ShellCheck i testowanie skryptów

Poziom: średniozaawansowany **Dotyczy: wszystkie dystrybucje** **Czas: ~20 min**

ShellCheck czyta skrypt tak, jak czytałby go doświadczony administrator, który setki razy widział te same błędy cytowania i podstawień. Ta strona pokazuje instalację, uruchomienie, najczęstsze ostrzeżenia i sposoby na regularne testowanie skryptów, zanim trafią na produkcję.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
CI	Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie
EPEL	Extra Packages for Enterprise Linux: dodatkowe repozytorium pakietów dla RHEL i klonów
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa

Fiszka

ZMIENIA SYSTEM

Instalacja na RHEL/Rocky/Fedora: dnf

```
sudo dnf install -y ShellCheck
```

Pakiet w repozytoriach EPEL na starszych wersjach RHEL/Rocky, w standardowym repozytorium na Fedorze i nowszych Rocky/RHEL.

RHEL = Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa · **EPEL** = Extra Packages for Enterprise Linux: dodatkowe repozytorium pakietów dla RHEL i klonów · **dnf** = Dandified YUM: menedżer pakietów RHEL/Rocky

□□ ZMIENIA SYSTEM

Instalacja na Debian/Ubuntu: apt

```
sudo apt install -y shellcheck
```

□□ **apt** = Advanced Package Tool: menedżer pakietów Debiana i Ubuntu

□□ ZMIENIA SYSTEM

Instalacja na SLES: zypper

```
sudo zypper install -y ShellCheck
```

□□ **SLES** = SUSE Linux Enterprise Server: płatna dystrybucja Linuksa firmy SUSE, konkurent Rocky/RHEL · **zypper** = menedżer pakietów dystrybucji SUSE i openSUSE

□□ OSTROŻNIE · NIEODWRACALNE

Uruchomienie bez instalacji: kontener

```
docker run --rm -v "$PWD:/mnt" koalaman/shellcheck:stable skrypt.sh
```

Przydatne na maszynie, na której nie chcemy doinstalowywać pakietów, albo w pipeline'ie CI korzystającym z obrazów kontenerowych.

□□ **CI** = Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie · **docker** = narzędzie do kontenerów

□□ TYLKO ODCZYT

Sprawdzenie jednego pliku

```
shellcheck skrypt.sh
```

Wypisuje listę ostrzeżeń z numerem linii, kodem `SCxxxx` i wyjaśnieniem, czego dotyczy. Kod wyjścia jest niezerowy, jeśli znaleziono cokolwiek na poziomie ostrzeżenia lub wyżej — dzięki temu `shellcheck` nadaje się do wpięcia w automatyzację jako bramka jakości.

□□ TYLKO ODCZYT

Sprawdzenie wszystkich skryptów w katalogu

```
shellcheck ./*.sh
```

ShellCheck nie ma trybu rekurencyjnego wbudowanego wprost w nazwę pliku — przeszukanie podkatalogów łączymy z `find`:

```
find . -name '*.sh' -exec shellcheck {} +
```

find = przeszukaj drzewo katalogów według nazwy, typu, daty albo rozmiaru

TYLKO ODCZYT

Wybór poziomu istotności: -S

```
shellcheck -S warning skrypt.sh
```

`-S` (*severity*) filtruje wynik do wskazanego poziomu i wyższych: `error`, `warning`, `info`, `style`. Przydatne w CI, gdzie chcemy blokować build tylko na poważniejszych problemach, a styl zostawić jako sugestię, nie twardy błąd.

CI = Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie

TYLKO ODCZYT

Świadome wyłączenie jednego ostrzeżenia: shellcheck disable

```
# shellcheck disable=SC2086
uruchom_z_flagami $flagi_dodatkowe
```

Dyrektywa w komentarzu tuż nad linią wyłącza konkretny kod ostrzeżenia dla tej jednej linii — nie dla całego pliku. Wyłączamy ostrzeżenie **świadomie i z komentarzem wyjaśniającym dlaczego**, nigdy odruchowo, żeby uciszyć narzędzie: rozdzielenie zmiennej na słowa bywa czasem zamierzonym zachowaniem (na przykład zmienna z listą flag do przekazania dalej), ale to wyjątek, nie regułę.

TYLKO ODCZYT

Formatowanie skryptu do jednego stylu: shfmt

```
shfmt -w -i 4 skrypt.sh
```

`shfmt` (osobne narzędzie, nie część ShellCheck) automatycznie ujednolica wcięcia, spacje i układ `case`/`if` w całym pliku — `-w` zapisuje zmiany do pliku, `-i 4` ustawia szerokość wcięcia na cztery spacje. Uruchamiamy go przed `shellcheck`, żeby różnice w formatowaniu nie przesłaniały realnych ostrzeżeń w code review.

TYLKO ODCZYT

Najczęstsze kody ostrzeżeń ShellCheck

Kod	Skrót problemu	Co naprawdę oznacza
`SC2086`	niecytowana zmienna	`$zmienna` bez cudzysłowu przejdzie przez podział na słowa i globbing — prawie zawsze chcemy `"$zmienna"`
`SC2046`	niecytowane podstawienie komendy	`$(polecenie)` bez cudzysłowu rozpada się na słowa tak samo jak niecytowana zmienna
`SC2006`	backticki zamiast `$( )`	`` `polecenie` `` działa, ale nie zagnieżdża się czytelnie i różnie traktuje backslash — `$(...)` jest nowocześniejszym standardem
`SC2164`	`cd` bez sprawdzenia powodzenia	nieudane `cd` zostawia skrypt w nieoczekiwanym katalogu, a kolejne komendy działają na złych plikach
`SC2155`	`declare`/`local` z przypisaniem z podstawienia komendy w jednej linii	maskuje kod wyjścia podstawienia kodem wyjścia `declare`/`local` — dokładnie pułapka opisana na stronie o funkcjach
`SC2034`	zmienna przypisana, ale nigdy nieużyta	zwykle literówka w nazwie przy odczycie, rzadziej zbędny kod
`SC2181`	sprawdzanie `$?` zamiast komendy wprost	`komenda; if [ $? -eq 0 ]` jest kruche wobec wstawienia linii pomiędzy — `if komenda; then` jest odporne
`SC2162`	`read` bez `-r`	bez `-r` backslash w danych wejściowych jest interpretowany jako znak ucieczki zamiast zachowany dosłownie

Uważaj

- `# shellcheck disable=SCxxxx` umieszczony na górze pliku (zamiast tuż nad konkretną linią) wyłącza ten kod dla **całego pliku** — świadomie sięgamy po tę formę tylko wtedy, gdy problem faktycznie dotyczy całego pliku, a nie jednej linii.
- ShellCheck domyślnie zakłada `bash`; skrypt z shebangiem `#!/bin/sh` jest sprawdzany pod kątem POSIX i zgłasza bashizmy jako błędy — to zamierzone zachowanie, nie fałszywy alarm.
- `bash -n` i ShellCheck to dwa różne poziomy sprawdzenia: `bash -n` łapie wyłącznie błędy składniowe, ShellCheck łapie też poprawne składniowo, ale niebezpieczne wzorce.

POSIX = Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

Z życia wzięte

Z ŻYCIA WZIĘTE

Środa, 14:00. Ola dopisuje `shellcheck -S warning` do pipeline'u na ci01, żeby złapać dokładnie ten sam błąd niecytowanej zmiennej, który miesiąc wcześniej skasował niewłaściwy katalog na app01. Pierwszy commit z nowym krokiem blokuje merge request kolegi, bo ShellCheck zgłasza `SC2086` w świeżo dodanym skrypcie backupowym. Poprawka cudzysłowów trafia do repozytorium, zanim skrypt w ogóle dotarł do produkcji.

Po co to istnieje

Pojęcie: ShellCheck. Statyczny analizator kodu powłoki — program, który czyta skrypt bez jego uruchamiania i wykrywa wzorce znane z tego, że regularnie prowadzą do błędów: niecytowane zmienne, pomylone `[` z `[[`, `cd` bez sprawdzenia, czy się powiodło. Różnica względem `bash -n` jest zasadnicza: `bash -n` odpowiada tylko na pytanie „czy to się sparsuje", ShellCheck odpowiada na pytanie „czy to zrobi to, co autor najprawdopodobniej zamierzał". Skrypt może być składniowo bez zarzutu i jednocześnie kasować niewłaściwy katalog przy pustej zmiennej — dokładnie taki przypadek ShellCheck wyłapuje, a `bash -n` nigdy by nie wykrył.

Uruchamiamy go nie dlatego, że nie umiemy pisać poprawnych skryptów, tylko dlatego, że pewne błędy — zwłaszcza cytowanie — są na tyle subtelne i na tyle rzadko widoczne w normalnym użyciu, że wychodzą na jaw dopiero przy konkretnej kombinacji danych wejściowych, miesiące po napisaniu skryptu. Narzędzie, które wyłapuje je w chwili pisania, jest tańsze niż wyłapywanie ich na produkcji.

Jak to działa

Poziomy istotności. ShellCheck klasyfikuje każde znalezisko w jedną z czterech kategorii: `error` (kod niemal na pewno się zepsuje), `warning` (typowe źródło błędów, wymaga uwagi), `info` (nie jest błędem, ale warto wiedzieć), `style` (kwestia czytelności, nie poprawności). Flaga `-S` pozwala odciąć niższe poziomy, gdy interesuje nas tylko to, co faktycznie może zawieść.

Dlaczego świadome wyłączenie, a nie ignorowanie. Dyrektywa `# shellcheck disable=SCxxxx` z komentarzem wyjaśniającym powód jest zupełnie inną praktyką niż globalne wyłączenie sprawdzania danego kodu dla całego projektu. Pierwsza mówi przyszłemu czytelnikowi (często nam samym za rok): „widziałem to ostrzeżenie, przemyślałem je i tu naprawdę chcę tego zachowania". Druga chowa problem tak skutecznie, że nikt nigdy się o nim nie dowie, także tam, gdzie ostrzeżenie faktycznie wskazywało na błąd.

`shellcheck disable` jako dokumentacja decyzji, nie jako obejście. Klasyczny, uzasadniony przypadek to zmienna świadomie niecytowana, bo ma się rozpaść na wiele argumentów:

```
# shellcheck disable=SC2086 -- flagi mają rozdzielić się na osobne argumenty
komenda $flagi_dodatkowe
```

Bez komentarza wyjaśniającego „dlaczego", dyrektywa `disable` jest nieodróżnialna od kogoś, kto po prostu chciał uciszyć narzędzie, nie rozumiejąc problemu.

Krok po kroku

1. Instalujemy ShellCheck i sprawdzamy wersję.

```
sudo dnf install -y ShellCheck
shellcheck --version
```

2. Piszemy skrypt z kilkoma typowymi problemami.

```
cat > wadliwy.sh <<'EOF'
#!/bin/bash
```

```
KATALOG=$1
cd $KATALOG
PLIKI=`ls *.txt`
for p in $PLIKI; do
    echo Przetwarzam $p
done
EOF
```

3. Uruchamiamy ShellCheck i czytamy wynik.

```
shellcheck wadliwy.sh
```

Przykład:

```
In wadliwy.sh line 2:
KATALOG=$1
^-- SC2086 (info): Double quote to prevent globbing and word splitting.

In wadliwy.sh line 3:
cd $KATALOG
^-- SC2164 (warning): Use 'cd ... || exit' or 'cd ... || return' in case cd fails.

In wadliwy.sh line 4:
PLIKI=`ls *.txt`
^-- SC2006 (style): Use $(...) notation instead of legacy backticks `...`.

In wadliwy.sh line 5:
for p in $PLIKI; do
    ^-- SC2086 (info): Double quote to prevent globbing and word splitting.
```

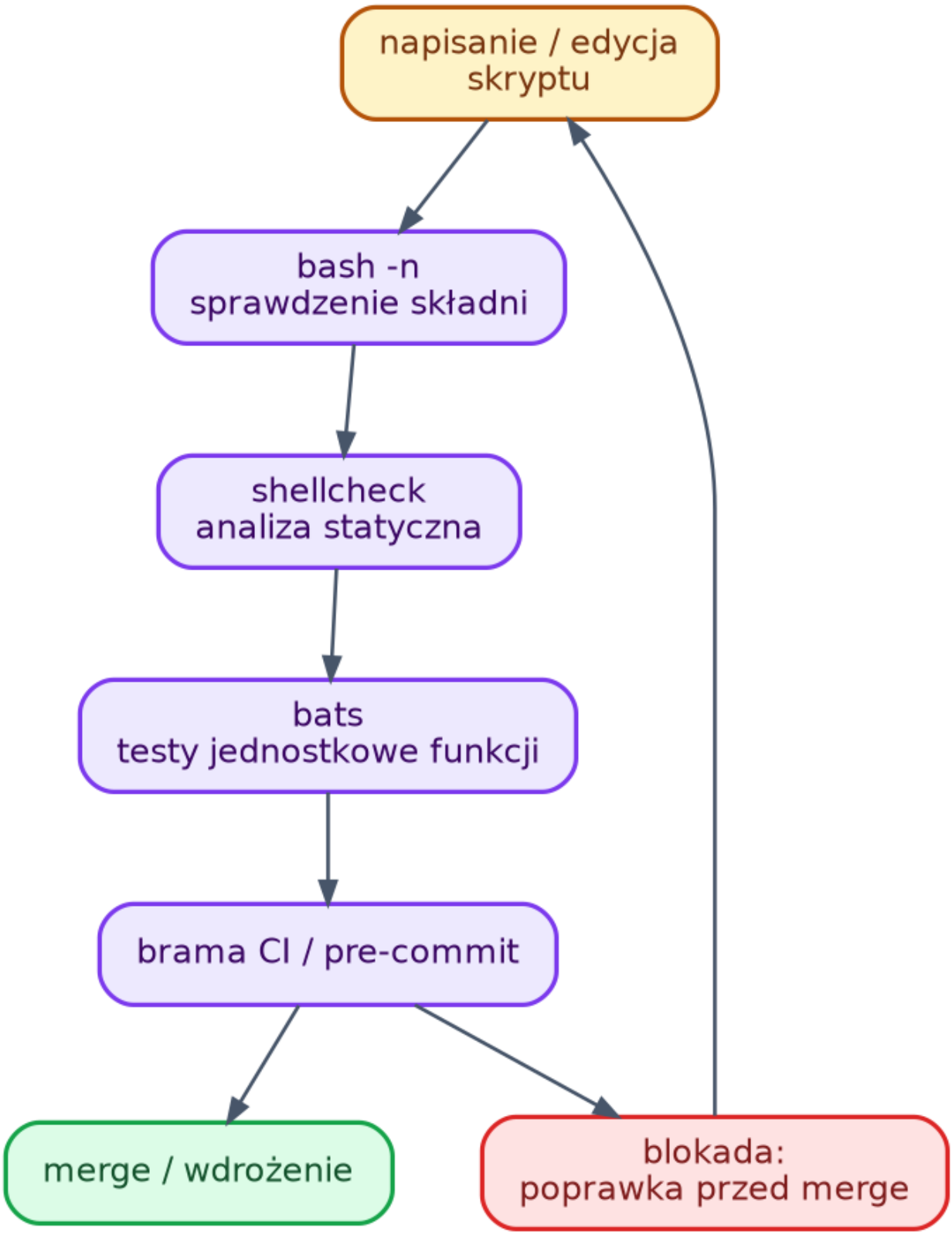
4. Poprawiamy skrypt zgodnie ze wskazówkami i uruchamiamy ponownie.

```
cat > poprawiony.sh <<'EOF'
#!/bin/bash
katalog="$1"
cd "$katalog" || exit 1
shopt -s nullglob
for plik in *.txt; do
    echo "Przetwarzam $plik"
done
EOF
```

```
shellcheck poprawiony.sh
```

Brak wyjścia oznacza brak ostrzeżeń.

Testowanie skryptów



Skrypt przechodzi kolejno przez `bash -n`, ShellCheck i bats, zanim brama CI wpuści go do merge'a; niepowodzenie w dowolnym miejscu odsyła z powrotem do edycji. (kliknięcie otwiera pełny rozmiar)

Sprawdzenie składni: `bash -n`**.** Pierwszy, najtańszy krok — opisany szczegółowo na stronie Pierwszy skrypt: shebang, uprawnienia, uruchomienie. Wykonujemy go zawsze przed czymkolwiek innym, bo test w `bats` na skrypcie z błędem składni po prostu się nie uruchomi.

> **Pojęcie: bats.** *Bash Automated Testing System* — framework do pisania testów jednostkowych dla skryptów powłoki, w którym każdy test to blok `@test` zawierający zwykłe komendy bashowe; test przechodzi, gdy ostatnia komenda w bloku zwróci kod 0.

Instalacja bats-core

```
sudo dnf install -y bats
```

Na dystrybucjach bez pakietu w repozytorium instalujemy z repozytorium `bats-core` na GitHubie przez dołączony `install.sh`.

Prosty test w bats

```
cat > test_powitanie.bats <<'EOF'
#!/usr/bin/env bats

setup() {
    source ./biblioteka.sh
}

@test "powitaj zwraca poprawny tekst" {
    run powitaj "adm_user"
    [ "$status" -eq 0 ]
    [ "$output" = "Cześć, adm_user" ]
}

@test "sprawdz_argument wymaga dokładnie jednego argumentu" {
    run sprawdz_argument
    [ "$status" -ne 0 ]
}
EOF
bats test_powitanie.bats
```

`run` przechwytuje kod wyjścia testowanej komendy do zmiennej `$status` i jej standardowe wyjście do `$output`, nie przerywając wykonania testu nawet jeśli komenda zawiedzie — dzięki temu można sprawdzić zarówno przypadek sukcesu, jak i oczekiwanego niepowodzenia.

Przykład wyniku:

```
 ✓ powitaj zwraca poprawny tekst
 ✓ sprawdz_argument wymaga dokładnie jednego argumentu


2 tests, 0 failures
```

Uruchomienie w kontenerze przed produkcją. Skrypt przeznaczony na konkretną dystrybucję (na przykład zależny od `dnf` albo od konkretnej wersji `systemd`) testujemy w kontenerze z tą samą dystrybucją, zanim trafi na serwer produkcyjny:

```
docker run --rm -v "$PWD:/opt/skrypty" rockylinux:9 /opt/skrypty/wdroz.sh --dry-run
```

Tańsze i szybsze niż odtwarzanie środowiska testowego na fizycznej maszynie, a wyłapuje różnice między dystrybucjami (nazwy pakietów, ścieżki, obecność konkretnych narzędzi) zanim zrobi to produkcja.

Wpięcie w CI i w pre-commit

```
# fragment .gitlab-ci.yml albo podobnego pipeline'u
lint_skrypty:
  stage: test
  script:
    - shellcheck -S warning $(find . -name '*.sh')
```

```
# .git/hooks/pre-commit, z prawem wykonania nadanym przez chmod +x
#!/bin/bash
set -euo pipefail
pliki=$(git diff --cached --name-only --diff-filter=ACM -- '*.sh')
[[ -z "$pliki" ]] && exit 0
shellcheck $pliki
```

Hak `pre-commit` uruchamia ShellCheck tylko na plikach `.sh` faktycznie zmienionych w danym commicie, tuż przed jego utworzeniem — błąd wykryty tutaj nigdy nie trafia do historii repozytorium, w odróżnieniu od błędu wykrytego dopiero w CI po wypchnięciu commita.

Częste błędy

Ignorowanie ostrzeżeń poziomu `info` jako „nieistotnych".

`SC2086` (niecytowana zmienna) bywa zgłaszany jako `info`, ale w praktyce jest jednym z najczęstszych źródeł realnych awarii skryptów — poziom istotności w ShellCheck odzwierciedla pewność narzędzia co do tego, że coś jest błędem, nie wagę konsekwencji, jeśli faktycznie nim jest.

Globalne wyłączenie kodu na poziomie całego zespołu bez dyskusji.

```
# shellcheck disable=SC2086,SC2046,SC2006
```

Trzy najczęstsze i najbardziej pouczające ostrzeżenia wyłączone naraz na górze pliku, żeby „przestały przeszkadzać", odbiera narzędziu sens — od tego momentu ShellCheck przestaje chronić przed dokładnie tymi błędami, które najczęściej się zdarzają.

Testowanie tylko `main`, bez pojedynczych funkcji.

Skrypt napisany zgodnie z konwencją z Funkcje — funkcje na górze, `main "$@"` na dole — pozwala `source`'ować plik w teście bats i wywoływać pojedyncze funkcje w izolacji, bez uruchamiania całego `main`. Test, który zawsze uruchamia cały skrypt od początku do końca, jest wolniejszy i trudniej wskazuje, która dokładnie funkcja zawiodła.

□□ Ćwiczenia

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

Ćwiczenie 1 — od zera do zielonego ShellCheck

```
cd /tmp
cat > cw_shellcheck.sh <<'EOF'
#!/bin/bash
NAZWA=$1
PLIK=/tmp/$NAZWA.txt
touch $PLIK
```

```
echo Utworzono $PLIK
EOF
shellcheck cw_shellcheck.sh
```

Poprawmy skrypt tak, żeby `shellcheck` nie zgłaszał żadnych ostrzeżeń, i uruchommy sprawdzenie ponownie.

Odpowiedź

```
cat > cw_shellcheck.sh <<'EOF'
#!/bin/bash
nazwa="$1"
plik="/tmp/${nazwa}.txt"
touch -- "$plik"
echo "Utworzono $plik"
EOF
shellcheck cw_shellcheck.sh
```

Brak wyjścia potwierdza brak ostrzeżeń — zmienne w cudzysłowach, nazwy pisane małymi literami zgodnie z konwencją zmiennych lokalnych, `--` przed argumentem `touch` na wypadek nazwy zaczynającej się od myślnika.

Ćwiczenie 2 — pierwszy test w bats

```
cd /tmp
cat > lib.sh <<'EOF'
jest_liczba_parzysta() {
    (( $1 % 2 == 0 ))
}
EOF
cat > test_lib.bats <<'EOF'
#!/usr/bin/env bats
setup() { source ./lib.sh; }

@test "4 jest parzyste" {
    run jest_liczba_parzysta 4
    [ "$status" -eq 0 ]
}
```

```
@test "5 nie jest parzyste" {
    run jest_liczba_parzysta 5
    [ "$status" -ne 0 ]
}
EOF
bats test_lib.bats
```

Odpowiedź

```
 ✓ 4 jest parzyste
 ✓ 5 nie jest parzyste

2 tests, 0 failures
```

Oba testy przechodzą, bo `run` łapie kod wyjścia funkcji bez przerywania testu — dzięki temu można jawnie sprawdzić zarówno przypadek prawdziwy, jak i fałszywy, jednym i tym samym mechanizmem.

Pytania kontrolne

1. Dlaczego `bash -n skrypt.sh` przechodzi bez błędu na skrypcie, który ShellCheck oznacza dziesięcioma ostrzeżeniami?

Odpowiedź

`bash -n` sprawdza wyłącznie poprawność składniową — czy plik da się sparsować jako prawidłowy program bashowy. Niecytowane zmienne, backticki zamiast `$( )` czy `cd` bez sprawdzenia powodzenia są w pełni poprawne składniowo — bash je zaakceptuje i uruchomi. ShellCheck sprawdza semantykę i typowe wzorce błędów, nie samą składnię, dlatego wyłapuje znacznie więcej niż `bash -n`.

2. Kiedy uzasadnione jest użycie `# shellcheck disable=SC2086`, a kiedy jest to obejście prawdziwego problemu?

Odpowiedź

Uzasadnione, gdy zmienna ma świadomie rozpaść się na wiele słów — na przykład zmienna zbierająca opcjonalne flagi do przekazania dalej (`komenda $flagi_dodatkowe`), gdzie każda flaga ma trafić jako osobny argument. Obejściem prawdziwego problemu, gdy dotyczy ścieżki do

pliku, nazwy zmiennej z potencjalną spacją czy dowolnej wartości pochodzącej od użytkownika — tam niecytowanie jest realnym ryzykiem, a nie zamierzonym zachowaniem, i dyrektywa `disable` tylko ukrywa błąd zamiast go dokumentować.

Powiązane: Pierwszy skrypt: shebang, uprawnienia, uruchomienie · Tryb ścisły: set -euo pipefail · Funkcje · Szkielet skryptu produkcyjnego

Źródła: `man 1 shellcheck` · strona projektu shellcheck.net (Wiki, lista kodów SC) · `man 1 shfmt` · dokumentacja bats-core

---

Revision #11
Created 2026-09-25 00:40:55 UTC by Tomasz Konieczny
Updated 2026-10-03 19:10:25 UTC by Tomasz Konieczny