Tryb ścisły: set -euo pipefail

Poziom: średniozaawansowany **Dotyczy: bash 4+/5+** **Czas: ~25 min**

`set -euo pipefail` to trzy flagi, które razem zamieniają domyślne, wybaczające zachowanie basha w zachowanie zatrzymujące się przy pierwszym problemie. Ta strona pokazuje, co każda z nich naprawia, gdzie mimo to zawodzą, i dlaczego same w sobie nie są jeszcze obsługą błędów.

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
CI	Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

Fiszka

TYLKO ODCZYT

Włączenie trybu ścisłego na początku skryptu

```
#!/bin/bash
set -euo pipefail
IFS=$'\n\t'
```

Cztery linie, które wklejamy na samym początku niemal każdego skryptu administracyjnego w tej książce, zaraz po shebangu. Reszta tej strony tłumaczy każdą z nich osobno.

IFS = Internal Field Separator: zmienna powłoki mówiąca, po jakim znaku dzielimy tekst na słowa, domyślnie spacja, tabulator i nowa linia

ZMIENIA SYSTEM

Zatrzymanie skryptu przy pierwszym błędzie: set -e

```
set -e
kopiuj_konfiguracje() {
    cp /etc/nginx/nginx.conf /backup/nginx.conf.bak
    echo "kopia zapasowa gotowa"
}
kopiuj_konfiguracje
```

Bez `-e` powłoka domyślnie leci dalej po niepowodzeniu dowolnej komendy — kod wyjścia trafia do `$?`, ale nikt go nie sprawdza, jeśli sami tego nie zrobimy. `set -e` (nazwa długa: `errexit`) sprawia, że skrypt kończy się natychmiast, gdy prosta komenda zwróci kod niezerowy, zamiast kontynuować z fałszywym założeniem, że wszystko poszło dobrze.

cp = copy: skopiuj

OSTROŻNIE · NIEODWRACALNE

Przerwanie przy użyciu niezdefiniowanej zmiennej: set -u

```
set -u
echo "kasuję zawartość: $KATALOG_DOCELOWY"
rm -rf "${KATALOG_DOCELOWY:?}"/*
```

Bez `-u` odwołanie do zmiennej, której nigdy nie ustawiliśmy — literówka w nazwie, brakujący argument, niewczytana konfiguracja — po cichu podstawia pusty napis i skrypt jedzie dalej z fałszywymi danymi. `-u` (nazwa długa: `nounset`) zamienia takie odwołanie w natychmiastowy błąd.

Przykład:

```
skrypt.sh: line 2: KATALOG_DOCELOWY: unbound variable
```

rm = remove: usuń (bez kosza, na zawsze)

TYLKO ODCZYT

Wykrycie niepowodzenia w środku potoku: set -o pipefail

```
set -o pipefail
grep "error" /var/log/aplikacja.log | sort | uniq -c
```

Bez `pipefail` kod wyjścia całego potoku to kod wyjścia wyłącznie **ostatniej** komendy w łańcuchu — jeśli `grep` zawiedzie (bo na przykład plik logu nie istnieje), ale `sort` i `uniq` przetworzą pustą daną i zakończą się sukcesem, potok jako całość zaraportuje sukces. `pipefail` sprawia, że kod wyjścia potoku to pierwszy niezerowy kod z całego łańcucha, a zero tylko wtedy, gdy każdy człon się powiódł.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Podgląd wykonywanych komend: set -x

```
set -x
kopia="/mnt/backup/$(date +%F).tar.gz"
tar -czf "$kopia" /etc
set +x
```

`-x` (nazwa długa: `xtrace`) wypisuje każdą komendę tuż przed jej uruchomieniem, dokładnie w takiej formie, w jakiej trafiła do wykonania po wszystkich podstawieniach — nieoceniona przy diagnozowaniu, co skrypt naprawdę robi, w odróżnieniu od tego, co miał robić naszym zdaniem. Nie jest częścią „trybu ścisłego" w sensie bezpieczeństwa — włączamy ją doraźnie do debugowania, zwykle tylko wokół podejrzanego fragmentu.

tar = tape archive: archiwum plików (kiedyś na taśmę)

TYLKO ODCZYT

Bezpieczniejszy separator słów: IFS=$'\n\t'

```
IFS=$'\n\t'
pliki=(raport\ koncowy.txt "dane 2026.csv")
for p in "${pliki[@]}"; do
    echo "plik: $p"
done
```

Domyślny `IFS` to spacja, tabulator i nowa linia jednocześnie — dlatego niecytowane podstawienie z nazwą pliku zawierającą spację rozpada się na dwa słowa. Ustawienie `IFS` na sam znak nowej linii i tabulator usuwa spację z listy separatorów podziału na słowa, więc pojedyncza spacja w środku wartości przestaje być punktem cięcia tam, gdzie cudzysłowu zabrakło. To nie zastępuje cytowania — to dodatkowa siatka bezpieczeństwa na wypadek, gdy cudzysłów gdzieś pominiemy.

□□ **IFS** = Internal Field Separator: zmienna powłoki mówiąca, po jakim znaku dzielimy tekst na słowa, domyślnie spacja, tabulator i nowa linia

□□ TYLKO ODCZYT

Zaakceptowanie oczekiwanego niepowodzenia: komenda || true

```
set -e
liczba_bledow=$(grep -c "ERROR" /var/log/app.log || true)
echo "liczba błędów: ${liczba_bledow:-0}"
```

`grep -c` zwraca kod 1, gdy nie znajdzie żadnego dopasowania — pod `set -e` to normalnie przerwałoby skrypt, mimo że brak błędów w logu jest dobrą wiadomością, nie awarią. `|| true` jawnie mówi „ten niezerowy kod jest tu oczekiwany i akceptowalny", nie zatrzymuj się. To jedyny poprawny sposób na wpuszczenie zaplanowanego niepowodzenia pod `set -e`, bez wyłączania trybu ścisłego dla całego skryptu.

□□ TYLKO ODCZYT

Wszystkie flagi trybu ścisłego

Flaga	Nazwa długa	Co naprawia
`set -e`	`errexit`	zatrzymuje skrypt przy niezerowym kodzie prostej komendy
`set -u`	`nounset`	zatrzymuje skrypt przy odwołaniu do nieustawionej zmiennej
`set -o pipefail`	`pipefail`	przekazuje niepowodzenie z środka potoku, nie tylko z jego końca
`set -x`	`xtrace`	wypisuje każdą komendę przed wykonaniem — narzędzie do debugowania, nie do bezpieczeństwa

Uważaj

- `set -e` **nie** działa w warunku `if`, po `&&` / `||`, w podpowłoce w niektórych kontekstach ani dla funkcji wywołanej jako warunek — pełne omówienie niżej, w sekcji „Jak to działa".

- Sam `set -euo pipefail` bez jednej świadomie zaplanowanej obsługi błędu (`trap`, `die`, sensowne komunikaty) to skrypt, który się zatrzymuje, ale nie mówi dlaczego — patrz Obsługa błędów, trap i sprzątanie.
- `IFS=$'\n\t'` zmienia zachowanie każdej niecytowanej ekspansji w reszcie skryptu — jeśli jakiś fragment kodu celowo polegał na podziale po spacji, trzeba go poprawić albo lokalnie przywrócić domyślny `IFS`.

IFS = Internal Field Separator: zmienna powłoki mówiąca, po jakim znaku dzielimy tekst na słowa, domyślnie spacja, tabulator i nowa linia

Z życia wzięte

Z ŻYCIA WZIĘTE

Piątek, 16:00. Ola uruchamia na app01 skrypt wdrożeniowy, który nie ma `set -euo pipefail`: krok kopiujący nowy plik konfiguracyjny cicho zawodzi z powodu braku miejsca na dysku, a skrypt mimo to restartuje usługę na starej, częściowo nadpisanej konfiguracji. Dopiero dodanie `set -euo pipefail` na początku skryptu sprawia, że kolejne wdrożenie zatrzymuje się dokładnie na feralnym `cp`, zanim dojdzie do restartu.

Po co to istnieje

Bash został zaprojektowany w latach osiemdziesiątych jako powłoka interaktywna, w której błąd jednej komendy nie powinien uniemożliwiać wpisania następnej — jeśli literówka w `cd` się nie powiedzie, nie chcemy, żeby terminal się zamknął. To rozsądne dla sesji interaktywnej i katastrofalne dla skryptu bez nadzoru: skrypt wdrożeniowy, który nie sprawdza kodu wyjścia `cd`, może wykonać `rm -rf *` w zupełnie niewłaściwym katalogu, bo `cd /nie/istnieje` zawiodło po cichu, a kolejna linia i tak wystartowała z katalogu, w którym akurat się znaleźliśmy.

Pojęcie: tryb ścisły. Potoczna nazwa (nieoficjalna, nie ma jej w dokumentacji basha) kombinacji `set -e`, `set -u` i `set -o pipefail` włączanej na początku skryptu, która zastępuje domyślne „loguj błąd i leć dalej" domyślnym „zatrzymaj się przy pierwszym problemie". Włączamy go świadomie w skryptach nienadzorowanych — uruchamianych z crona, z systemd, z pipeline'u CI — gdzie nikt nie patrzy na ekran w chwili, gdy coś idzie nie tak.

Analogia

Tryb ścisły działa jak **wyłącznik bezpieczeństwa na maszynie przemysłowej**, a nie jak dodatkowy czujnik.

Maszyna bez wyłącznika bezpieczeństwa pracuje dalej, nawet gdy coś zacięło się w środku — operator dowiaduje się o awarii dopiero z efektu końcowego, czasem zepsutego na wielu kolejnych elementach. Wyłącznik bezpieczeństwa zatrzymuje maszynę natychmiast przy pierwszej nieprawidłowości, zanim uszkodzenie rozniesie się dalej po linii produkcyjnej.

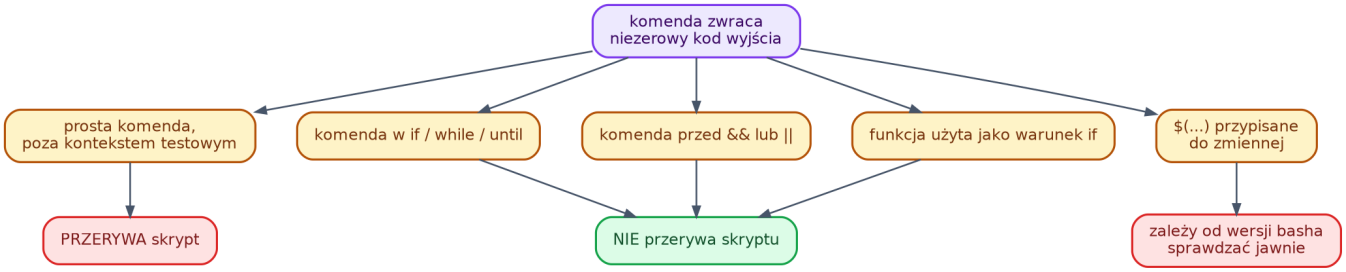
To jest dokładnie to, co robi `set -e` — zatrzymuje wykonanie, zamiast pozwolić błędowi propagować się przez kolejne linie z coraz bardziej fałszywymi założeniami o stanie systemu. Ograniczenie tej analogii jest jednak istotne: wyłącznik bezpieczeństwa **zatrzymuje** maszynę, ale nie naprawia usterki, nie informuje serwisu i nie sprząta rozlanego oleju. Dokładnie tak samo `set -e` samo z siebie nie loguje przyczyny, nie czyści plików tymczasowych i nie powiadamia nikogo — zatrzymuje tylko wykonanie. Resztę trzeba dopisać samodzielnie, i to jest temat kolejnej strony, Obsługa błędów, trap i sprzątanie.

Jak to działa

Rozwiń: jak to działa pod spodem

`set -e` — uczciwie o tym, kiedy NIE zadziała

To jest najczęściej źle rozumiana flaga basha, dlatego zasługuje na dokładne rozłożenie na cztery konkretne sytuacje, w których `errexit` **nie** przerwie skryptu, mimo że komenda zawiodła.



Ten sam niezerowy kod wyjścia przerywa skrypt w jednym kontekście, a w czterech innych zostaje po cichu zignorowany przez `set -e`. (kliknięcie otwiera pełny rozmiar)

1. Komenda w warunku `if`, `while`, `until`.

```
set -e
if grep -q "error" /var/log/app.log; then
    echo "znaleziono błędy"
fi
echo "ten napis wypisze się zawsze"
```

`grep` może zwrócić kod 1 (nic nie znalazł) i skrypt leci dalej bez zająknięcia — to jest zamierzone zachowanie basha: komenda testowana jako warunek nigdy nie uruchamia `errexit`, bo cały sens `if` polega na tym, że oczekujemy obu możliwych wyników.

2. Komenda po lewej stronie `&&` lub `||`.

```
set -e
mkdir /tmp/istniejacy_katalog && echo "utworzono"
echo "ta linia i tak się wykona, nawet jeśli mkdir zawiedzie"
```

Podobnie jak w warunku — lewa strona operatora logicznego jest już „testowana" przez sam operator, więc `set -e` uznaje to za świadomie obsłużone i nie przerywa.

3. Komenda w podpowłoce w niektórych kontekstach oraz w wywołaniu podstawienia komendy przypisanym do zmiennej.

```
set -e
wynik=$(polecenie_ktore_zawodzi)
echo "ten napis czasem się wykona, mimo błędu wyżej"
```

To zależy od wersji basha i od tego, czy `polecenie_ktore_zawodzi` jest ostatnią komendą w podstawieniu — w praktyce najbezpieczniej założyć, że kod wyjścia podstawienia komendy

przypisanego do zmiennej **nie zawsze** przerywa skrypt pod samym `-e`, i sprawdzać go jawnie, gdy ma znaczenie.

4. Funkcja wywołana jako warunek.

```
set -e
sprawdz_dysk() {
    df -h / | awk 'NR==2 {exit ($5+0 > 90)}'
}
if sprawdz_dysk; then
    echo "dysk w normie"
else
    echo "dysk zapełniony ponad normę"
fi
echo "ta linia wykona się zawsze, niezależnie od wyniku funkcji"
```

Cała funkcja użyta jako warunek `if` dziedziczy dokładnie tę samą zasadę co pojedyncza komenda w warunku — nawet jeśli wewnątrz funkcji jakaś komenda zawiedzie, `set -e` nie przerwie skryptu, bo z punktu widzenia `if` cała funkcja jest jednym testowanym wyrażeniem. To zaskakuje najbardziej, bo wygląda na wielolinijkowy kod, a zachowuje się jak prosty test.

Częściowym lekarstwem na przypadki 3 i 4 jest `set -E` (`errtrace`) w połączeniu z `trap ... ERR` — sprawia, że pułapka `ERR` dziedziczy się też do funkcji i podpowłok, co omawiamy na stronie Obsługa błędów, trap i sprzątanie. Samo `-e` tego nie załatwia.

`set -u` — literówka, która przestaje być niewidzialna

```
sciezka_docelowa="/mnt/archiwum"
rm -rf "$SCIEZKA_DOCELOWA"/*
```

Bez `-u` różnica wielkości liter w nazwie zmiennej (`sciezka_docelowa` kontra `SCIEZKA_DOCELOWA`) po cichu podstawia pusty napis, a `rm -rf ""/*` w zależności od wersji basha i cytowania może rozwinąć się w coś znacznie groźniejszego niż zamierzony katalog. `set -u` zamienia każde takie odwołanie w natychmiastowy błąd `unbound variable`, zanim komenda w ogóle wystartuje.

Wyjątek, o którym trzeba pamiętać: `"$@"` i `"$*"` w skrypcie wywołanym bez argumentów są bezpieczne nawet pod `-u` — rozwijają się w zero elementów, nie w błąd. Kłopotem bywają za to zmienne z domyślnymi wartościami zdefiniowanymi warunkowo dalej w skrypcie — jeśli coś

odwołuje się do nich wcześniej, `-u` to wyłapie, co jest właśnie jego zadaniem.

`set -o pipefail` — kod wyjścia potoku, nie tylko ostatniego ogniwa

Zasadę i mechanizm `PIPESTATUS` opisuje szczegółowo strona Kod wyjścia, $?, &&, ||, średnik — tutaj liczy się zastosowanie: bez `pipefail`, `polecenie_ktore_moze_zawiesc | tee /var/log/wynik.log` zawsze zaraportuje sukces, bo `tee` prawie nigdy nie zawodzi, niezależnie od tego, co się stało z lewej strony potoku. `pipefail` sprawia, że taki potok raportuje niepowodzenie, jeśli zawiodło cokolwiek w łańcuchu, nie tylko ostatni człon.

`IFS=$'\n\t'` — co konkretnie naprawia

Domyślny `IFS` to trzy znaki naraz: spacja, tabulator, nowa linia. Ustawienie `IFS=$'\n\t'` usuwa spację z tej listy, zostawiając tylko tabulator i nową linię jako separatory podziału na słowa. Efekt praktyczny: pętla `for p in $lista_plikow` (bez cudzysłowu — nadal błąd sam w sobie, ale częsty w istniejącym kodzie) przestaje rozbijać nazwy plików ze spacjami na osobne słowa, bo spacja przestała być punktem cięcia. To jest siatka bezpieczeństwa na wypadek zapomnianego cudzysłowu — nie licencja na pomijanie cudzysłowów świadomie, bo `IFS` nie chroni przed globbingiem ani przed pustymi zmiennymi, którymi zajmuje się `-u`.

Dlaczego sam `set -e` to nie jest obsługa błędów

`set -e` odpowiada wyłącznie na pytanie „czy kontynuować, czy się zatrzymać". Nie odpowiada na żadne z pytań, które realnie interesują kogoś czytającego log po fakcie: co konkretnie zawiodło, jaki był stan systemu w tym momencie, czy trzeba posprzątać plik tymczasowy, czy ktoś ma zostać powiadomiony. Skrypt z samym `set -e` i bez `trap` kończy się komunikatem basha w stylu `linia 42: polecenie: command not found`, co jest lepsze niż nic, ale wciąż zostawia administratora z koniecznością czytania kodu, żeby zrozumieć kontekst. Prawdziwa obsługa błędów — `trap`, funkcja `die` z sensownym komunikatem, sprzątanie plików tymczasowych — jest tematem kolejnej strony.

Przykład: ten sam skrypt z trybem ścisłym i bez niego

Bez trybu ścisłego, przy pustej zmiennej konfiguracyjnej:

```
#!/bin/bash
# BRAK trybu ścisłego — celowo, dla pokazania różnicy

KATALOG_ROBOCZY=$(cat /etc/myapp/workdir.conf 2>/dev/null)
echo "czyszczę katalog roboczy: $KATALOG_ROBOCZY"
rm -rf "$KATALOG_ROBOCZY"/*
echo "gotowe"
```

Jeśli plik `/etc/myapp/workdir.conf` nie istnieje albo jest pusty, `KATALOG_ROBOCZY` przyjmuje pusty napis, `cat` zgłasza błąd wyciszony przez `2>/dev/null`, a skrypt mimo to leci dalej i wykonuje `rm -rf ""/*`, co w bashu rozwija się względem katalogu bieżącego — w najgorszym razie, uruchomiony jako root z katalogu `/`, usuwa zawartość całego systemu plików. Skrypt kończy się słowem „gotowe", więc z zewnątrz wygląda na sukces.

Z trybem ścisłym:

```
#!/bin/bash
set -euo pipefail
IFS=$'\n\t'

KATALOG_ROBOCZY=$(cat /etc/myapp/workdir.conf)
echo "czyszczę katalog roboczy: $KATALOG_ROBOCZY"
rm -rf "$KATALOG_ROBOCZY"/*
echo "gotowe"
```

Ten sam brakujący plik konfiguracyjny: `cat` zwraca kod niezerowy, `pipefail` i `-e` razem przerywają skrypt na tej właśnie linii, `KATALOG_ROBOCZY` nigdy nie zostaje ustawiona, a `rm -rf` nigdy się nie uruchamia.

```
cat: /etc/myapp/workdir.conf: No such file or directory
```

Skrypt zatrzymuje się z komunikatem basha i niezerowym kodem wyjścia, zanim doszło do jakiejkolwiek szkody — dokładnie to jest różnica, dla której wprowadzamy tryb ścisły, mimo że wymaga dodatkowej uwagi przy pisaniu (świadome `|| true` tam, gdzie niepowodzenie jest

oczekiwane, jak w fiszce wyżej).

□□ Ćwiczenia

> Wszystko poniżej wykonujemy na własnej maszynie, w katalogu `/tmp` — nigdy jako root i nigdy na katalogu `/`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — zobacz różnicę na sucho

```
mkdir -p /tmp/cw_scisly/cel && cd /tmp/cw_scisly

cat > bez_scislego.sh <<'EOF'
#!/bin/bash
KATALOG=$(cat brak_pliku.conf 2>/dev/null)
echo "katalog: [$KATALOG]"
echo "tu wykonałby się rm -rf, pomijamy dla bezpieczeństwa"
echo "gotowe"
EOF

cat > ze_scislym.sh <<'EOF'
#!/bin/bash
set -euo pipefail
KATALOG=$(cat brak_pliku.conf)
echo "katalog: [$KATALOG]"
echo "ta linia się nie wykona"
EOF

chmod +x bez_scislego.sh ze_scislym.sh
echo "=== bez trybu ścisłego ==="
./bez_scislego.sh
echo "kod: $?"
echo "=== z trybem ścisłym ==="
./ze_scislym.sh
echo "kod: $?"
```

Odpowiedź

Pierwszy skrypt wypisze `katalog: []` i dojdzie do końca z kodem `0`, mimo że `cat` zawiódł — dokładnie tak wyglądałby początek katastrofy, gdyby zamiast `echo` stało tam prawdziwe `rm -rf`. Drugi skrypt zatrzyma się na linii z `cat` z komunikatem `No such file or directory` i kodem różnym od zera, nigdy nie docierając do linii, która mogłaby coś skasować.

Ćwiczenie 2 — pułapka `set -e` w warunku

```
cd /tmp/cw_scisly
cat > pulapka.sh <<'EOF'
#!/bin/bash
set -e
if grep -q "TAKIEGONAPISUNAPEWNONIEMA" /etc/hostname; then
    echo "znaleziono"
fi
echo "ta linia się wykonuje mimo set -e"
EOF
chmod +x pulapka.sh
./pulapka.sh
echo "kod: $?"
```

Dlaczego skrypt nie przerywa się, mimo że `grep` zwraca kod 1, a mamy włączone `set -e`?

Odpowiedź

Komenda testowana bezpośrednio jako warunek `if` nigdy nie uruchamia `errexit` — to jest udokumentowane, zamierzone zachowanie basha, nie błąd. `set -e` reaguje tylko na niezerowy kod prostej komendy wykonanej samodzielnie, poza kontekstem testowym. Stąd zasada: tryb ścisły nie zwalnia nas z jawnego przemyślenia, co ma się stać w gałęzi `else`.

Pytania kontrolne

1. Skrypt ma `set -euo pipefail`, a mimo to nie zatrzymuje się, gdy `polecenie1 && polecenie2` — `polecenie1` zawodzi. Dlaczego?

Odpowiedź

Lewa strona `&&` jest, z punktu widzenia `errexit`, komendą testowaną przez operator logiczny — dokładnie tak samo jak komenda w warunku `if`. `set -e` zakłada, że taki układ jest świadomie obsłużony przez samą obecność `&&`/`||`, więc nie przerywa wykonania. Trzeba to sprawdzić jawnie, na przykład zamieniając na pełne `if polecenie1; then polecenie2; fi`, jeśli niepowodzenie `polecenie1` ma realnie zatrzymać skrypt.

2. Po co dopisujemy `|| true` do komendy, skoro właśnie włączyliśmy `set -e`, żeby wyłapywać błędy?

Odpowiedź

Nie każdy niezerowy kod wyjścia jest błędem w sensie biznesowym — `grep` zwracające 1, bo nic nie znalazło, `mkdir -p` na już istniejącym katalogu w starszych wariantach, czy narzędzie diagnostyczne zwracające kod informacyjny to sytuacje, w których chcemy złapać wynik i jechać dalej, a nie przerywać skrypt. `|| true` jawnie i czytelnie oznacza w kodzie „ten konkretny niezerowy kod jest tu oczekiwany", w odróżnieniu od milczącego pominięcia błędu, które wyglądałoby tak samo dla kogoś czytającego kod bez tego komentarza.

Powiązane: Obsługa błędów, trap i sprzątanie · Kod wyjścia, $?, &&, ||, średnik · Cudzysłowy, escapowanie i ekspansja · Funkcje · Szkielet skryptu produkcyjnego

Źródła: `man 1 bash` (sekcja *SHELL BUILTIN COMMANDS* → `set`) · `help set` · POSIX.1-2024, Shell Command Language, 2.8.1 Consequences of Shell Errors

---

Revision #12
Created 2026-09-25 00:40:50 UTC by Tomasz Konieczny
Updated 2026-10-03 19:10:20 UTC by Tomasz Konieczny