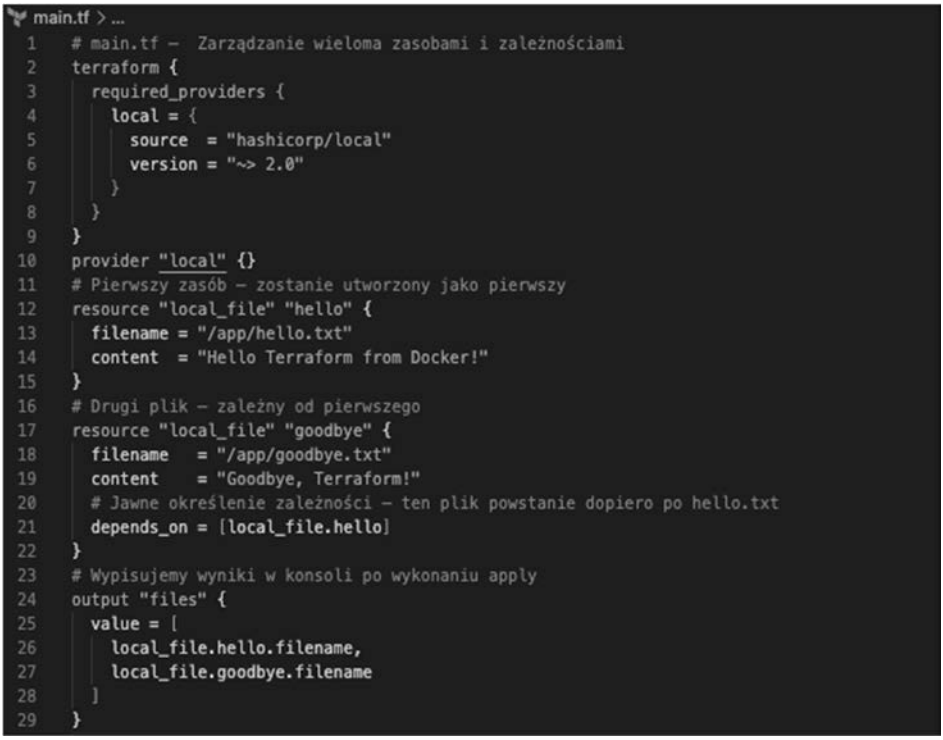
Krok 1. Aktualizacja pliku main.tf

Otwórz swój plik w folderze *terraform-demo* (rysunek 3.2) i zmodyfikuj go zgodnie z poniższym wzorem:

LISTING 3.6. Kod do Lab 02 — zarządzanie wieloma zasobami i zależnościami

```
# main.tf —  Zarządzanie wieloma zasobami i zależnościami
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "~> 2.0"
    }
  }
}
provider "local" {}
# Pierwszy zasób — zostanie utworzony jako pierwszy
resource "local_file" "hello" {
  filename = "/app/hello.txt"
  content  = "Hello Terraform from Docker!"
```

```
}
# Drugi plik — zależny od pierwszego
resource "local_file" "goodbye" {
  filename   = "/app/goodbye.txt"
  content    = "Goodbye, Terraform!"
  # Jawne określenie zależności — ten plik powstanie dopiero po hello.txt
  depends_on = [local_file.hello]
}
# Wypisujemy wyniki w konsoli po wykonaniu apply
output "files" {
  value = [
    local_file.hello.filename,
    local_file.goodbye.filename
  ]
}
```



RYSUNEK 3.2. Widok kodu do Lab 02 w Visual Code

Krok 2. Uruchomienie laboratorium

Wróć do terminala (lub wiersza poleceń Windows) i przejdź do folderu projektu. Następnie zainicjalizuj Terraform w folderze.

Dla macOS/Linux:

```
cd ~
cd  terraform-course-lab/terraform-demo
docker run -it --rm -v $(pwd):/app -w /app hashicorp/terraform:latest init
```

Dla Windows (wiersz poleceń):

```
cd /d %USERPROFILE%
cd  terraform-course-lab/terraform-demo
docker run -it --rm -v %cd%:/app -w /app hashicorp/terraform:latest init
```

Uruchom teraz wdrożenie za pomocą komendy `apply`:

Dla macOS/Linux:

```
docker run -it --rm -v $(pwd):/app -w /app hashicorp/terraform:latest apply
↪--auto-approve
```

Dla Windows (wiersz poleceń):

```
docker run -it --rm -v %cd%:/app -w /app hashicorp/terraform:latest apply
↪--auto-approve
```

Krok 3. Weryfikacja i analiza

Terraform wyświetli w konsoli kolejność działań. Zauważysz, że najpierw tworzony jest zasób `local_file.hello`, a dopiero po jego zakończeniu `local_file.goodbye`.

Sprawdź obecność plików: komenda `ls` w macOS/Linux lub `dir` w Windows.

Dodatkowo możemy wyświetlić zawartość pliku *goodbye.txt*:

Dla macOS/Linux: `cat goodbye.txt`,

Dla Windows: `type goodbye.txt`.

Posprzątajmy po sobie:

Dla macOS/Linux:

```
docker run -it --rm -v $(pwd):/app -w /app hashicorp/terraform:latest destroy
↪--auto-approve
```

Dla Windows (wiersz poleceń):

```
docker run -it --rm -v %cd%:/app -w /app hashicorp/terraform:latest destroy
↪--auto-approve
```

Kluczowe informacje:

- **Automatyzacja:** Terraform zwykle sam buduje graf zależności, jeśli jeden zasób odwołuje się do innego.
- **Instrukcja `depends_on`:** używamy jej, aby wymusić kolejność, gdy nie wynika ona bezpośrednio z referencji w kodzie.
- **Zastosowanie:** jest to niezbędne przy łączeniu usług, np. grupy bezpieczeństwa z maszyną wirtualną.

3.5. Sprawdź swoją wiedzę: cykl życia i zależności

Gratulacje! Masz za sobą dwa pierwsze laboratoria. Zanim przejdziemy do zagadnień związanych ze stanem (*state*), upewnijmy się, że fundamenty są stabilne. Poniższe pytania pomogą Ci usystematyzować wiedzę o tym, jak Terraform „myśli" podczas wdrażania zmian.

Zadanie 3.3. Kolejność działań

Załóżmy, że masz dwa zasoby w Terraformie:

LISTING 3.7. Zależności pomiędzy dwoma plikami

```
resource "local_file" "a" {
  filename = "/app/a.txt"
  content  = "File A"
}

resource "local_file" "b" {
  filename   = "/app/b.txt"
  content    = "File B"
  depends_on = [local_file.a]
}
```

Co się stanie podczas wykonywania komendy `terraform apply`?

A) Terraform utworzy oba pliki jednocześnie, aby zaoszczędzić czas.

B) Terraform utworzy najpierw *b.txt*, a potem *a.txt*.

C) Terraform najpierw utworzy *a.txt*, a dopiero potem *b.txt*.

D) Terraform pominie tworzenie *b.txt*, ponieważ zależy on od innego zasobu.

Zadanie 3.4. Rola argumentu depends_on

Do czego w ekosystemie Terraform służy argument `depends_on` w definicji zasobu?

A) Aby dodać techniczny komentarz do zasobu, widoczny tylko dla programisty.

B) Aby określić, że zasób powinien zostać utworzony dopiero po pomyślnym utworzeniu innego zasobu.

C) Aby przypisać zmienną środowiskową do konkretnego zasobu.

D) Aby zignorować błędy przy usuwaniu zasobów.

Zadanie 3.5. Magiczne słowo — idempotentność

Podczas Lab 01 zauważyłeś, że ponowne uruchomienie `apply` nie stworzyło drugiego pliku. To kluczowa cecha systemów IaC.

Co oznacza, że Terraform jest **idempotentny**?

A) Że zawsze usuwa starą infrastrukturę przed utworzeniem nowej.

B) Że wielokrotne uruchomienie tego samego kodu z tymi samymi parametrami da zawsze ten sam wynik, nie zmieniając nic, jeśli stan docelowy jest już osiągnięty.

C) Że można go używać tylko na jednym systemie operacyjnym jednocześnie.

D) Że automatycznie naprawia błędy w kodzie HCL.

Zadanie 3.6. Szybka powtórka z komend

Dopasuj komendę do jej opisu „z życia wziętego”:

1. `terraform init`
2. `terraform plan`

3. `terraform apply`
4. `terraform destroy`

A) „Pokaż mi kosztorys i plan budowy, zanim wbijesz pierwszą łopatę”.
B) „Spakuj narzędzia i przygotuj plac budowy”.
C) „Zrównaj wszystko z ziemią”.
D) „Buduj zgodnie z planem”.

Klucz odpowiedzi i wyjaśnienia:

Zadanie 3.3.

Poprawna odpowiedź: **C.** *Wyjaśnienie: Argument* `depends_on = [local_file.a]` *tworzy tzw. jawną zależność. Wymusza ona na Terraformie, aby wstrzymał się z tworzeniem pliku* b.txt *do momentu, aż operacja tworzenia* a.txt *zakończy się sukcesem.*

Zadanie 3.4.

Poprawna odpowiedź: **B.** *Wyjaśnienie:* `depends_on` *definiuje jawne zależności między zasobami. Dzięki temu Terraform buduje poprawny graf zależności i wie, w jakiej kolejności tworzyć lub usuwać elementy infrastruktury, co jest kluczowe np. przy tworzeniu sieci przed serwerem.*

Zadanie 3.5.

Poprawna odpowiedź: **B.** *Wyjaśnienie: Jeśli Twoim celem jest posiadanie pliku* hello.txt *i on już istnieje w niezmienionej formie, Terraform to rozpozna i nie wykona żadnej akcji podczas* `apply`*. To daje Ci pewność, że Twoje środowisko jest stabilne.*

Zadanie 3.6.

1. Poprawna odpowiedź: **B.**
2. Poprawna odpowiedź: **A.**
3. Poprawna odpowiedź: **D.**
4. Poprawna odpowiedź: **C.**

Czy wiesz, że Terraform buduje w pamięci mapę połączeń między wszystkimi Twoimi zasobami? Dzięki niej wie, że jeśli usuniesz *local_file.a*, to musi najpierw zająć się zasobem *local_file.b*, który od niego zależy. Nazywa się to grafem acyklicznym.

3.6. Lab 03: użycie zmiennych (variables) i wyników (outputs) w Terraformie

W poprzednich zadaniach wpisywaliśmy wartości, takie jak nazwy plików czy ich treść, „na sztywno” (*hardcoding*). W profesjonalnych projektach takie podejście jest niepraktyczne. Wyobraź sobie, że musisz stworzyć sto podobnych serwerów, nie chciałbyś przecież pisać stu osobnych plików *.tf*.

Terraform
W PRAKTYCE

16 gotowych scenariuszy do uruchomienia w każdym systemie

Zarządzaj infrastrukturą jak kodem

Terraform to oprogramowanie typu open source umożliwiające zarządzanie infrastrukturą IT jako kodem (IaC, ang. *infrastructure as code*). Pozwala deklaratywnie zdefiniować zasoby chmurowe — takie jak AWS, Azure czy Google Cloud — a także zasoby lokalne przy użyciu języka HCL, a potem zautomatyzować ich wdrażanie, wersjonowanie i modyfikowanie.

To niezwykle popularne narzędzie doczekało się już wielu opracowań. Unikalną cechą tej publikacji jest wskazanie, jak można stworzyć uniwersalne laboratorium z wykorzystaniem kontenerów Docker i tym samym uniknąć problemów związanych z konfiguracją środowiska pod różnymi systemami (Windows, Mac, Linux). Pracując z tą książką, zrealizujesz szesnaście praktycznych projektów obejmujących zagadnienia od podstaw składni HCL, przez zarządzanie plikiem stanu, aż po zaawansowane testowanie kodu i bezpieczeństwo w duchu DevSecOps.

Mariusz Dworniczak

Doświadczony menedżer techniczny i certyfikowany architekt chmurowy (AWS, Azure, Google Cloud). Od ponad dwudziestu lat zarządza złożonymi programami IT i migracjami infrastruktury w globalnych organizacjach, takich jak IBM, BOX czy Toyota. Wykładowca na Uniwersytecie WSB Merito oraz doświadczony mentor techniczny. Specjalizuje się w automatyzacji infrastruktury (IaC), a także łączeniu twardych kompetencji technicznych z nowoczesnymi metodykami zarządzania projektami. Prywatnie pasjonat technologii blockchain, AI i gry w szachy.

Helion

helion.pl

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-3838-0

9 788328 938380

Cena: 59,90 zł