Idempotentność: skrypt, który wolno uruchomić dwa razy

Poziom: średniozaawansowany **Dotyczy: bash 4+/5+, wszystkie dystrybucje**

Czas: ~18 min

Skrypt wdrożeniowy uruchamiamy rzadko dokładnie raz. Uruchamiamy go, gdy coś nie zadziała za pierwszym razem, gdy chcemy sprawdzić, czy wszystko nadal jest tak, jak trzeba, albo gdy ten sam playbook trafia na maszynę, na której część kroków już wykonano ręcznie. Ta strona pokazuje wzorce, które sprawiają, że drugie uruchomienie jest tak samo bezpieczne jak pierwsze.

Fiszka

ZMIENIA SYSTEM

Sprawdzenie przed działaniem: grep -q ... || echo ... >>

```
linia="PermitRootLogin no"
plik="/etc/ssh/sshd_config"
grep -qxF "$linia" "$plik" || echo "$linia" >> "$plik"
```

Zamiast bezwarunkowo dopisywać linię, najpierw sprawdzamy, czy już tam jest. `-x` wymaga dopasowania całej linii, `-F` traktuje wzorzec dosłownie, nie jako wyrażenie regularne — bez tego druga litera specjalna w `linia` mogłaby dopasować coś innego, niż zamierzaliśmy. Uruchomione dziesięć razy, dopisze linię dokładnie raz.

grep = global regular expression print: szukaj wzorca w tekście

ZMIENIA SYSTEM

Utworzenie katalogu, które nie zawodzi na istniejącym: mkdir -p

```
mkdir -p /var/lib/NazwaUslugi/dane
```

Zwykłe `mkdir` na istniejącym katalogu kończy się błędem i niezerowym kodem wyjścia — pod trybem ścisłym przerwałoby to skrypt przy drugim uruchomieniu. `-p` tworzy też brakujące katalogi nadrzędne i milczy, jeśli katalog docelowy już istnieje.

mkdir = make directory: utwórz katalog

ZMIENIA SYSTEM

Utworzenie albo podmiana dowiązania symbolicznego: ln -sfn

```
ln -sfn /opt/aplikacja/wersje/2.4.1 /opt/aplikacja/aktualna
```

Zwykłe `ln -s` na istniejącym dowiązaniu zwraca błąd „File exists". `-f` nadpisuje istniejący cel, `-n` traktuje istniejące dowiązanie do katalogu jako plik do podmiany, a nie jako katalog, wewnątrz którego trzeba by utworzyć nowe dowiązanie o tej samej nazwie co katalog docelowy — bez `-n` powtórne uruchomienie tworzyłoby zagnieżdżone dowiązania zamiast podmieniać istniejące.

ln = link: utwórz dowiązanie do pliku, twarde albo symboliczne z flagą -s

ZMIENIA SYSTEM

Utworzenie konta tylko wtedy, gdy nie istnieje

```
id nazwa_usera &>/dev/null || useradd --system --shell /usr/sbin/nologin nazwa_usera
```

`id` na nieistniejącym koncie zwraca kod niezerowy, co uruchamia `useradd` po prawej stronie `||`. Uruchomione drugi raz, `id` znajdzie konto i `useradd` się nie wykona — bez tego drugie uruchomienie zakończyłoby się błędem `user already exists`.

id = pokaż użytkownika, jego numer UID oraz grupy, do których należy

ZMIENIA SYSTEM

Włączenie i uruchomienie usługi w jednym kroku: systemctl enable --now

```
systemctl enable --now NazwaUslugi
```

`enable --now` włącza autostart usługi i uruchamia ją natychmiast, jednym poleceniem. Wywołane na usłudze już włączonej i uruchomionej po prostu nic nie zmienia i kończy się sukcesem — nie trzeba osobno sprawdzać stanu przed wywołaniem.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Instalacja pliku z ustawieniem uprawnień w jednej operacji: install -m

```
install -m 0640 -o root -g NazwaUslugi SciezkaDoPliku /etc/NazwaUslugi/config.yaml
```

`install` kopiuje plik i od razu nadaje mu wskazane uprawnienia (`-m`), właściciela (`-o`) i grupę (`-g`) w jednej operacji atomowej. Odpowiednik przez `cp` i `chmod` osobno zostawia krótkie okno, w którym plik istnieje z domyślnymi, zwykle zbyt szerokimi uprawnieniami — nieistotne dla pliku tekstowego czytanego przez nas samych, istotne dla pliku z hasłem czy kluczem prywatnym.

install = install: kopiuje pliki i tworzy katalogi z jednoczesnym ustawieniem uprawnień

ZMIENIA SYSTEM

Tryb testowy bez wprowadzania zmian: --dry-run

```
#!/bin/bash
dry_run=0
[[ "${1:-}" == "--dry-run" ]] && dry_run=1

wykonaj() {
    if (( dry_run )); then
        echo "[dry-run] wykonałbym: $*"
    else
        "$@"
    fi
}

wykonaj mkdir -p /var/lib/NazwaUslugi
wykonaj systemctl enable --now NazwaUslugi
```

Funkcja pośrednicząca `wykonaj` albo faktycznie uruchamia komendę, albo tylko ją wypisuje, w zależności od flagi. Pozwala uruchomić cały skrypt na produkcji „na sucho" i przeczytać, co

by zrobił, zanim faktycznie coś zmieni.

Przykład:

```
[dry-run] wykonałbym: mkdir -p /var/lib/NazwaUslugi
[dry-run] wykonałbym: systemctl enable --now NazwaUslugi
```

□□ ZMIENIA SYSTEM

Kopia zapasowa pliku przed modyfikacją, ze znacznikiem czasu

```
sciezka="/etc/NazwaUslugi/config.yaml"
cp -a "$sciezka" "${sciezka}.$(date +%Y%m%d-%H%M%S).bak"
```

Znacznik czasu w nazwie kopii zapasowej sprawia, że kolejne uruchomienia skryptu nie nadpisują wzajemnie swoich kopii — każde wywołanie zostawia osobny, chronologicznie uporządkowany ślad tego, jak plik wyglądał tuż przed zmianą.

□□ **cp** = copy: skopiuj

□□ ZMIENIA SYSTEM

Sprawdzenie stanu docelowego na końcu skryptu

```
if systemctl is-active --quiet NazwaUslugi && [[ -f /etc/NazwaUslugi/config.yaml ]];
then
    echo "stan docelowy osiągnięty"
else
    echo "stan docelowy NIE został osiągnięty" >&2
    exit 1
fi
```

Ostatnim krokiem skryptu wdrożeniowego jest sprawdzenie, czy system faktycznie znajduje się w oczekiwanym stanie — nie zakładamy tego na podstawie tego, że wcześniejsze komendy się powiodły, tylko weryfikujemy fakt bezpośrednio, tak jak zrobiłby to ktoś, kto loguje się na serwer po raz pierwszy.

□□ **exit** = zakończ bieżącą powłokę albo skrypt z podanym kodem wyjścia

◫◫ TYLKO ODCZYT

Wszystkie wzorce idempotentności w jednym zestawieniu

Operacja	Wariant nieidempotentny	Wariant idempotentny
Katalog	`mkdir NazwaKatalogu`	`mkdir -p NazwaKatalogu`
Dowiązanie symboliczne	`ln -s cel dowiazanie`	`ln -sfn cel dowiazanie`
Konto systemowe	`useradd nazwa_usera`	`id nazwa_usera &>/dev/null || useradd -r nazwa_usera`
Usługa	`systemctl start` + `systemctl enable` osobno	`systemctl enable --now`
Linia w pliku konfiguracyjnym	`echo linia >> plik`	`grep -qxF linia plik || echo linia >> plik`
Kopiowanie z uprawnieniami	`cp plik cel; chmod 640 cel`	`install -m 0640 plik cel`

Uważaj

- `grep -q` bez `-F` na wzorcu zawierającym znaki specjalne wyrażeń regularnych (kropka, nawiasy kwadratowe) może dopasować więcej albo mniej, niż zamierzaliśmy — `-F` wymuszamy odruchowo przy porównaniu dosłownej linii.
- Idempotentność sprawdzenia „czy usługa istnieje" nie jest tym samym co sprawdzenie „czy usługa ma poprawną konfigurację" — skrypt może być idempotentny w sensie „nie wywali się przy drugim uruchomieniu" i jednocześnie nie naprawiać configu zmienionego ręcznie po fakcie.
- Kopia zapasowa ze znacznikiem czasu bez limitu retencji rośnie w nieskończoność przy częstych uruchomieniach — w skryptach cyklicznych dodajemy sprzątanie starszych kopii, tak jak przy logach.

Z życia wzięte

Z ŻYCIA WZIĘTE

Wtorek, dzień po Patch Tuesday, 10:00. Tomek ponownie uruchamia na app01 skrypt wdrożeniowy, który za pierwszym razem padł w połowie przez chwilową awarię sieci. Drugie uruchomienie wywraca się na `useradd: user 'nazwa_usera' already exists`, bo konto zdążyło powstać przy pierwszej, przerwanej próbie. Dopiero `id nazwa_usera &>/dev/null || useradd ...` pozwala wznowić wdrożenie od tego samego skryptu, bez ręcznego śledzenia, który krok już wykonano.

Po co to istnieje

Pojęcie: idempotentność. Właściwość operacji polegająca na tym, że wykonanie jej wielokrotnie daje dokładnie taki sam efekt końcowy jak wykonanie jej raz. `mkdir -p /var/lib/dane` uruchomione trzy razy z rzędu zostawia dokładnie jeden katalog `/var/lib/dane` — tak samo jak uruchomione raz. `mkdir /var/lib/dane` bez `-p` uruchomione drugi raz kończy się błędem, mimo że katalog jest dokładnie w takim stanie, w jakim miał być.

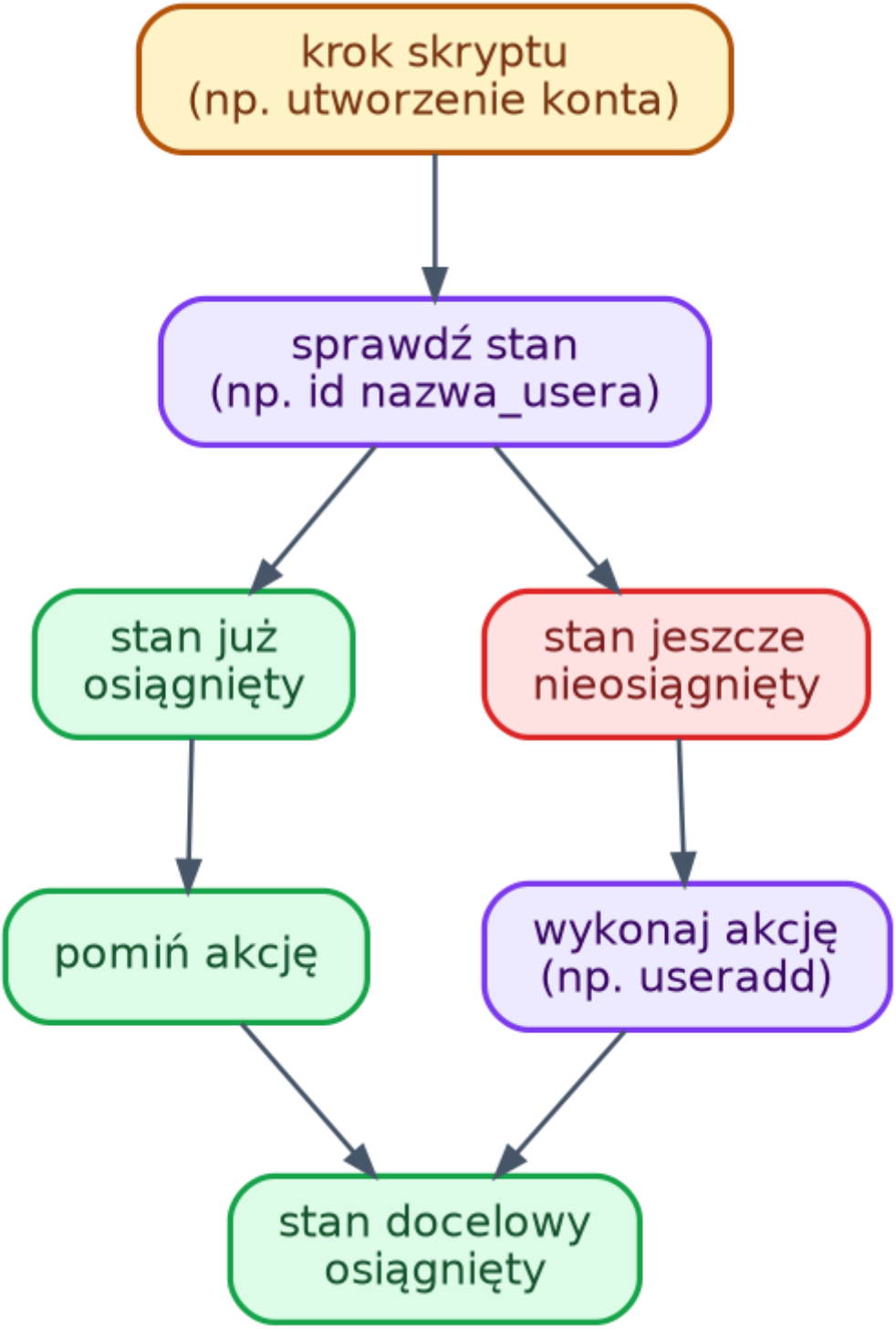
Wymagamy tej cechy od każdego skryptu wdrożeniowego z jednego praktycznego powodu: skrypty uruchamiane raz i tylko raz istnieją głównie w wyobraźni. W rzeczywistości ten sam skrypt trafia ponownie na tę samą maszynę, gdy poprzednie uruchomienie padło w połowie i trzeba je powtórzyć od nowa, gdy playbook konfiguracyjny jest częścią większej automatyzacji uruchamianej co noc niezależnie od tego, czy coś się zmieniło, albo gdy ktoś ręcznie poprawił jeden wiersz

konfiguracji i uruchamia skrypt ponownie, żeby upewnić się, że reszta systemu nadal jest zgodna z oczekiwaniami. Skrypt, który przy drugim uruchomieniu wywala się na `useradd: user already exists`, wymusza na administratorze ręczne śledzenie, który krok już wykonano — dokładnie to, co automatyzacja miała wyeliminować.

Jak to działa

Rozwiń: jak to działa pod spodem

Wzorzec ogólny: sprawdź, potem działaj. Prawie każdy wzorzec idempotentności sprowadza się do tej samej struktury: `warunek_sprawdzający || akcja` albo `if ! warunek_sprawdzający; then akcja; fi`. Różnica między poszczególnymi przykładami w Fiszce to tylko to, czym jest „warunek sprawdzający" dla danego typu zasobu — istnienie pliku, istnienie linii w pliku, istnienie konta, stan usługi. Poznanie tego wzorca raz pozwala rozpoznać go i zastosować do dowolnego nowego typu zasobu, którego dotąd nie było w tym zestawieniu.



Ten sam schemat sprawdź-potem-działaj dla dowolnego typu zasobu: obie gałęzie prowadzą do tego samego stanu docelowego. (kliknięcie otwiera pełny rozmiar)

Dlaczego niektóre komendy są idempotentne z natury, a inne nie. Część komend uniksowych została zaprojektowana z myślą o powtarzalności — `touch` na istniejącym pliku

tylko odświeża znacznik czasu i nigdy nie zwraca błędu, `mkdir -p` i `ln -sfn` mają flagi właśnie po to. Inne komendy — `useradd`, zwykłe `mkdir`, `ln -s` bez flag — zostały zaprojektowane pod kątem jednorazowego wykonania i traktują powtórne wywołanie jako błąd użytkownika, bo z ich perspektywy zaraportowanie „to już istnieje" jest bardziej pomocne niż ciche zignorowanie. Skrypt idempotentny musi więc wiedzieć, które komendy są „bezpieczne z natury", a które wymagają opakowania w sprawdzenie.

Sprawdzenie stanu docelowego jako ostatni krok. Skrypt, który wykonuje same idempotentne kroki, ale nigdy nie weryfikuje wyniku na końcu, wciąż może zakończyć się fałszywym sukcesem — na przykład gdy `systemctl enable --now` zwróci sukces, ale usługa i tak nie wstanie z powodu błędu w konfiguracji wykrywanego dopiero przy starcie procesu. Zamknięcie skryptu jawną weryfikacją stanu końcowego (usługa aktywna, plik istnieje, port nasłuchuje) sprawia, że kod wyjścia skryptu naprawdę odzwierciedla to, czy system jest w oczekiwanym stanie — nie tylko to, że żadna komenda po drodze nie zwróciła błędu.

Porównanie z podejściem deklaratywnym. Narzędzia takie jak Ansible czy Puppet odwracają ten model: zamiast pisać krok po kroku, jak doprowadzić system do stanu docelowego (podejście imperatywne, dokładnie to, co robimy w bashu), opisujemy sam stan docelowy („ten pakiet ma być zainstalowany", „ta usługa ma być włączona"), a silnik narzędzia sam decyduje, czy i jaką akcję wykonać, żeby ten stan osiągnąć — sprawdzenie „czy trzeba coś zrobić" jest tam wbudowane w każdy moduł z góry, więc idempotentność dostajemy bez pisania jej ręcznie. Bash nie ma takiego silnika, więc odtwarzamy tę samą własność ręcznie, wzorcami z tej strony — świadomie, komenda po komendzie.

Krok po kroku

1. Piszemy funkcję dodającą linię do pliku konfiguracyjnego dokładnie raz.

```
dodaj_jesli_brak() {
    local linia="$1"
    local plik="$2"
    grep -qxF "$linia" "$plik" 2>/dev/null || echo "$linia" >> "$plik"
}
```

2. Wywołujemy ją dwukrotnie i sprawdzamy wynik.

```
touch /tmp/test.conf
dodaj_jesli_brak "opcja=wlaczona" /tmp/test.conf
dodaj_jesli_brak "opcja=wlaczona" /tmp/test.conf
cat /tmp/test.conf
```

```
wc -l < /tmp/test.conf
```

```
opcja=wlaczona
1
```

3. Zamykamy skrypt weryfikacją stanu docelowego.

```
grep -qxF "opcja=wlaczona" /tmp/test.conf && echo "stan docelowy potwierdzony"
```

Częste błędy

Dopisywanie linii bez sprawdzenia, czy już istnieje.

```
echo "PermitRootLogin no" >> /etc/ssh/sshd_config
```

Uruchomione trzy razy, zostawia trzy identyczne linie w pliku. Dla `sshd` to nieszkodliwe (ostatnia wygrywa), ale dla wielu innych plików konfiguracyjnych zduplikowane wpisy albo powodują błąd parsowania, albo zmieniają znaczenie (na przykład zduplikowane reguły w pliku `crontab`, które uruchomią zadanie dwa razy zamiast raz).

`useradd` bez sprawdzenia istnienia konta.

```
useradd --system nazwa_usera
```

```
useradd: user 'nazwa_usera' already exists
```

Pod trybem ścisłym ten błąd przerywa cały skrypt przy drugim uruchomieniu, mimo że stan systemu (konto istnieje) jest dokładnie taki, jaki miał być osiągnięty. Naprawa: `id nazwa_usera &>/dev/null || useradd --system nazwa_usera`.

Kopia zapasowa bez znacznika czasu nadpisywana przy każdym uruchomieniu.

```
cp /etc/NazwaUslugi/config.yaml /etc/NazwaUslugi/config.yaml.bak
```

Druga awaria z rzędu nadpisuje jedyną kopię zapasową kopią z poprzedniej, już wadliwej wersji — w chwili, gdy kopia zapasowa jest najbardziej potrzebna, okazuje się bezużyteczna. Naprawa: znacznik czasu w nazwie, jak w Fiszce wyżej.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — dwukrotne uruchomienie tego samego skryptu

```
cd /tmp
cat > wdroz.sh <<'EOF'
#!/bin/bash
set -euo pipefail
mkdir -p /tmp/cw_idempotencja/dane
grep -qxF "tryb=produkcyjny" /tmp/cw_idempotencja/config 2>/dev/null || \
    echo "tryb=produkcyjny" >> /tmp/cw_idempotencja/config
echo "gotowe, kod: $?"
EOF
chmod +x wdroz.sh
./wdroz.sh
./wdroz.sh
./wdroz.sh
wc -l < /tmp/cw_idempotencja/config
```

Ile linii ma plik `config` po trzech uruchomieniach?

Odpowiedź

Jedną. `mkdir -p` nie zawodzi na istniejącym katalogu, a `grep -qxF` przed każdym dopisaniem sprawdza, czy linia już istnieje — trzecie i kolejne uruchomienie nic nie dopisuje, bo warunek po lewej stronie `||` jest już prawdziwy.

Ćwiczenie 2 — tryb dry-run

```
cd /tmp
cat > z_dry_run.sh <<'EOF'
#!/bin/bash
set -euo pipefail
dry_run=0
```

```
[[ "${1:-}" == "--dry-run" ]] && dry_run=1

wykonaj() {
    if (( dry_run )); then
        echo "[dry-run] $*"
    else
        "$@"
    fi
}

wykonaj mkdir -p /tmp/cw_idempotencja/nowy_katalog
EOF
chmod +x z_dry_run.sh
./z_dry_run.sh --dry-run
ls /tmp/cw_idempotencja/ 2>/dev/null
```

Odpowiedź

Skrypt wypisze `[dry-run] mkdir -p /tmp/cw_idempotencja/nowy_katalog`, a `ls` nie pokaże katalogu `nowy_katalog` — nic nie zostało faktycznie utworzone, mimo że skrypt przeszedł przez całą logikę tak, jakby to zrobił.

Pytania kontrolne

1. Dlaczego `systemctl enable --now nazwa` jest bardziej idempotentne niż osobne `systemctl start nazwa` i `systemctl enable nazwa`?

Odpowiedź

Obie formy są w istocie tak samo idempotentne — `systemctl start` na już uruchomionej usłudze i `systemctl enable` na już włączonej też kończą się sukcesem bez błędu. Różnica jest praktyczna, nie związana z idempotentnością samą w sobie: jedna komenda zamiast dwóch to mniej miejsc, w których coś może się rozjechać (na przykład skrypt przerwany między `start` a `enable` zostawiłby usługę uruchomioną, ale bez autostartu), i krótszy, czytelniejszy skrypt.

2. Skrypt kopiuje plik przez `cp` i osobno ustawia uprawnienia przez `chmod`. Jakie mikroskopijne, ale realne ryzyko eliminuje zastąpienie tego jedną komendą `install -m`?

Odpowiedź

Między zakończeniem `cp` a wykonaniem `chmod` istnieje krótkie okno czasowe, w którym plik ma domyślne uprawnienia wynikające z `umask`, zwykle szersze niż docelowe. Dla pliku zawierającego dane wrażliwe (hasło, klucz prywatny, token) to okno — nawet ułamek sekundy — jest oknem, w którym inny proces na tej samej maszynie mógłby teoretycznie odczytać plik z niewłaściwymi uprawnieniami. `install -m` nadaje docelowe uprawnienia w tej samej operacji, która tworzy plik, więc takiego okna nie ma.

Powiązane: Obsługa błędów, trap i sprzątanie · Warunki i pętle · Szkielet skryptu produkcyjnego · Jednostki systemd: pliki .service

Źródła: `man 1 mkdir` · `man 1 ln` · `man 1 install` · `man 8 useradd` · `man 1 systemctl` · dokumentacja Ansible, rozdział *Idempotency*

Revision #12
Created 2026-09-25 00:40:53 UTC by Tomasz Konieczny
Updated 2026-10-03 19:10:23 UTC by Tomasz Konieczny