Pierwszy skrypt: shebang, uprawnienia, uruchomienie

Poziom: podstawy **Dotyczy: wszystkie dystrybucje (bash 4+/5+)** **Czas: ~15 min**

Skrypt powłoki to plik z listą komend, który wykonujemy jednym poleceniem zamiast wklejać je ręcznie jedna po drugiej. Ta strona pokazuje, jak taki plik oznaczamy, jak go uruchamiamy na trzy różne sposoby i jak sprawdzamy go, zanim cokolwiek zrobi.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
PID	Process ID: numer procesu w systemie
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
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera

Fiszka

ZMIENIA SYSTEM

Zapisanie pierwszego skryptu

```
cat > pierwszy.sh <<'EOF'
#!/bin/bash
echo "Uruchamiamy skrypt jako $(whoami) w katalogu $(pwd)"
EOF
```

Zapisujemy zwykły plik tekstowy zawierający komendy. Here-dokument jest tu tylko wygodnym sposobem zrobienia tego z jednej komendy — w praktyce sięgamy po edytor (`vim`, `nano`) i zapisujemy plik tak jak każdy inny.

□□ **cat** = concatenate: sklej i wypisz pliki

□□ ZMIENIA SYSTEM

Nadanie prawa wykonywania: chmod +x

```
chmod +x pierwszy.sh
```

Bez bitu wykonywalnego jądro odmawia uruchomienia pliku jako programu, nawet gdy zawiera poprawny kod. Komenda przy powodzeniu milczy.

□□ **chmod** = change mode: zmień prawa dostępu

□□ TYLKO ODCZYT

Uruchomienie przez ścieżkę do pliku: ./

```
./pierwszy.sh
```

Kropka i ukośnik mówią powłoce „ten konkretny plik w tym katalogu", a nie „szukaj komendy o tej nazwie w PATH". Powłoka uruchamia nowy proces potomny, jądro czyta pierwszą linię pliku i na jej podstawie decyduje, czym ten plik wykonać.

Przykład:

```
Uruchamiamy skrypt jako adm_user w katalogu /home/adm_user
```

□□ TYLKO ODCZYT

Uruchomienie przez wskazanie interpretera wprost: bash skrypt.sh

```
bash pierwszy.sh
```

Wywołujemy interpreter bezpośrednio i podajemy mu plik jako argument. Bit `x` nie jest wtedy wymagany — to `bash`, a nie jądro, otwiera i czyta plik. Pierwsza linia pliku jest ignorowana jako instrukcja dla jądra, choć jeśli zaczyna się od `#`, `bash` i tak potraktuje ją jak zwykły komentarz.

□□ **bash** = Bourne Again Shell: uruchom powłokę bash albo wskazany w niej skrypt

TYLKO ODCZYT

Wczytanie skryptu do bieżącej powłoki: source

```
source pierwszy.sh
```

Skrót `.` robi dokładnie to samo: `. pierwszy.sh`. Zamiast nowego procesu potomnego komendy wykonują się w **bieżącej** powłoce — zmienne, `cd`, `export` i funkcje zdefiniowane w skrypcie zostają dostępne po jego zakończeniu, w powłoce, z której go wywołaliśmy. To jedyny z trzech sposobów, który potrafi trwale zmienić nasze bieżące środowisko, i dlatego pliki startowe (`~/.bashrc`) są właśnie w ten sposób wczytywane.

source = wykonaj plik w bieżącej powłoce zamiast w nowej, zmienne z niego zostają dostępne dalej

TYLKO ODCZYT

Sprawdzenie składni bez uruchamiania: bash -n

```
bash -n pierwszy.sh
```

Powłoka parsuje plik i sprawdza błędy składniowe, ale nie wykonuje ani jednej linii. Milczenie oznacza plik poprawny składniowo — nie oznacza, że logika w środku robi to, co zamierzaliśmy.

Przykład błędu:

```
pierwszy.sh: line 4: syntax error: unexpected end of file
```

bash = Bourne Again Shell: uruchom powłokę bash albo wskazany w niej skrypt

TYLKO ODCZYT

Śledzenie wykonania krok po kroku: bash -x

```
bash -x pierwszy.sh
```

Przed uruchomieniem każdej komendy powłoka wypisuje ją tak, jak wygląda po wszystkich podstawieniach, poprzedzoną znakiem `+`. Ten sam efekt włączony w środku skryptu daje `set -x`, a `set +x` go wyłącza — przydaje się, gdy chcemy śledzić tylko fragment, a nie cały plik.

Przykład:

```
+ whoami
+ pwd
+ echo 'Uruchamiamy skrypt jako adm_user w katalogu /home/adm_user'
Uruchamiamy skrypt jako adm_user w katalogu /home/adm_user
```

bash = Bourne Again Shell: uruchom powłokę bash albo wskazany w niej skrypt

TYLKO ODCZYT

Zakończenie skryptu z własnym kodem: exit

```
exit 0
```

Kod wyjścia skryptu trafia do zmiennej `$?` powłoki, która go uruchomiła — dokładnie tak samo jak przy zwykłej komendzie. Skrypt bez jawnego `exit` kończy się kodem ostatniej wykonanej linii. Znaczenie poszczególnych wartości opisujemy na stronie Kod wyjścia, $?, &&, ||, średnik.

exit = zakończ bieżącą powłokę albo skrypt z podanym kodem wyjścia

TYLKO ODCZYT

Wszystkie trzy sposoby uruchomienia

Sposób	Nowy proces	Wymaga `chmod +x`	Zmiany przeżywają zakończenie skryptu
`./skrypt.sh`	tak	tak	nie
`bash skrypt.sh`	tak	nie	nie
`source skrypt.sh` (`. skrypt.sh`)	nie	nie	tak

Uważaj

- `source` na skrypcie, który kończy się `exit`, zamknie naszą bieżącą powłokę interaktywną — w środku skryptu przeznaczonego do `source`'owania używamy `return`, nie `exit`.

- Skrypt przekopiowany z Windowsa ma zakończenia linii CRLF, co psuje shebang — objawy i naprawę opisujemy w sekcji „Częste błędy".
- `bash skrypt.sh` uruchamia plik wskazaną powłoką niezależnie od tego, co wpisano w shebangu, więc shebang testujemy tylko przez `./skrypt.sh`.

Z życia wzięte

Z ŻYCIA WZIĘTE

Poniedziałek, 9:00: Tomek przegrywa `wdroz.sh` na app01 z laptopa z Windowsem przez współdzielony folder, nadaje `chmod +x` i dostaje `bad interpreter: No such file or directory` zamiast zwykłego uruchomienia. `cat -A wdroz.sh` pokazuje `^M` na końcu każdej linii, dopisane przez edytor na Windowsie. Dopiero `sed -i 's/\r$//' wdroz.sh` usuwa te znaki i skrypt startuje tak, jak powinien od początku.

Po co to istnieje

Pojęcie: skrypt powłoki. Plik tekstowy zawierający sekwencję komend powłoki, traktowany przez system jak samodzielny program. Różnica między wklejeniem trzech komend do terminala a zapisaniem ich w pliku jest praktyczna, nie teoretyczna: skrypt uruchamiamy tysiąc razy identycznie, wersjonujemy w gicie, wywołujemy z crona o czwartej nad ranem bez naszego udziału i przekazujemy koledze, który nie musi znać kolejności kroków — musi tylko znać nazwę pliku.

To, co odróżnia skrypt od zwykłego pliku tekstowego, to dwie rzeczy: bit wykonywalny w uprawnieniach oraz pierwsza linia, która mówi jądru, jakim programem ten plik zinterpretować. Reszta strony tłumaczy obie.

Analogia

Wyobraźmy sobie różnicę między **tłumaczeniem ustnym na żywo** a **tłumaczeniem pisemnym dołączonym do dokumentu**.

Gdy dyktujemy komendy w terminalu, jesteśmy tłumaczem na żywo: mówimy jedno zdanie, czekamy na efekt, mówimy następne. Jeśli coś pójdzie nie tak w połowie, przerywamy i reagujemy

natychmiast, bo jesteśmy przy tym cały czas.

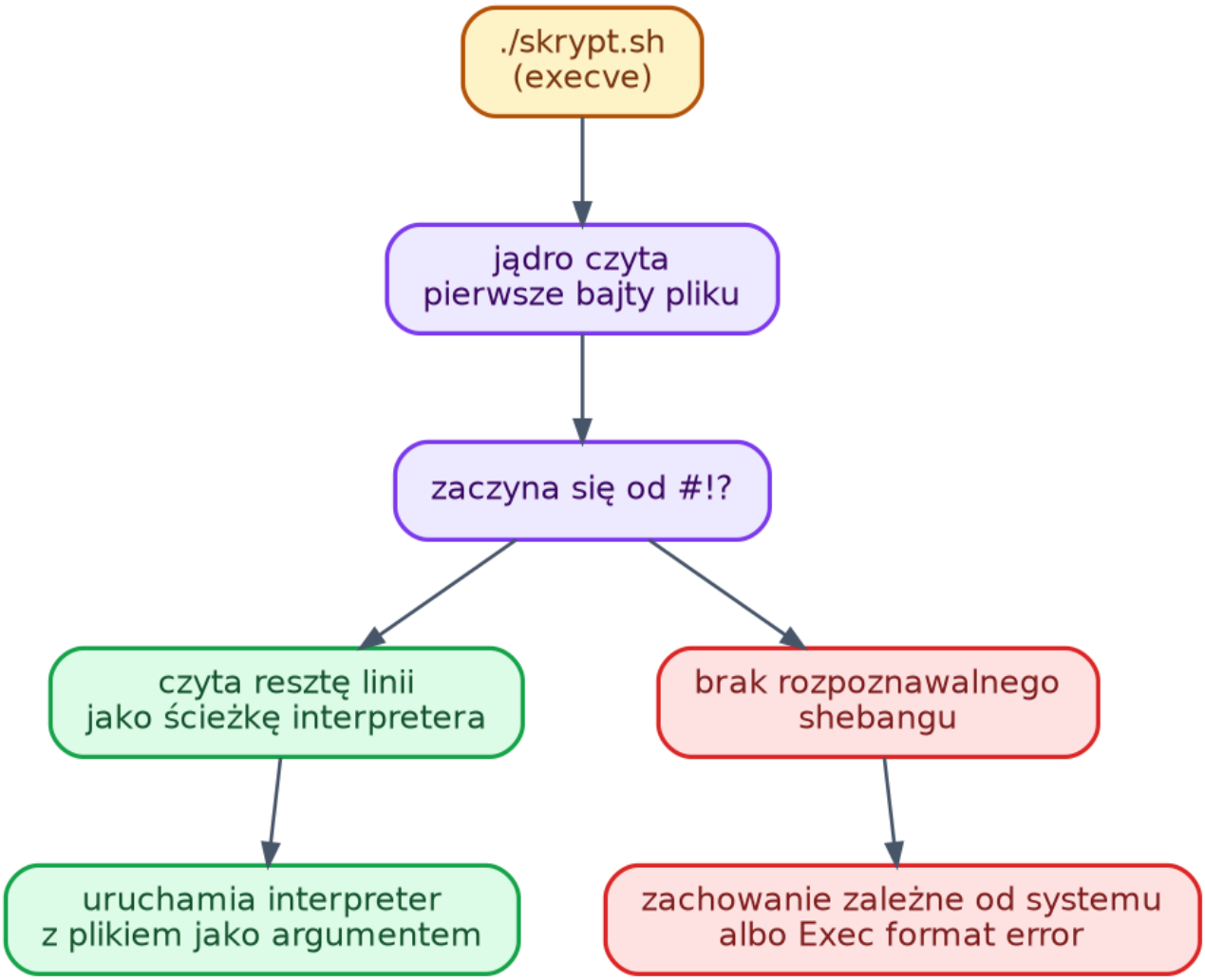
Skrypt to tłumaczenie pisemne — kompletny dokument, który ktoś (albo my sami za miesiąc, albo cron o czwartej nad ranem) odczyta od góry do dołu bez naszej obecności. Dlatego dokument potrzebuje nagłówka mówiącego, w jakim języku jest napisany — to jest dokładnie rola shebangu. Bez niego odbiorca próbowałby czytać go w domyślnym dla siebie języku, co przy prostych zdaniach czasem się udaje, a przy czymkolwiek bardziej złożonym prowadzi do nieporozumień.

Ograniczenie tej analogii jest pouczające: tłumaczenie pisemne nie pyta o potwierdzenie w połowie zdania. Skrypt uruchomiony przez `./` czy `bash` leci dalej po napotkaniu błędu, chyba że jawnie powiemy mu, żeby się zatrzymał — to jest właśnie temat trybu ścisłego, opisanego na stronie Tryb ścisły: set -euo pipefail.

Jak to działa

Rozwiń: jak to działa pod spodem

> **Pojęcie: shebang.** Dwa znaki `#!` na samym początku pliku, po których następuje ścieżka do programu mającego ten plik wykonać. Gdy jądro dostaje polecenie uruchomienia pliku (przez `execve`), sprawdza jego pierwsze bajty. Jeśli to `#!`, jądro nie próbuje wykonać pliku samodzielnie — czyta resztę pierwszej linii jako ścieżkę do interpretera i uruchamia ten interpreter, podając mu nasz plik jako argument. `./skrypt.sh` z shebangiem `#!/bin/bash` zamienia się więc w praktyce w `/bin/bash ./skrypt.sh`.



Jądro czyta pierwsze bajty pliku; jeśli to #!, uruchamia wskazany interpreter z plikiem jako argumentem, inaczej zachowanie zależy od systemu (kliknięcie otwiera pełny rozmiar)

Ten mechanizm tłumaczy, dlaczego shebang działa tylko przy uruchomieniu przez `./skrypt.sh`, a nie przy `bash skrypt.sh` — w drugim przypadku to my już wskazaliśmy interpreter wprost, jądro nigdy nie musiało zaglądać do pierwszej linii pliku.

Trzy warianty shebangu

Shebang	Co robi	Kiedy zawodzi
`#!/bin/bash`	Uruchamia konkretnie bash spod tej ścieżki	Gdy bash jest zainstalowany gdzie indziej (część systemów BSD, niektóre kontenery minimalne)

Shebang	Co robi	Kiedy zawodzi
`#!/usr/bin/env bash`	Prosi `env` o znalezienie `bash` przez `PATH` bieżącego użytkownika	Praktycznie nie zawodzi, dopóki `bash` jest w `PATH` — ale uruchomiona wersja zależy od tego, która znajdzie się pierwsza
`#!/bin/sh`	Uruchamia powłokę zarejestrowaną w systemie jako `sh` — na wielu dystrybucjach to `dash`, nie `bash`	Skrypt używający bashizmów (`[[ ]]`, tablice, `local`) wywali błąd składni albo zachowa się inaczej

`#!/bin/bash` jest wyborem domyślnym w tej książce i w większości skryptów administracyjnych na RHEL/Rocky/Debian/Ubuntu — ścieżka jest stała na tych dystrybucjach od dekad. `#!/usr/bin/env bash` bywa preferowany w projektach przenośnych między systemami, gdzie ścieżka do `bash` nie jest pewna. `#!/bin/sh` wybieramy świadomie, gdy chcemy wymusić przenośność do POSIX i zrezygnować z rozszerzeń basha — inaczej jest to pułapka, bo skrypt napisany z bashizmami, ale opatrzony `#!/bin/sh`, działa na maszynie, gdzie `sh` jest linkiem do `bash`, i wywala się dopiero na maszynie, gdzie `sh` to `dash`.

Gdzie trzymamy skrypty

> **Pojęcie: PATH.** Lista katalogów, w których powłoka szuka programu, gdy wpisujemy jego nazwę bez ścieżki — opisana szerzej na stronie Zmienne środowiskowe i PATH. Skrypt leżący w katalogu spoza tej listy uruchamiamy tylko przez pełną albo względną ścieżkę (`./skrypt.sh`, `/opt/skrypty/skrypt.sh`).

Dla skryptów, które mają być dostępne jak zwykłe komendy systemowe:

- `/usr/local/bin` — skrypty administracyjne dla wszystkich użytkowników danej maszyny, zwykle wymagają uprawnień roota do zapisu. To jest właściwe miejsce dla skryptu wdrażanego na serwer produkcyjny.
- `~/bin` (albo `~/.local/bin`) — skrypty prywatne bieżącego użytkownika, bez potrzeby uprawnień administracyjnych. Wiele dystrybucji dodaje ten katalog do `PATH` automatycznie, jeśli istnieje w chwili logowania.

```
mkdir -p ~/bin
cp pierwszy.sh ~/bin/
echo $PATH
```

Przykład:

```
/home/adm_user/.local/bin:/home/adm_user/bin:/usr/local/bin:/usr/bin:/bin
```

Jeśli `~/bin` nie widnieje na liście, dopisujemy go w pliku startowym powłoki i przeładowujemy sesję — to jest już temat plików startowych, nie tej strony.

Krok po kroku

1. Piszemy plik z poprawnym shebangiem.

```
cat > ~/bin/powitanie.sh <<'EOF'
#!/bin/bash
echo "System: $(uname -s), host: $(hostname)"
EOF
```

2. Nadajemy prawo wykonywania.

```
chmod +x ~/bin/powitanie.sh
```

3. Sprawdzamy, czy skrypt jest widoczny jak zwykła komenda.

```
which powitanie.sh
```

```
/home/adm_user/bin/powitanie.sh
```

`which` przeszukuje `PATH` dokładnie tak samo jak powłoka przed uruchomieniem komendy — jeśli katalog nie jest na liście, `which` nic nie znajdzie, mimo że plik fizycznie istnieje.

4. Uruchamiamy go samą nazwą.

```
powitanie.sh
```

```
System: Linux, host: app01
```

Częste błędy

Brak prawa wykonywania.

```
./skrypt.sh
```

```
bash: ./skrypt.sh: Permission denied
```

Kod wyjścia to 126 — plik istnieje, ale system nie może go uruchomić. Naprawa: `chmod +x skrypt.sh`.

Zakończenia linii w stylu Windows psują shebang.

```
./skrypt.sh
```

```
bash: ./skrypt.sh: /bin/bash^M: bad interpreter: No such file or directory
```

Plik przeklejony z edytora na Windowsie ma na końcu każdej linii dodatkowy znak `\r`. Jądro czyta shebang aż do znaku nowej linii, więc szuka programu o nazwie `/bin/bash\r` — takiego nie ma. Naprawa: `sed -i 's/\r$//' skrypt.sh` albo `dos2unix skrypt.sh`, jeśli pakiet jest zainstalowany.

Uruchomienie samą nazwą bez `./` i bez katalogu w PATH.

```
skrypt.sh
```

```
bash: skrypt.sh: command not found
```

Kod 127. Plik leży w katalogu bieżącym, ale powłoka nie szuka komend w katalogu bieżącym ze względów bezpieczeństwa — trzeba napisać `./skrypt.sh` albo dopisać katalog do `PATH`.

Pomylenie `source` z uruchomieniem w nowym procesie.

```
source deploy.sh
```

Skrypt kończy się linią `exit 1` po nieudanej walidacji. Ponieważ `source` wykonuje go w bieżącej powłoce, `exit 1` zamyka naszą sesję terminala (albo, na serwerze przez SSH, całe połączenie). Skrypty przeznaczone do `source`'owania — biblioteki funkcji, pliki z ustawieniami — kończymy przez `return`, nigdy przez `exit`.

□□ Ćwiczenia

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

Ćwiczenie 1 — trzy sposoby uruchomienia

```
cd /tmp
cat > test.sh <<'EOF'
#!/bin/bash
zmienna="ustawiona w skrypcie"
echo "działam jako proces $$"
EOF
chmod +x test.sh
echo "PID powłoki przed: $$"
./test.sh
echo "zmienna po ./: [$zmienna]"
source test.sh
echo "zmienna po source: [$zmienna]"
```

Dlaczego `zmienna` jest pusta po `./test.sh`, a ma wartość po `source test.sh`?

Odpowiedź

`./test.sh` uruchamia skrypt w osobnym procesie potomnym z własną kopią środowiska — `$$` w środku pokazuje inny PID niż w powłoce macierzystej, a wszystko, co skrypt ustawił, znika razem z tym procesem. `source test.sh` wykonuje te same komendy w bieżącym procesie powłoki, więc `zmienna` zostaje.

Ćwiczenie 2 — naprawa zepsutego shebangu

```
cd /tmp
printf '#!/bin/bash\r\necho "test"\r\n' > crlf.sh
chmod +x crlf.sh
./crlf.sh
```

Zobaczymy komunikat `bad interpreter`. Zdiagnozujmy przyczynę przez `cat -A crlf.sh`, naprawmy przez `sed -i 's/\r$//' crlf.sh` i uruchommy ponownie.

Odpowiedź

```
#!/bin/bash^M$
echo "test"^M$
```

`^M` na końcu każdej linii to znak powrotu karetki. Po `sed -i 's/\r$//' crlf.sh` plik uruchamia się poprawnie i wypisuje `test`.

Pytania kontrolne

1. Mamy plik `raport.sh` bez shebangu na początku. Co się stanie przy `./raport.sh`, a co przy `bash raport.sh`?

Odpowiedź

`./raport.sh` bez rozpoznawalnego shebangu w wielu przypadkach zostanie uruchomiony przez powłokę wywołującą jako skrypt tej powłoki (zachowanie zależne od systemu, bywa też błędem `Exec format error`) — to niepewne zachowanie, dlatego shebang zawsze piszemy. `bash raport.sh` zadziała niezależnie od zawartości pierwszej linii, bo interpreter jest już wskazany wprost w komendzie.

2. Dlaczego `bash -n skrypt.sh` nie wykryje, że skrypt usuwa niewłaściwy katalog?

Odpowiedź

`bash -n` sprawdza wyłącznie składnię — czy plik da się poprawnie sparsować. Nie uruchamia ani jednej komendy, więc nie ma pojęcia o logice biznesowej ani o tym, jakie wartości przyjmą zmienne w praktyce. Błąd logiczny w poprawnym składniowo skrypcie wykryjemy dopiero testem albo `bash -x`.

Powiązane: Kod wyjścia, $?, &&, ||, średnik · Zmienne środowiskowe i PATH · Cudzysłowy, escapowanie i ekspansja · Zmienne, parametry i argumenty · Tryb ścisły: set -euo pipefail

Źródła: `man 1 bash` (sekcja *Shell Scripts*) · `man 2 execve` · `man 1 env` · POSIX.1-2024

Revision #12
Created 2026-09-25 00:40:43 UTC by Tomasz Konieczny
Updated 2026-10-03 19:10:13 UTC by Tomasz Konieczny