

Materiały do przedmiotu:

Programowanie skryptowe

Zajęcia laboratoryjne nr 1

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	4
Ćwiczenia samodzielne	13
Pytania pomocnicze	17
Podsumowanie	17
Literatura	18



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Niniejszy przedmiot ma charakter laboratoryjny i składa się z dziewięciu tematów, które w sposób spójny przedstawiają zasady pracy z językami skryptowymi oraz praktyczne aspekty automatyzacji zadań w środowisku systemowym. Podstawowym elementem weryfikacji efektów uczenia się jest kolokwium realizowane po wykonaniu wszystkich instrukcji laboratoryjnych, którego celem jest sprawdzenie stopnia utrwalenia umiejętności rozwijanych w trakcie semestru.

Pierwsze laboratorium stanowi wprowadzenie do pracy ze skryptami powłoki w środowisku Unix/Linux. Jego celem jest uporządkowanie podstawowych pojęć związanych z językami skryptowymi oraz przygotowanie środowiska do dalszych ćwiczeń, w których skrypty będą wykorzystywane do automatyzacji zadań i przetwarzania danych. W trakcie zajęć uczestnik przechodzi przez kluczowe elementy pracy w terminalu: identyfikację używanej powłoki i konfiguracji systemu, tworzenie i uruchamianie pierwszego skryptu, interpretację kodów wyjścia, a także stosowanie przekierowań i potoków. Zdobyte umiejętności stanowią fundament do kolejnego laboratorium, w którym omawiane są podstawowe konstrukcje języka Bash i budowa bardziej rozbudowanych skryptów.

Cel ćwiczenia/laboratorium

- Zrozumienie roli języków skryptowych oraz skryptów powłoki w automatyzacji zadań systemowych i orkiestracji narzędzi.
- Zapoznanie się z pojęciem powłoki systemowej i jej funkcją jako środowiska uruchomieniowego (procesy, strumienie, kody wyjścia, przekierowania).
- Weryfikacja podstawowych parametrów środowiska pracy w systemie Unix/Linux (powłoka, wersja Basha, katalog roboczy, użytkownik, konfiguracja PATH).
- Nabycie umiejętności tworzenia pliku skryptu i przygotowania go do uruchomienia (shebang, prawa wykonywania, uruchamianie przez interpreter oraz jako plik wykonywalny).
- Przećwiczenie interpretacji kodów wyjścia oraz podstawowych technik przekierowań STDOUT/STDERR do plików.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- Utrwalenie praktycznego wykorzystania potoków do łączenia narzędzi tekstowych (printf/cat → grep → wc) w prostych scenariuszach analizy danych.

Instrukcja laboratoryjna/ćwiczeniowa

Języki skryptowe i skrypty powłoki

Język skryptowy jest językiem programowania, w którym kod jest zwykle wykonywany przez interpreter (lub maszynę uruchomieniową), bez konieczności wcześniejszej kompilacji do natywnego pliku wykonywalnego. Taki model sprzyja szybkiemu iterowaniu nad rozwiązaniem i jest szczególnie użyteczny w automatyzacji zadań, integracji narzędzi, przetwarzaniu plików oraz orkiestracji działań w środowisku systemowym. Skrypty są często wykorzystywane do realizacji logiki sterującej (pipeline), w której poszczególne etapy są delegowane do wyspecjalizowanych programów i narzędzi.

Skrypty powłoki (shell scripts) stanowią szczególną klasę skryptów, w których językiem wykonawczym jest powłoka systemowa, tj. program obsługujący interakcję użytkownika z systemem operacyjnym poprzez terminal. Powłoka pełni jednocześnie rolę: (1) interfejsu do uruchamiania programów i operacji na systemie plików, (2) języka oferującego konstrukcje sterujące (zmienne, instrukcje warunkowe, pętle, funkcje), oraz (3) środowiska uruchomieniowego zarządzającego m.in. zmiennymi środowiskowymi, kodami zakończenia procesów i mechanizmami przekierowań strumieni. W praktyce najczęściej spotyka się powłoki takie jak bash, zsh i fish w systemach Linux/macOS oraz PowerShell i cmd w systemie Windows.

Istotną cechą skryptów powłoki jest to, że zwykle nie implementują one pełnej funkcjonalności od podstaw, lecz organizują pracę poprzez uruchamianie zewnętrznych narzędzi i łączenie ich wyników. W konsekwencji kluczowe znaczenie mają pojęcia strumieni wejścia/wyjścia, przekierowań, potoków oraz kodów zakończenia. Powłoka uruchamia polecenia jako procesy, które standardowo operują



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



na strumieniach: STDIN (wejście), STDOUT (wyjście) oraz STDERR (wyjście diagnostyczne). Dodatkowo każdy proces zwraca kod wyjścia (exit code), gdzie konwencjonalnie wartość 0 oznacza poprawne zakończenie, a wartości niezerowe sygnalizują błąd lub stan szczególny. Mechanizmy te umożliwiają budowę potoków przetwarzania danych, w których wyjście jednego procesu staje się wejściem kolejnego, np.:

```
cat plik.txt | grep "ERROR" | wc -l
```

Warto pamiętać, że bez dodatkowych ustawień kod wyjścia potoku pochodzi z ostatniego polecenia w potoku.

Uwaga (Bash): po wykonaniu potoku Bash udostępnia tablicę PIPESTATUS, która zawiera kody wyjścia poszczególnych poleceń w potoku (np. \${PIPESTATUS[0]} dla pierwszego). Dodatkowo opcja set -o pipefail powoduje, że cały potok zwraca kod błędu, jeśli którekolwiek polecenie w potoku zakończy się niepowodzeniem (przydatne w skryptach do wykrywania błędów wcześniej niż na ostatnim etapie). W skrypcie można dodatkowo włączyć set -o pipefail, aby potok zwracał błąd, jeśli którykolwiek etap zakończy się niepowodzeniem.

Przykład: `printf "OK\nERROR\n" | grep "ERROR" | wc -l`

```
echo "${PIPESTATUS[@]}"
```

```
# set -o pipefail
```

Różnice między skryptem a pracą w powłocie

Podczas tworzenia skryptów powłoki należy uwzględnić sposób ich uruchamiania przez interpreter oraz wymagania systemu operacyjnego. W praktyce plik skryptu powinien rozpoczynać się tzw. shebangiem (np. `#!/bin/bash`), który jednoznacznie wskazuje interpreter odpowiedzialny za wykonanie kodu.

`#!/bin/bash` wskazuje interpreter po stałej ścieżce i działa w typowych dystrybucjach Linux, ale w niektórych środowiskach (np. niestandardowe instalacje, systemy z inną lokalizacją basha) spotyka się zapis `#!/usr/bin/env bash`, który wybiera bash



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



znaleziony przez PATH. W ramach laboratoriów przyjmujemy `#!/bin/bash` jako domyślny wariant. Jeżeli skrypt ma być uruchamiany bezpośrednio jako plik wykonywalny, konieczne jest nadanie mu odpowiednich uprawnień, np. poleceniem `chmod +x nazwa_skryptu.sh`. Skrypt można uruchomić na kilka równoważnych sposobów: poprzez jawne wywołanie interpretera (np. `bash nazwa_skryptu.sh`) albo jako plik wykonywalny z bieżącego katalogu (np. `./nazwa_skryptu.sh`), co jest możliwe po ustawieniu uprawnień i poprawnym shebangu.

W odróżnieniu od interaktywnego wydawania poleceń w terminalu, skrypt jest utrwalonym scenariuszem działań zapisanym w pliku, który może być wielokrotnie uruchamiany, parametryzowany argumentami, integrowany z harmonogramami zadań (np. cron lub Harmonogram zadań Windows) oraz z pipeline'ami CI/CD. Przykładowy minimalny skrypt w bash może przyjmować postać:

```
#!/bin/bash
```

```
echo "Start"
```

```
ls -la
```

```
echo "Koniec"
```

W powyższym przykładzie linia `#!/bin/bash` określa interpreter używany do wykonania pliku.

Analogiczny, minimalny przykład w PowerShell może wyglądać następująco:

```
Write-Host "Start"
```

```
Get-ChildItem
```

```
Write-Host "Koniec"
```

W kontekście laboratoriów szczególną rolę odgrywa środowisko uruchomieniowe, ponieważ zachowanie skryptów zależy m.in. od systemu operacyjnego i powłoki, wersji narzędzi dostępnych w systemie, zmiennych środowiskowych (np. PATH, HOME, TMP), uprawnień dostępu, a także zgodności kodowania i końców linii



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



(LF/CRLF). Z tego względu w ramach L1 istotne jest, aby potrafić zidentyfikować używaną powłokę i jej wersję, rozumieć rolę zmiennej PATH oraz uruchamiać skrypty zarówno poprzez bezpośrednie wywołanie interpretera, jak i jako plik wykonywalny.

Przygotowanie środowiska do pracy

W ramach zajęć zalecane jest środowisko zgodne z systemami Unix, w szczególności dystrybucje Linux oparte na Debianie (np. Kubuntu, Ubuntu, Zorin etc.), ze względu na dostępność standardowych narzędzi powłoki oraz spójność zachowania poleceń. Praca odbywa się w terminalu z użyciem powłoki bash (lub kompatybilnej), edytora tekstu oraz standardowych narzędzi systemowych. Należy zwrócić uwagę na poprawne kodowanie plików i końce linii (LF), ponieważ skrypty przygotowane w środowisku Windows mogą wymagać konwersji (np. ze względu na CRLF). W trakcie ćwiczeń istotne jest również rozumienie zmiennej środowiskowej PATH, katalogu roboczego oraz uprawnień w systemie plików, gdyż elementy te bezpośrednio wpływają na możliwość uruchamiania poleceń i skryptów.

Ćwiczenie praktyczne

Celem poniższego ćwiczenia jest potwierdzenie poprawnej konfiguracji środowiska terminalowego, przygotowanie katalogu roboczego, utworzenie pierwszego skryptu powłoki oraz przećwiczenie podstaw uruchamiania, kodów wyjścia, przekierowań i potoków. Mechanizmy grep oraz potokowania są wykorzystywane zgodnie z wiedzą z podstaw systemów operacyjnych i stanowią naturalny element pracy w powłoce.

Krok 1: Weryfikacja środowiska uruchomieniowego

Na początku należy ustalić, w jakiej powłoce pracuje użytkownik, jaka jest wersja Basha, jaki jest bieżący katalog roboczy oraz jaka jest konfiguracja zmiennej PATH (czyli lista katalogów, w których system wyszukuje programy uruchamiane bez podawania pełnej ścieżki).

Wykonaj kolejno (Rysunek 1):



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



`echo "$SHELL"`

`echo "$0"`

`ps -p $$ -o comm=`

`bash --version | head`

`command -v bash`

`pwd`

`whoami`

`echo "$PATH"`

```
student@student-VirtualBox:~/Desktop/Skryptowe$ echo "$SHELL"
/bin/bash
student@student-VirtualBox:~/Desktop/Skryptowe$ echo "$0"
bash
student@student-VirtualBox:~/Desktop/Skryptowe$ ps -p $$ -o comm=
bash
student@student-VirtualBox:~/Desktop/Skryptowe$ bash --version | head
GNU bash, version 5.0.17(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
student@student-VirtualBox:~/Desktop/Skryptowe$ command -v bash
/usr/bin/bash
student@student-VirtualBox:~/Desktop/Skryptowe$ pwd
/home/student/Desktop/Skryptowe
student@student-VirtualBox:~/Desktop/Skryptowe$ whoami
student
student@student-VirtualBox:~/Desktop/Skryptowe$ echo "$PATH"
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
```

Rys. 1. Polecenia diagnostyczne środowiska uruchomieniowego

Interpretacja wyników:

1. `echo "$SHELL"` - zwykle wskazuje domyślną powłokę przypisaną do konta użytkownika.
2. `echo "$0"` - wypisuje nazwę bieżącej powłoki/uruchomionego programu (w terminalu interaktywnym często pokazuje, w czym aktualnie pracujesz, np. `bash` lub `zsh`).



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



3. `ps -p $$ -o comm=` - sprawdza nazwę procesu o PID bieżącej powłoki (`$$`), czyli "systemowo" potwierdza, jaki program jest aktualnie uruchomioną powłoką.
4. `bash --version` - potwierdza dostępność Basha i jego wersję (często w kilku liniach).
5. `command -v bash` - pokazuje, z jakiej ścieżki uruchamiany jest bash w tym systemie.
6. `pwd` - pokazuje bieżący katalog roboczy, istotny przy pracy na plikach.
7. `whoami` - informuje, na jakim koncie wykonywane są polecenia (ważne w kontekście uprawnień).
8. `echo "$PATH"` - pozwala zrozumieć, skąd system „widzi” polecenia; błędny PATH jest częstą przyczyną problemów z uruchamianiem narzędzi.

Uwaga: `echo "$SHELL"` zwykle pokazuje domyślną powłokę przypisaną do konta (login shell), a nie zawsze aktualnie uruchomiony proces powłoki. Aby sprawdzić bieżącą powłokę/proces, można użyć `echo "$0"` (w interaktywnym terminalu często wystarczy) oraz `ps -p $$ -o comm=` (podejście systemowe, które sprawdza nazwę procesu o PID bieżącej powłoki).

Krok 2: Utworzenie katalogu roboczego i przygotowanie pliku skryptu

Dla przejrzystości zaleca się pracę w dedykowanym katalogu laboratoriów. Utwórz katalog `lab1`, przejdź do niego, a następnie utwórz plik skryptu:

```
mkdir -p lab1
```

```
cd lab1
```

```
nano hello.sh
```

Jeżeli zamiast nano używany jest inny edytor, dopuszczalne są m.in. vim lub edytor graficzny (np. Kate). Kluczowe jest, aby plik był zwykłym plikiem tekstowym.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

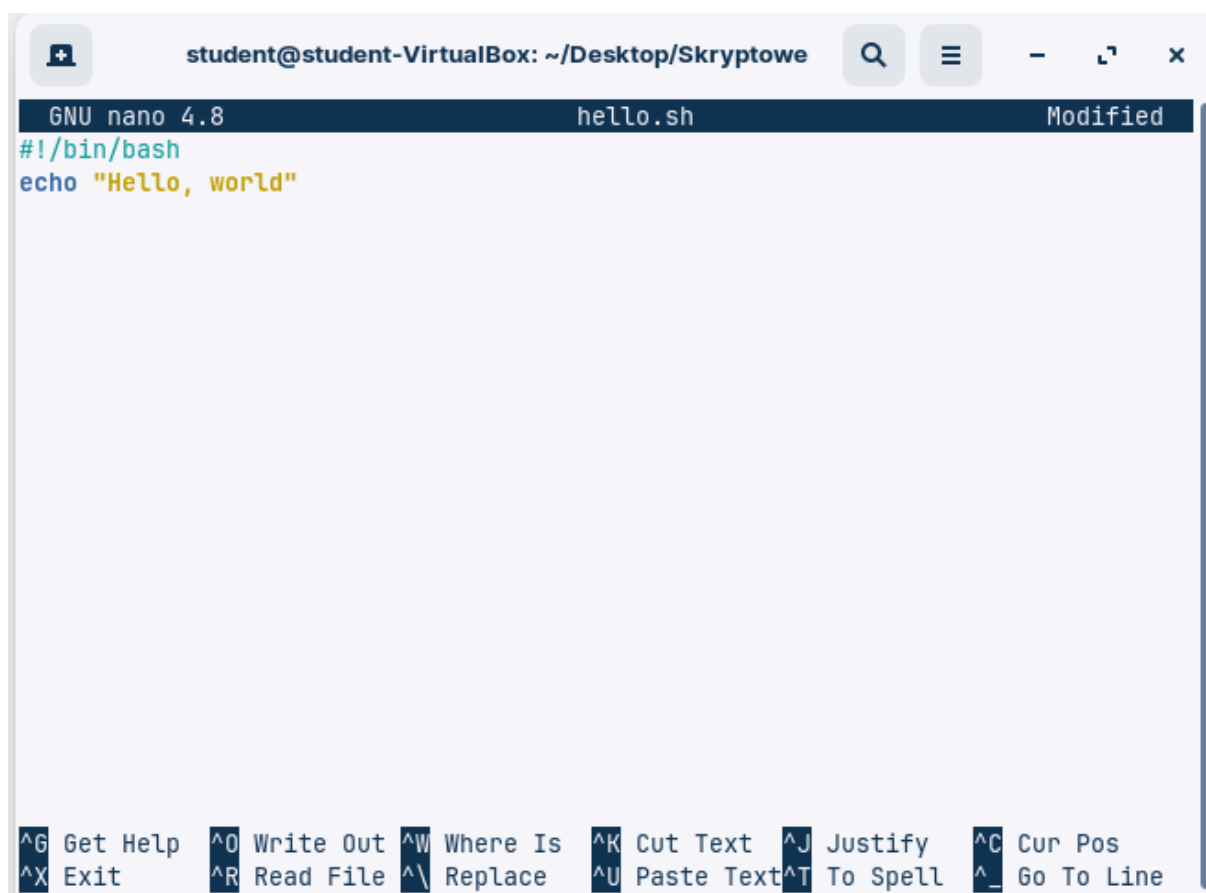


Krok 3: Implementacja „Hello world” w Bash

W pliku hello.sh umieść poniższą zawartość, rysunek 2:

```
#!/bin/bash  
  
echo "Hello, world"
```

Zapisz plik i zamknij edytor.



The screenshot shows a terminal window titled "student@student-VirtualBox: ~/Desktop/Skryptowe". Inside, the GNU nano 4.8 editor is open, editing a file named "hello.sh". The editor's status bar at the top indicates "Modified". The content of the file is:
#!/bin/bash
echo "Hello, world"
The bottom of the window displays a series of keyboard shortcuts for nano, such as ^G Get Help, ^O Write Out, ^W Where Is, ^K Cut Text, ^J Justify, ^C Cur Pos, ^X Exit, ^R Read File, ^_ Replace, ^U Paste Text, ^T To Spell, and ^_ Go To Line.

Rys. 2. Pierwszy skrypt w powłoce bash

Krok 4: Uruchomienie skryptu na dwa sposoby

4.1 Uruchomienie przez jawne wywołanie interpretera



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

To podejście nie wymaga nadawania praw wykonywania plikowi:

```
bash hello.sh
```

4.2 Uruchomienie jako plik wykonywalny

Aby uruchomić skrypt bezpośrednio, należy nadać mu atrybut wykonywalności, a następnie wywołać go z bieżącego katalogu (rysunek 3):

```
chmod +x hello.sh
```

```
./hello.sh
```

Uwagi:

- `chmod +x` ustawia prawo wykonywania dla pliku w systemie uprawnień Unix.
- Prefiks `./` jest istotny: oznacza uruchomienie pliku z bieżącego katalogu. Bieżący katalog zwykle nie jest częścią PATH, co ogranicza ryzyko przypadkowego uruchomienia nieautoryzowanych plików.

```
student@student-VirtualBox:~/Desktop/Skryptowe$ ls -l
total 4
-rw-rw-r-- 1 student student 32 lut 22 22:46 hello.sh
student@student-VirtualBox:~/Desktop/Skryptowe$ chmod +x hello.sh
student@student-VirtualBox:~/Desktop/Skryptowe$ ls -l
total 4
-rwxrwxr-x 1 student student 32 lut 22 22:46 hello.sh
student@student-VirtualBox:~/Desktop/Skryptowe$ bash hello.sh
Hello, world
student@student-VirtualBox:~/Desktop/Skryptowe$ ./hello.sh
Hello, world
student@student-VirtualBox:~/Desktop/Skryptowe$
```

Rys. 3. Nadanie uprawnień i uruchomienie skryptu

Krok 5: Kod wyjścia procesu i podstawowe przekierowanie



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Każde polecenie w powłoce kończy się kodem wyjścia. Wartość 0 oznacza typowo poprawne zakończenie. Po uruchomieniu skryptu można odczytać kod wyjścia z użyciem zmiennej \$?.

Wykonaj poniższe polecenia (rysunek 4):

```
./hello.sh
```

```
echo $?
```

Następnie przećwicz przekierowanie standardowego wyjścia do pliku (rysunek 4):

```
./hello.sh > out.txt
```

```
cat out.txt
```

```
student@student-VirtualBox:~/Desktop/Skryptowe$ ./hello.sh; echo $?  
Hello, world  
0  
student@student-VirtualBox:~/Desktop/Skryptowe$ ./hello.sh > out.txt  
student@student-VirtualBox:~/Desktop/Skryptowe$ cat out.txt  
Hello, world
```

Rys. 4. Kod wyjścia i przekierowanie wyniku skryptu do pliku

Poniżej przykład różnicy między nadpisaniem (>) a dopisaniem (>>):

```
echo "pierwsza linia" > test.txt
```

```
echo "druga linia" >> test.txt
```

```
cat test.txt
```

Uwagi:

- Operator > przekierowuje STDOUT do pliku (nadpisując go).
- Operator >> dopisuje dane na koniec pliku (bez nadpisywania), co jest przydatne np. przy budowie raportów z wielu uruchomień.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- cat out.txt pozwala potwierdzić zawartość pliku i zweryfikować poprawność przekierowania.

Krok 6: Minimalny przykład potoku z użyciem grep i wc

Potokowanie jest standardową techniką łączenia narzędzi tekstowych. Poniższy przykład generuje trzy linie tekstu, filtruje je przez grep, a następnie zlicza liczbę dopasowań poleceniem wc -l.

Wykonaj:

- `printf "OK\nERROR\nOK\n" | grep "ERROR" | wc -l`

Interpretacja:

- printf ... generuje strumień danych na STDOUT.
- grep "ERROR" przepuszcza wyłącznie linie zawierające wzorzec.
- wc -l zlicza liczbę linii, czyli liczbę dopasowań.

Wynik powinien wynosić 1, ponieważ w danych wejściowych występuje jedna linia zawierająca ERROR.

Typowe błędy podczas uruchamiania skryptów i sposoby ich rozwiązania

Na początku najczęściej psuje się nie sam skrypt, tylko jego uruchomienie. Typowe problemy to brak prawa wykonywania (po ./skrypt.sh pojawia się „Permission denied” i trzeba nadać uprawnienia poleceniem `chmod +x`), praca w złym katalogu lub literówka w nazwie pliku (stąd „No such file or directory”), a także próba uruchomienia skryptu bez prefiksu ./, ponieważ bieżący katalog zwykle nie należy do PATH. Częstą przeszkodą bywa też zapis pliku z końcami linii Windows (CRLF), który objawia się komunikatem o „bad interpreter ... ^M” i wymaga konwersji pliku do formatu Linux.

Można to zrobić poleceniami:

`dos2unix hello.sh`

albo



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



`sed -i 's/r$//' hello.sh` Wreszcie wielu początkujących myli przekierowania: operator `>` zapisuje tylko STDOUT, więc komunikaty błędów pozostają na ekranie, dopóki nie zostaną przekierowane osobno przez `2> stderr.txt` lub połączone ze STDOUT przez `2>&1`.

`2>&1` wykonuje się w momencie, gdy STDOUT jest jeszcze skierowany na terminal, więc STDERR zostaje „przypięty” do terminala, a dopiero potem STDOUT trafia do pliku.

Ćwiczenia samodzielne

Dopuszcza się wstępną realizację zadań w terminalu, natomiast wymaganym rezultatem jest skrypt Bash automatyzujący rozwiązanie. Ćwiczenia dotyczące przekierowań i potoków można wykonać interaktywnie w terminalu; w razie wymagań prowadzącego należy zebrać polecenia do jednego skryptu i uruchomić go jednokrotnie.

1. W katalogu lab1 należy utworzyć skrypt (np. `env_report.sh`), który wygeneruje plik `env_report.txt`. W raporcie mają znaleźć się, w czytelnej formie i w stałej kolejności, wyniki poleceń: `echo "$SHELL"`, `bash --version`, `command -v bash`, `pwd`, `whoami` oraz `echo "$PATH"`. Na końcu pliku należy dopisać bieżącą datę i godzinę uzyskaną poleceniem `date`.
2. Należy utworzyć skrypt `hello2.sh`. Skrypt ma wypisać nazwę bieżącego użytkownika, bieżący katalog roboczy oraz informację „Skrypt uruchomiony poprawnie”. Następnie należy uruchomić go na dwa sposoby: (1) poprzez jawne wywołanie interpretera (`bash hello2.sh`) oraz (2) jako plik wykonywalny (`./hello2.sh`) po nadaniu uprawnień poleceniem `chmod +x hello2.sh`.
3. Należy uruchomić `hello2.sh` w taki sposób, aby jego standardowe wyjście zostało zapisane do pliku `run_output.txt` przy użyciu przekierowania `>`. Następnie należy wyświetlić zawartość pliku poleceniem `cat run_output.txt`.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



oraz policzyć liczbę linii i liczbę znaków w pliku, używając wc -l oraz wc -m (lub wc -c).

4. Wykonaj:

```
{ echo "To jest STDOUT"; echo "To jest STDERR" >&2; ls  
/katalog_ktory_nie_istnieje; } > stdout.txt 2> stderr.txt
```

Sprawdź zawartość plików:

```
cat stdout.txt
```

```
cat stderr.txt
```

Zaobserwuj, że stdout.txt zawiera tylko dane wypisane na STDOUT, a komunikaty błędów (oraz tekst wysłany przez >&2) trafiają do stderr.txt.

5. Utwórz plik i dopisz kolejne linie:

```
echo "pierwsza linia" > append_test.txt
```

```
echo "druga linia" >> append_test.txt
```

```
echo "trzecia linia" >> append_test.txt
```

```
cat append_test.txt
```

Następnie nadpisz plik:

```
echo "NOWA ZAWARTOŚĆ" > append_test.txt
```

```
cat append_test.txt
```

Zaobserwuj, że operator > nadpisuje plik (usuwa poprzednią zawartość), natomiast >> dopisuje kolejne linie na końcu bez kasowania wcześniejszych danych.

6. Wykonaj:

```
{ echo "OK"; ls /katalog_ktory_nie_istnieje; } > all.txt 2>&1
```

```
cat all.txt
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zaobserwuj, że w all.txt znajdują się zarówno komunikaty STDOUT, jak i STDERR, ponieważ 2>&1 kieruje STDERR tam, gdzie aktualnie skierowany jest STDOUT.

7. Wykonaj:

```
{ echo "OK"; ls /katalog_ktory_nie_istnieje; } 2>&1 > all_wrong_order.txt
```

Mimo że STDOUT trafia do all_wrong_order.txt, komunikat błędu może pojawić się na ekranie, a kolejność przekierowań ma znaczenie.

Wyjaśnij krótko w 1–2 zdaniach, dlaczego w tym wariancie komunikat błędu może pojawić się na ekranie zamiast w pliku.

8. Należy wykonać jedno polecenie, które zakończy się sukcesem (np. ls), a następnie jedno polecenie błędne (np. ls /katalog_ktory_nie_istnieje). Po każdym z nich należy wypisać kod wyjścia poleceniem echo \$? i krótko zinterpretować, co oznacza kod 0 oraz co oznacza kod niezerowy.

9. Utwórz plik sample.log poleceniem:

```
cat > sample.log <<'EOF'
```

```
2026-02-22 09:00:00 INFO Start aplikacji
```

```
2026-02-22 09:00:01 INFO Wczytano konfigurację
```

```
2026-02-22 09:00:02 WARN Brak wpisu CACHE_SIZE, użyto domyślnej wartości
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



2026-02-22 09:00:03 INFO Połączono z bazą danych

2026-02-22 09:00:04 ERROR Nie udało się otworzyć pliku
/var/data/input.csv

2026-02-22 09:00:05 INFO Próba ponowienia operacji

2026-02-22 09:00:06 WARN Wolna odpowiedź serwera (timeout=2s)

2026-02-22 09:00:07 INFO Zapisano wynik pośredni do /tmp/out.tmp

2026-02-22 09:00:08 ERROR Błąd walidacji rekordu id=17

2026-02-22 09:00:09 INFO Kontynuacja przetwarzania

2026-02-22 09:00:10 WARN Niski poziom miejsca na dysku

2026-02-22 09:00:11 INFO Zakończono zadanie

2026-02-22 09:00:12 ERROR Nieoczekiwany kod odpowiedzi HTTP 503

EOF

Następnie należy przygotować i wykonać trzy polecenia z użyciem potoków, które: (1) zliczą liczbę linii zawierających ERROR, (2) zliczą liczbę linii zawierających WARN, oraz (3) zapiszą do pliku errors_only.txt wyłącznie linie zawierające ERROR. W każdym przypadku wymagane jest użycie grep oraz wc -l w postaci potoku tam, gdzie wykonywane jest zliczanie.

10. Należy sprawdzić, gdzie w systemie znajduje się polecenie bash oraz grep, a następnie zapisać wyniki do pliku tools_location.txt. Do tego samego pliku należy dopisać aktualną wartość zmiennej PATH. Wymagane jest użycie command -v (zamiast which) jako metody sprawdzania lokalizacji poleceń.
11. Należy utworzyć skrypt report.sh, którego zadaniem jest wypisanie na ekran bieżącej daty i godziny, nazwy użytkownika, bieżącego katalogu roboczego oraz listy plików w katalogu lab1 (np. poprzez ls -la). Następnie należy nadać



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



skryptowi uprawnienia do wykonywania i uruchomić go jako `./report.sh`, przekierowując całe standardowe wyjście do pliku `report.txt`.

Pytania pomocnicze

1. Czym różni się język skryptowy od języka kompilowalnego w kontekście modelu uruchamiania programu?
2. Jaką rolę pełni `#!/bin/bash`?
3. Jakie warunki muszą być spełnione, aby skrypt można było uruchomić poleceniem `./skrypt.sh`?
4. Na czym polega różnica między uruchomieniem `bash skrypt.sh` a `./skrypt.sh`?
5. Co oznacza kod wyjścia 0, a co oznacza kod niezerowy? Jak odczytać kod wyjścia ostatniego polecenia?
6. Jakie są role strumieni `STDIN`, `STDOUT` i `STDERR`?
7. W jaki sposób rozdzielić `STDOUT` i `STDERR` do dwóch różnych plików i po co się to robi?
8. Co to jest potok i jaki jest sens łączenia narzędzi typu `grep` i `wc -l`?
9. Do czego służy zmienna `PATH` i jak może wpływać na uruchamianie poleceń?
10. Dlaczego końce linii (`LF/CRLF`) mogą powodować problemy przy uruchamianiu skryptów w Linuxie?

Podsumowanie

W trakcie pierwszego laboratorium omówiono rolę języków skryptowych i skryptów powłoki jako narzędzi do automatyzacji zadań oraz integracji programów w środowisku systemowym. Utrwalono, że skrypt powłoki jest wykonywany przez interpreter, a jego poprawne uruchamianie zależy zarówno od zawartości pliku (w szczególności od shebangu wskazującego interpreter), jak i od ustawień systemu plików (nadanie prawa wykonywania) oraz bieżącego kontekstu uruchomienia (katalog roboczy, `PATH`). Przećwiczono podstawowe mechanizmy pracy w powłoce, obejmujące interpretację kodów wyjścia, przekierowania strumieni standardowych do plików oraz budowę potoków łączących narzędzia tekstowe. Zwrócono również uwagę na znaczenie środowiska uruchomieniowego, w tym wersji powłoki,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



dostępności narzędzi oraz zgodności kodowania i końców linii, które w praktyce często determinują poprawność działania skryptów.

W praktyce środowiska pracy są często „oskryptowane” poprzez utrzymywanie własnej skrzynki narzędziowej: repozytorium konfiguracji oraz zestawu powtarzalnych skryptów i komend standaryzujących diagnostykę, raportowanie i automatyzację. Ponieważ w ramach jednego przedmiotu nie da się zbudować kompletnego zestawu takich narzędzi, zajęcia mogą równolegle rozwijać dwa typy rozwiązań: proste skrypty dydaktyczne uczące mechanizmów powłoki i dobrych praktyk oraz praktyczne skrypty użytkowe uporządkowane w spójnej strukturze repozytorium. Efekty pracy warto traktować jako załączek prywatnej skrzynki narzędziowej, która w realnym zastosowaniu wymaga dalszego, ciągłego rozwoju i aktualizacji.

Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:
Programowanie skryptowe
Zajęcia laboratoryjne nr 2

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	3
Ćwiczenia samodzielne	12
Pytania pomocnicze	14
Podsumowanie	15
Literatura	16



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Laboratorium drugie ma na celu opanowanie elementarnych konstrukcji języka Bash wykorzystywanych do sterowania przebiegiem skryptu. W ramach zajęć realizowane jest definiowanie i poprawne wykorzystywanie zmiennych, świadome stosowanie cytowania (quoting) oraz bezpieczna praca z argumentami wiersza poleceń. Utrwalane jest również użycie podstawowych instrukcji sterujących (if, case) oraz pętli (for, while, until) w prostych scenariuszach decyzyjnych i iteracyjnych.

Cel ćwiczenia/laboratorium

- Zrozumienie i poprawne stosowanie zmiennych w Bash (przypisanie, odczyt, \${...} w kontekstach niejednoznacznych).
- Świadome stosowanie cytowania i podstawowych zasad bezpieczeństwa argumentów ("...", '...', unikanie niezamierzonego word splitting i globbingu).
- Parametryzacja skryptu argumentami wiersza poleceń (\$0, \$1..\$n, \$#, "\$@") oraz prosta walidacja minimalnych wymagań wywołania (np. liczba argumentów).
- Implementacja sterowania przepływem: instrukcje warunkowe if/elif/else, konstrukcje testowe [] i [[]], oraz case dla wyboru wariantu działania.
- Implementacja iteracji: pętle for, while, until, w tym przerwanie/sterowanie przebiegiem pętli (break, continue).
- Budowa prostych skryptów realizujących logikę decyzyjną i iteracyjną

Instrukcja laboratoryjna/ćwiczeniowa

Zmienne, podstawienia i cytowanie

Zmienne w Bash służą do przechowywania i ponownego wykorzystywania wartości w trakcie działania skryptu. Umożliwiają parametryzację kodu, ograniczają duplikację oraz upraszczają modyfikację skryptu, ponieważ wartości konfiguracyjne (np. ścieżki, nazwy plików, progi, flagi uruchomieniowe) można utrzymywać w jednym miejscu. W praktyce zmienne są również nośnikami danych wejściowych użytkownika, argumentów, wyników poleceń oraz informacji o środowisku uruchomieniowym.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Warto podkreślić, że w Bash zmienne są domyślnie typu tekstowego, a konwersje (np. do liczb) mają charakter kontekstowy, zależny od użytej konstrukcji.

Zmienną tworzy się przez przypisanie w postaci NAZWA=wartość, bez spacji po obu stronach znaku =. Nazwy zmiennych powinny składać się z liter, cyfr i znaku podkreślenia, przy czym nie mogą zaczynać się od cyfry. Odczyt wartości zmiennej realizowany jest przez podstawienie parametru, tj. użycie prefiksu \$, np. \$NAZWA. W sytuacjach, w których zmienna sąsiaduje z innym tekstem (np. przy konkatenaacji), zaleca się formę \${NAZWA}, ponieważ jednoznacznie wyznacza zakres nazwy zmiennej i eliminuje błędy parsowania.

Ponieważ Bash jest powłoką uruchamiającą polecenia z listy argumentów, mechanizm cytowania ma bezpośredni wpływ na to, jakie argumenty faktycznie zostaną przekazane do programu. Bez cytowania powłoka może wykonać tzw. word splitting (dzielenie po białych znakach) oraz pathname expansion (globbing, np. rozwinięcie * do listy plików). W konsekwencji, jeżeli zmienna zawiera spacje lub znaki specjalne, brak cytowania może zmienić liczbę argumentów i prowadzić do błędów logicznych lub niezamierzonego przetwarzania plików. Cudzysłów podwójny ("...") zachowuje spacje i większość znaków, a jednocześnie pozwala na podstawienia zmiennych i poleceń. Apostrof ('...') traktuje zawartość dosłownie i blokuje podstawienia, co jest użyteczne m.in. w tekstach stałych, wzorcach lub fragmentach, które mają trafić do narzędzia bez interpretacji przez powłokę.

W praktyce poprawne cytowanie jest jednym z najważniejszych elementów niezawodności skryptów. Zasada inżynierska jest następująca: zmienne, które mogą zawierać spacje lub znaki specjalne (np. ścieżki, nazwy plików, dane wejściowe użytkownika), należy w większości przypadków ujmować w cudzysłów podwójny, np. "\$plik" lub "\${katalog}". Wyjątki wymagają świadomej decyzji (np. celowe wykorzystanie globbingu), a nie przypadkowego braku cudzysłowu.

Poniższe przykłady ilustrują typowe zastosowania oraz konsekwencje cytowania.

Rysunek 1 przedstawia różnice między poprawnym a błędnym cytowaniem.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Poprawne przypisanie (bez spacji)

```
name="Jan Kowalski"
```

```
file_path="/home/user/Moje pliki/dane.txt"
```

Odczyt wartości

```
echo "$name"
```

```
echo "${file_path}"
```

Konkatenacja - zalecane użycie nawiasów klamrowych

```
prefix="wynik"
```

```
echo "${prefix}_2026.txt"
```

UWAGA: Poniższy przykład bez cytowania przedstawia celowo błędny wzorec użycia i służy wyłącznie demonstracji skutków word splitting i globbingu.

Brak cytowania: potencjalny błąd (word splitting + globbing)

Jeżeli file_path zawiera spacje, program dostanie kilka argumentów zamiast jednego.

```
cat $file_path
```

Cytowanie: poprawne przekazanie jednego argumentu



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



cat "\$file_path"

```
albert@DESKTOP-3NTR2GI:~/skryptowe$ cd lab2
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ mkdir -p "/tmp/Moje pliki"
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ echo "Przykładowa zawartosc" > "/tmp/Moje pliki/dane.txt"
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ file_path="/tmp/Moje pliki/dane.txt"
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ echo "=== Bez cytowania ==="
=== Bez cytowania ===
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ cat $file_path
cat: /tmp/Moje: No such file or directory
cat: pliki/dane.txt: No such file or directory
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ echo "=== Z cytowaniem ==="
=== Z cytowaniem ===
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ cat "$file_path"
Przykładowa zawartosc
```

Rys. 1. Różnice między poprawnym a błędnym cytowaniem.

Różnica między cudzysłowem podwójnym a apostrofem

x="ABC"

echo "\$x" # wypisze: ABC

echo '\$x' # wypisze dosłownie: \$x

Podstawienie wyniku polecenia do zmiennej (command substitution)

Uwaga: wynik może zawierać spację, dlatego późniejsze użycie zwykle cytujemy.

today="\$(date +%F)"

echo "Data: \$today"

Argumenty, wejście/wyjście oraz podstawy obsługi błędów

Skrypty w Bash są projektowane tak, aby mogły być uruchamiane w różnych wariantach bez modyfikowania kodu, a ich zachowanie było sterowane parametrami przekazanymi przy uruchomieniu. Podstawowym mechanizmem parametryzacji są argumenty wiersza poleceń. Dzięki nim ten sam skrypt może pracować na różnych plikach, katalogach, trybach działania lub wartościach progowych. Argumenty są



Fundusze Europejskie
dla Rozwoju Społecznego

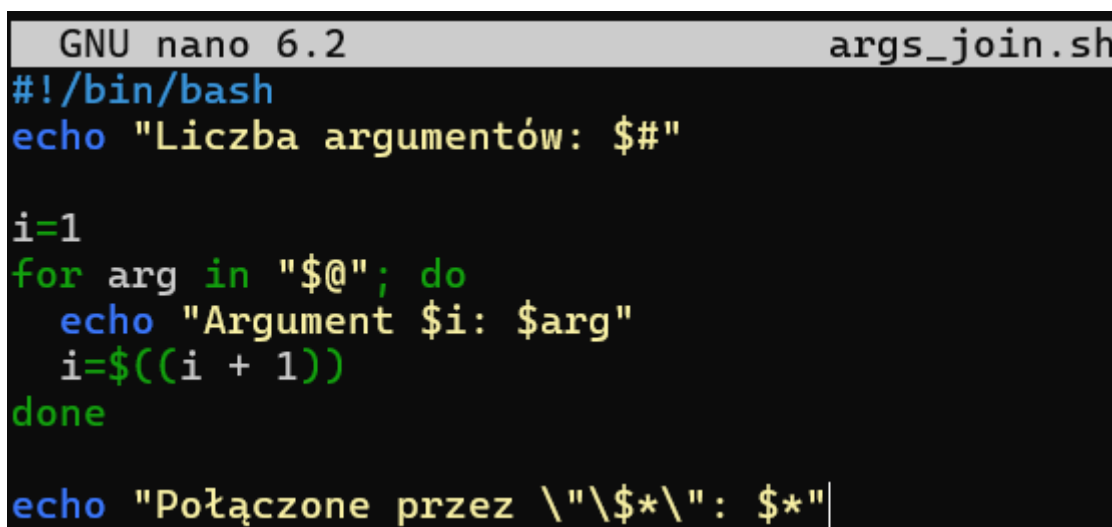


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



również podstawą integracji skryptu z innymi narzędziami, ponieważ umożliwiają automatyczne wywołania w harmonogramach zadań (cron), pipeline'ach CI/CD i systemach monitoringu. W Bash argumenty dostępne są poprzez tzw. parametry pozycyjne. Zmienna \$0 zawiera nazwę (lub ścieżkę) skryptu, natomiast \$1, \$2, ..., \$n odpowiadają kolejnym argumentom przekazanym przez użytkownika. Liczba argumentów jest dostępna w \$# , której użycie zostało zaprezentowane na rysunku 2 i 3. Bash udostępnia także zbiory argumentów jako całość: \$* oraz \$@. W praktyce kluczowa jest różnica w zachowaniu tych dwóch konstrukcji, szczególnie w obecności cudzysłówów. W wariancie "\$@" każdy argument pozostaje osobnym elementem listy (zachowuje granice), natomiast "\$*" łączy argumenty w jeden łańcuch. Z punktu widzenia poprawności i bezpieczeństwa przekazywania parametrów do innych poleceń, standardową praktyką jest stosowanie "\$@" , ponieważ eliminuje błędy wynikające z argumentów zawierających spacje lub znaki specjalne.



```
GNU nano 6.2                                args_join.sh
#!/bin/bash
echo "Liczba argumentów: $#"
```



```
i=1
for arg in "$@"; do
    echo "Argument $i: $arg"
    i=$((i + 1))
done
```



```
echo "Połączone przez \"\$*\": $*" |
```

Rys. 2. Wykorzystanie \$# w skrypcie.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ chmod +x args_join.sh
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ ./args_join.sh Ala "Ala ma kota" 123
Liczba argumentów: 3
Argument 1: Ala
Argument 2: Ala ma kota
Argument 3: 123
Połączone przez "$*": Ala Ala ma kota 123
```

Rys. 3. Skrypt prezentujący różnice pomiędzy \$#, #@ oraz #*.

W skryptach inżynierskich konieczne jest jednoznaczne określenie interfejsu uruchomieniowego, tj. jakie argumenty są wymagane, jakie są opcjonalne oraz w jakiej kolejności mają się pojawić. Minimalnym mechanizmem kontroli jest weryfikacja liczby argumentów przez \$# i wyświetlenie komunikatu użycia (usage) w przypadku błędnego wywołania. Przykład obsługi parametrów wejściowych na rysunku 4 i 5. Dobrą praktyką jest kierowanie komunikatu użycia na STDERR oraz zakończenie działania skryptu kodem niezerowym, co umożliwia automatyczne wykrywanie błędów przez narzędzia nadrzędne.

Przykładowa walidacja argumentów i komunikat użycia:

```
#!/bin/bash

if [ "$#" -lt 2 ]; then

    echo "Użycie: $0 <plik_wejściowy> <katalog_wyjściowy>" >&2

    exit 2

fi

input="$1"

out_dir="$2"
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
GNU nano 6.2                                     usage_demo.sh *
#!/bin/bash
if [ "$#" -lt 2 ]; then
    echo "Użycie: $0 <plik_wejściowy> <katalog_wyjściowy>" >&2
    exit 2
fi
echo "OK"
```

Rys. 4. Skrypt prezentujący wymagane parametry wejściowe.

```
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ ./usage_demo.sh
Użycie: ./usage_demo.sh <plik_wejściowy> <katalog_wyjściowy>
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ echo $?
2
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ ./usage_demo.sh input.txt out/
OK
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ echo $?
0
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$
```

Rys. 5. Wywołanie skryptu i odczytanie kodu wykonania.

Warto także stosować podstawową walidację danych wejściowych przekazanych argumentami. Typowym przykładem jest weryfikacja istnienia pliku oraz uprawnień do odczytu lub zapisu. W Bash można to realizować testami plikowymi (np. -f, -d, -r, -w), a w razie niespełnienia warunku generować komunikat na STDERR i kończyć działanie z kodem niezerowym.

Przykład walidacji pliku i katalogu:

```
if [ ! -f "$input" ]; then
    echo "Błąd: plik wejściowy nie istnieje: $input" >&2
    exit 1
fi

if [ ! -d "$out_dir" ]; then
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
echo "Błąd: katalog wyjściowy nie istnieje: $out_dir" >&2
```

```
exit 1
```

```
fi
```

Sterowanie przepływem: instrukcje warunkowe i wybór wariantu działania

W skryptach Bash sterowanie przepływem realizowane jest przez instrukcje warunkowe `if/elif/else` oraz konstrukcję `case`. Instrukcja `if` pozwala wykonać różne fragmenty kodu w zależności od spełnienia warunku logicznego, natomiast `case` umożliwia czytelny wybór wariantu działania na podstawie wartości zmiennej (np. trybu pracy lub operatora). Warunki w Bash są najczęściej zapisywane z użyciem testów `[ ... ]` lub `[[ ... ]]`. W praktyce `[[ ... ]]` jest wygodniejsze dla porównań tekstowych i dopasowań, natomiast `[ ... ]` zapewnia bardziej klasyczny, przenośny zapis. Dla porównań liczbowych stosuje się operatory takie jak `-lt`, `-le`, `-eq`, `-ne`, `-ge`, `-gt`, a dla porównań tekstowych `m.in. =` oraz `!=`.

Minimalny przykład warunku liczbowego:

```
if [ "$n" -gt 0 ]; then  
    echo "dodatnia"  
elif [ "$n" -eq 0 ]; then  
    echo "zero"  
else  
    echo "ujemna"  
fi
```

Minimalny przykład wyboru wariantu (`case`):



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
case "$mode" in
    dev) LOG_LEVEL="DEBUG" ;;
    test) LOG_LEVEL="INFO" ;;
    prod) LOG_LEVEL="WARN" ;;
    *) echo "Nieznany tryb: $mode" >&2; exit 2 ;;
esac
```

Iteracja: pętle for, while, until oraz break i continue

Pętle umożliwiają wielokrotne wykonanie fragmentu kodu dla kolejnych wartości lub do momentu spełnienia warunku zakończenia. Pętla for jest wykorzystywana przede wszystkim do iteracji po listach wartości lub po indeksach. W Bash powszechnie stosuje się również pętlę arytmetyczną for ((...)), która jest wygodna przy iteracji po liczbach całkowitych.

Przykład iteracji po zakresie liczbowym (pętla arytmetyczna):

```
for ((i=start; i<=end; i++)); do
    echo "$i"
done
```

W powyższym przykładzie start i end są zmiennymi liczbowymi (np. start=1, end=10) i powinny zawierać liczby całkowite.

Pętla while wykonuje blok kodu dopóki warunek jest spełniony. Jest często stosowana do przetwarzania danych wejściowych wiersz po wierszu, np. poprzez read -r.

Przykład pętli while z wczytywaniem linii:

```
count=0
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

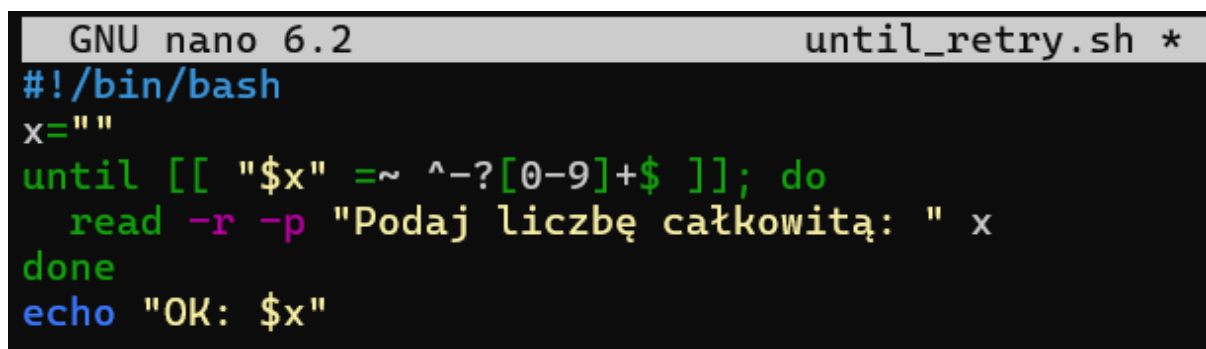


```
while read -r line; do
    [ "$line" = "END" ] && break
    # przetwarzanie line
done
```

Pętla until jest logicznie odwrotna do while: wykonuje blok kodu dopóki warunek nie zostanie spełniony, jej działanie zostało zaprezentowane na rysunkach 6 i 7. Jest użyteczna np. wtedy, gdy dane wejściowe mają być pobierane wielokrotnie aż do uzyskania poprawnej wartości.

Minimalny przykład until dla walidacji liczby:

```
x=""
until [[ "$x" =~ ^-[0-9]+$ ]]; do
    read -r x
done
echo "Poprawna liczba: $x"
```



```
GNU nano 6.2                                until_retry.sh *
#!/bin/bash
x=""
until [[ "$x" =~ ^-[0-9]+$ ]]; do
    read -r -p "Podaj liczbę całkowitą: " x
done
echo "OK: $x"
```

Rys. 6. Skrypt prezentujący działanie pętli until



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ chmod +x until_retry.sh
./until_retry.sh
Podaj liczbę całkowitą: 1
OK: 1
albert@DESKTOP-3NTR2GI:~/skryptowe/lab2$ ./until_retry.sh
Podaj liczbę całkowitą: sdfsd
Podaj liczbę całkowitą: 12.6
Podaj liczbę całkowitą: -9
OK: -9
```

Rys. 7. Przykład działania pętli until

Sterowanie przebiegiem pętli realizowane jest przez:

- break – natychmiastowe przerwanie pętli,
- continue – pominięcie pozostałej części bieżącej iteracji i przejście do następnej.

Ćwiczenia samodzielne

Dopuszcza się wstępną realizację w terminalu, natomiast wymaganym rezultatem jest skrypt Bash automatyzujący rozwiązanie. W przypadku błędnego użycia (zła liczba argumentów / zły typ danych) skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

1. Należy utworzyć skrypt info.sh, który przyjmuje dwa argumenty: imię oraz miasto. Skrypt ma wypisać jedno zdanie zawierające oba argumenty oraz bieżącą datę w formacie YYYY-MM-DD. Wymagane jest poprawne cytowanie argumentów, tak aby wartości zawierające spacje były przetwarzane jako pojedyncze argumenty. W przypadku braku dwóch argumentów skrypt ma wyświetlić krótką informację o sposobie uruchomienia.
2. Należy utworzyć skrypt path_check.sh, który przyjmuje jeden argument będący ścieżką. Skrypt ma rozpoznać, czy wskazana ścieżka jest istniejącym plikiem regularnym, istniejącym katalogiem, czy nie istnieje. Rezultat ma zostać wypisany w postaci jednoznacznego komunikatu. Wymagane jest użycie testów plikowych w Bash.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



3. Należy utworzyć skrypt `calc.sh`, który przyjmuje trzy argumenty: liczba, operator i liczba, np. `5 + 3`. Skrypt ma obsłużyć co najmniej operatory `+`, `-`, `*`, `/` poprzez konstrukcję `case` i wypisać wynik. Operator `*` należy przekazywać jako `"*"` lub `'*'`. Dopuszczalne jest ograniczenie do liczb całkowitych. W przypadku nieznanego operatora skrypt ma wyświetlić informację o poprawnych operatorach. W przypadku dzielenia przez 0 skrypt ma wypisać błąd na `STDERR` i zakończyć działanie kodem niezerowym.
4. Należy utworzyć skrypt `seq_print.sh`, który przyjmuje dwa argumenty: początek i koniec przedziału (liczby całkowite). Skrypt ma wypisać kolejne liczby od początku do końca, każdą w osobnej linii. Wymagane jest użycie pętli `for` oraz poprawna obsługa przypadku, gdy początek jest większy niż koniec (komunikat i zakończenie działania).
5. Należy utworzyć skrypt `count_word.sh`, który przyjmuje jeden argument będący szukanym słowem. Skrypt ma następnie wczytywać linie tekstu z klawiatury aż do wpisania `END`, a na końcu wypisać, ile razy szukane słowo pojawiło się w podanych liniach. W wersji podstawowej należy zliczyć liczbę linii zawierających podany wzorzec co najmniej raz. Wymagane jest użycie `while` oraz `read` i poprawne cytowanie argumentu wyszukiwania.
6. Należy utworzyć skrypt `args_list.sh`, który przyjmuje dowolną liczbę argumentów i wypisuje je kolejno w formie: Argument 1: ..., Argument 2: ... itd. Wymagane jest wykorzystanie `"$@"` i pętli, tak aby argumenty zawierające spacje były traktowane jako jeden element.
7. Należy utworzyć skrypt `classify.sh`, który przyjmuje jedną liczbę całkowitą i klasyfikuje ją jako wartość ujemną, równą zero lub dodatnią. Wymagane jest zastosowanie `if/elif/else` oraz porównań liczbowych. W przypadku braku argumentu skrypt ma wypisać sposób użycia.
8. Należy utworzyć skrypt `mode.sh`, który przyjmuje jeden argument określający tryb działania: `dev`, `test` albo `prod`. W zależności od trybu skrypt ma ustawić zmienną `LOG_LEVEL` (np. `DEBUG/INFO/WARN`) oraz wypisać



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



podsumowanie konfiguracji. Wymagane jest użycie case oraz zmiennych. W przypadku nieznanego trybu skrypt ma wypisać listę dozwolonych wartości. W trybie dev ma być wypisany dodatkowy komunikat diagnostyczny, w trybie prod wyłącznie podsumowanie.

9. Należy utworzyć skrypt `until_retry.sh`, który pobiera od użytkownika wartość z klawiatury i wymusza podanie liczby całkowitej. Skrypt ma wielokrotnie pytać o wartość dopóki nie zostanie wprowadzona poprawna liczba całkowita (pętla `until`). Wymagane jest użycie `read -r` oraz sprawdzanie poprawności formatu liczby w warunku pętli (np. z użyciem `[[ ... ]]` i wyrażenia regularnego). Po podaniu poprawnej wartości skrypt ma wypisać komunikat w formacie `OK: <liczba>` i zakończyć działanie kodem wyjścia 0.
10. Należy utworzyć skrypt `scan_numbers.sh`, który przyjmuje dowolną liczbę argumentów wiersza poleceń i przetwarza je kolejno w pętli. Dla każdego argumentu mają obowiązywać następujące reguły: jeżeli argument ma wartość `SKIP`, bieżąca iteracja ma zostać pominięta (`continue`); jeżeli argument ma wartość `STOP`, przetwarzanie ma zostać przerwane (`break`); jeżeli argument jest liczbą całkowitą, skrypt ma wypisać linię w formacie `NUM: <wartość>`; w przeciwnym razie skrypt ma wypisać ostrzeżenie na `STDERR` w formacie `WARN: niepoprawna wartość '<argument>'`. Wymagane jest poprawne cytowanie argumentów oraz walidacja liczby całkowitej.
11. Należy utworzyć skrypt `args_join.sh`, który przyjmuje dowolną liczbę argumentów, a następnie wypisuje: (1) liczbę argumentów (`$#`), (2) każdy argument w osobnej linii z numeracją, iterując po `"$@"`, oraz (3) wszystkie argumenty połączone w jeden napis z użyciem `"$@"`. W ramach zadania należy uruchomić skrypt co najmniej raz z argumentem zawierającym spację (np. `"Ala ma kota"`) i na tej podstawie krótko porównać zachowanie `"$@"` i `"$@"`.

Pytania pomocnicze

1. Jak w Bash definiuje się zmienną i dlaczego w zapisie `NAZWA=wartość` nie mogą występować spacje wokół znaku `=`?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



2. W jakich sytuacjach zalecane jest użycie formy `${NAZWA}` zamiast `$NAZWA`?
3. Na czym polega różnica między cudzysłowem podwójnym `"..."` a apostrofem `'...'` w kontekście podstawień zmiennych i interpretacji znaków specjalnych?
4. Czym są `word splitting` i `globbing` oraz w jaki sposób brak cytowania zmiennej może zmienić liczbę argumentów przekazywanych do polecenia?
5. Jaką funkcję pełnią parametry pozycyjne `$0`, `$1`, `$2`, ... oraz zmienna `$#`?
6. Jaka jest różnica między `"$@"` i `"$"` oraz kiedy użycie `"$@"` jest konieczne dla poprawności przetwarzania argumentów?
7. W jakim celu w skryptach stosuje się komunikat użycia (`usage`) i dlaczego zaleca się wypisywanie go na `STDERR`?
8. Jakie znaczenie ma kod wyjścia skryptu i dlaczego w przypadku błędnego użycia często stosuje się `exit 2`?
9. Jaka jest różnica między testami `[ ... ]` i `[[ ... ]]` oraz w jakich przypadkach `[[ ... ]]` jest wygodniejsze?
10. Jakie operatory stosuje się w Bash do porównań liczbowych (np. „większe”, „równe”) i dlaczego nie należy używać operatorów tekstowych do porównań liczb?
11. Jak działa instrukcja `case` i dlaczego jest często stosowana zamiast wielu gałęzi `if/elif`?
12. Jakie są podstawowe zastosowania pętli `for` w Bash i czym różni się iteracja po argumentach `"$@"` od iteracji po wyniku polecenia w `$(...)`?
13. Jak działa pętla `while` i w jakim typowym scenariuszu używa się jej razem z `read`?
14. Czym różni się pętla `until` od `while` i w jakim celu może być użyta do walidacji wejścia użytkownika?
15. Jaką rolę pełnią instrukcje `break` i `continue` oraz jaki jest efekt ich użycia wewnątrz pętli?
16. Dlaczego w praktyce zaleca się użycie `read -r` podczas wczytywania linii tekstu?
17. Dlaczego operator `*` w wywołaniu skryptu (np. w kalkulatorze) powinien być przekazywany jako `"*"` lub `'*'`?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Podsumowanie

W trakcie drugiego laboratorium utrwalono elementarne konstrukcje języka Bash wykorzystywane do budowy skryptów sterujących przebiegiem programu. Omówiono zasady definiowania i odczytu zmiennych oraz znaczenie formy `${...}` w kontekstach, w których konieczne jest jednoznaczne wyznaczenie nazwy zmiennej. Szczególny nacisk położono na cytowanie (quoting) jako mechanizm zapewniający poprawne przekazywanie argumentów do poleceń oraz eliminujący typowe błędy wynikające z niezamierzonego dzielenia argumentów i rozwijania wzorców (globbing). Przećwiczono parametryzację skryptów argumentami wiersza poleceń, w tym poprawne użycie `$0`, `$1..$n`, `$#` oraz `"$@"`, a także podstawową walidację minimalnych wymagań wywołania. W części poświęconej sterowaniu przepływem zastosowano instrukcje `if/elif/else` wraz z testami `[ ]` i `[[ ]]` oraz konstrukcję `case` do wyboru wariantu działania. Uzupełnieniem były pętle `for`, `while` i `until`, wykorzystywane w prostych scenariuszach iteracyjnych, w tym z użyciem `break` i `continue`. Rezultatem laboratorium jest umiejętność przygotowania prostych, parametryzowanych skryptów Bash, które w sposób przewidywalny realizują logikę decyzyjną i iteracyjną oraz poprawnie obsługują dane wejściowe przekazywane jako argumenty.

Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:
Programowanie skryptowe
Zajęcia laboratoryjne nr 3

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	3
Ćwiczenia samodzielne	12
Pytania pomocnicze	14
Podsumowanie	15
Literatura	16



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Laboratorium trzecie ma na celu wprowadzenie modularności skryptów poprzez funkcje i uporządkowaną organizację kodu. W ramach zajęć realizowane jest definiowanie funkcji, przekazywanie parametrów, stosowanie zmiennych lokalnych oraz dzielenie skryptu na segmenty odpowiedzialności (np. walidacja danych, logika główna, operacje pomocnicze). Dodatkowo utrwalane jest włączanie kodu zewnętrznego (source) oraz przygotowanie struktury skryptu ułatwiającej rozwój i utrzymanie.

Cel ćwiczenia/laboratorium

- Opanowanie umiejętności definiowania i stosowania funkcji w Bash w celu modularizacji skryptu i eliminacji duplikacji kodu.
- Przekazywanie parametrów do funkcji oraz interpretacja $\$1..\n w kontekście funkcji.
- Stosowanie zmiennych lokalnych (local) w celu ograniczenia efektów ubocznych i utrzymania przewidywalności działania.
- Wydzielenie typowych modułów skryptu (np. inicjalizacja, walidacja parametrów, logika główna, operacje pomocnicze) i uporządkowanie kolejności wykonania.
- Wprowadzenie wielokrotnego wykorzystania kodu poprzez dołączanie plików z funkcjami (source / .) i budowę prostego „modułu” narzędziowego.
- Zastosowanie konwencji dot. odpowiedzialności funkcji (czy funkcja zwraca kodem return, czy generuje wynik na STDOUT).

Instrukcja laboratoryjna/ćwiczeniowa

Funkcje i ponowne wykorzystanie kodu w skryptach powłoki

Funkcje w Bash stanowią podstawowy mechanizm organizacji skryptu w sposób modularny. Zamiast budować jeden, monolityczny blok poleceń, logika może zostać rozdzielona na mniejsze fragmenty o jasno określonej odpowiedzialności. Takie podejście ogranicza duplikację kodu, upraszcza wprowadzanie zmian oraz ułatwia diagnostykę, ponieważ pojedyncza funkcja obejmuje wąski zakres działań i może być



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



testowana niezależnie od pozostałych elementów skryptu. W praktyce funkcje są wykorzystywane do wydzielania operacji powtarzalnych (np. formatowania komunikatów), etapów przetwarzania (np. walidacja danych, obliczenia, przygotowanie podsumowania) oraz do porządkowania kontroli przepływu w skrypcie (np. poprzez centralną funkcję main).

Definiowanie i wywołanie funkcji

Funkcję w Bash definiuje się przez podanie jej nazwy oraz bloku instrukcji ujętego w nawiasy klamrowe. Stosowane są dwa równoważne zapisy: `fname() { ...; }` lub `function fname { ...; }`, przy czym najczęściej wykorzystuje się wariant pierwszy. W ramach laboratoriów jako zalecany i domyślny zapis należy przyjąć formę `fname() { ...; }`, aby utrzymać spójny styl kodu w całym przedmiocie. Ciało funkcji może zawierać dowolne polecenia powłoki, w tym pętle, warunki i wywołania innych funkcji. Wywołanie funkcji polega na podaniu jej nazwy jak zwykłego polecenia, a argumenty przekazywane są analogicznie jak w przypadku uruchamiania skryptu.

```
log_info() {  
    echo "INFO: $1"  
  
}  
  
log_info "Uruchomiono moduł logowania"
```

W kontekście jakości kodu istotne jest utrzymanie jednoznacznych nazw funkcji oraz konsekwentnego stylu ich wywołań. Funkcja powinna realizować jeden, dobrze określony cel, a jej nazwa powinna odzwierciedlać działanie (np. `validate_args`, `calc_age`, `to_upper`, `min_max`).

Parametry funkcji i interpretacja \$1..\$n wewnątrz funkcji

Argumenty przekazane do funkcji są dostępne poprzez parametry pozycyjne, analogicznie jak argumenty skryptu. Wewnątrz funkcji `$1` oznacza pierwszy argument funkcji, `$2` drugi itd., natomiast `$#` oznacza liczbę argumentów przekazanych do



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



funkcji. Parametry `$@` i `$*` reprezentują zbiór argumentów, przy czym w praktyce standardem jest stosowanie `"$@"`, ponieważ zachowuje granice argumentów nawet wtedy, gdy zawierają spacje.

```
args_list() {  
local i=1  
for arg in "$@"; do  
echo "Argument $i: $arg"  
i=$((i+1))  
done  
}
```

Ważną konsekwencją jest to, że przekazując argumenty skryptu do funkcji, należy zachować tę samą zasadę cytowania:

```
main() {  
    args_list "$@"  
}
```

Takie podejście utrzymuje integralność listy argumentów i eliminuje błędy wynikające z niezamierzonego dzielenia argumentów zawierających spacje.

Zmienne lokalne i kontrola zakresu

Bash domyślnie operuje na zmiennych o zasięgu globalnym, co oznacza, że zmienna ustawiona w funkcji jest widoczna również poza funkcją. Taka własność



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

może prowadzić do efektów ubocznych, w szczególności gdy w różnych częściach skryptu używane są te same nazwy zmiennych. W celu ograniczenia tego ryzyka stosuje się zmienne lokalne definiowane słowem kluczowym `local`. Zmienna zadeklarowana jako lokalna istnieje wyłącznie w obrębie funkcji, a jej modyfikacje nie wpływają na stan globalny.

```
counter=0  
inc_counter() {  
  counter=$((counter+1))  
}  
demo_local() {  
  local counter=100  
  counter=$((counter+1))  
  echo "Wewnątrz funkcji: $counter"  
}
```

W powyższym przykładzie zaprezentowanym na rysunkach 1 i 2 funkcja `demo_local` operuje na lokalnym `counter`, dzięki czemu wartość globalna pozostaje niezmienną. Stosowanie `local` jest podstawowym narzędziem budowania przewidywalności skryptu oraz eliminowania błędów wynikających z niejawnego nadpisywania zmiennych.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
GNU nano 6.2 scopes_demo.sh *
#!/bin/bash

counter=0

inc_counter() {
    counter=$((counter + 1))
}

demo_local() {
    local counter=100
    counter=$((counter + 1))
    echo "Wewnątrz demo_local: $counter"
}

echo "Start: $counter"
inc_counter
echo "Po inc_counter: $counter"
demo_local
echo "Po demo_local: $counter"
```

Rys.1. Skrypt przedstawiający różnice między zmienną lokalną a globalną.

```
albert@DESKTOP-3NTR2GI:~/skryptowe$ cd lab3
albert@DESKTOP-3NTR2GI:~/skryptowe/lab3$ nano scopes_demo.sh
albert@DESKTOP-3NTR2GI:~/skryptowe/lab3$ chmod +x scopes_demo.sh
albert@DESKTOP-3NTR2GI:~/skryptowe/lab3$ ./scopes_demo.sh
Start: 0
Po inc_counter: 1
Wewnątrz demo_local: 101
Po demo_local: 1
albert@DESKTOP-3NTR2GI:~/skryptowe/lab3$ |
```

Rys. 2. Wywołanie skryptu prezentującego zakres zmiennych.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Organizacja kodu: moduły odpowiedzialności i funkcja main

Skrypt o rosnącej złożoności wymaga uporządkowania przebiegu wykonania. Powszechną praktyką jest wydzielenie funkcji odpowiadających za konkretne etapy działania: funkcji prezentującej sposób uruchomienia (usage), funkcji walidującej parametry (validate_args), funkcji realizujących obliczenia lub przetwarzanie (calc_age, min_max) oraz funkcji spinającej całość (main). Taki układ ogranicza liczbę zależności między fragmentami kodu i pozwala na utrzymanie przejrzystego „szkieletu” programu.

```
usage() {  
    echo "Użycie: $0 <parametry...>"  
}  
  
validate_args() {  
    [ "$#" -ge 1 ]  
}  
  
main() {  
    if ! validate_args "$@"; then  
        usage  
        exit 1  
    fi  
  
    # logika główna  
}  
  
main "$@"
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



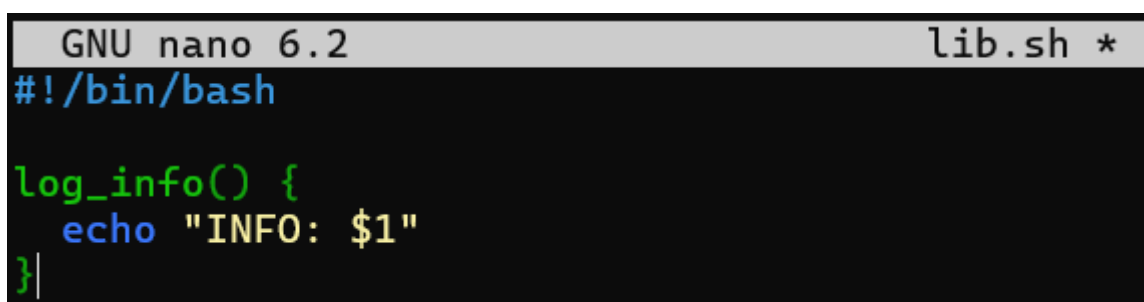
Funkcja `main` pozwala jednoznacznie określić punkt wejścia skryptu oraz utrzymać porządek w kolejności wykonywania etapów, co jest szczególnie istotne w skryptach rozwijanych iteracyjnie. Jeżeli ostatnim poleceniem funkcji jest test lub polecenie, jego kod zakończenia staje się kodem zakończenia funkcji.

Wielokrotne wykorzystanie kodu: `source` i pliki bibliotek

W większych skryptach naturalnie pojawia się potrzeba ponownego wykorzystania tych samych funkcji w wielu plikach. W tym celu stosuje się dołączanie kodu zewnętrznego, najczęściej przez polecenie `source` (równoważnie `.`). Dołączany plik jest wykonywany w bieżącym kontekście powłoki, dzięki czemu zdefiniowane w nim funkcje i zmienne stają się dostępne dla skryptu głównego. W poniższym przykładzie zapis `source "./lib.sh"` jest poprawny w scenariuszu, w którym bieżący katalog roboczy jest zgodny z lokalizacją pliku, jednak w narzędziach produkcyjnych zaleca się wyznaczanie ścieżki do biblioteki niezależnie od CWD

```
source "./lib.sh"
```

```
log_info "Załadowano bibliotekę"
```



```
GNU nano 6.2 lib.sh *
#!/bin/bash

log_info() {
  echo "INFO: $1"
}
```

Rys. 3. Przykład pliku biblioteki z definicją funkcji pomocniczej.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
GNU nano 6.2 app_bad.sh
#!/bin/bash

source "../lib/lib.sh"

log_info "Załadowano bibliotekę"
```

Rys. 4. Przykład niepoprawnego załadowania biblioteki

```
GNU nano 6.2 app.sh
#!/bin/bash

SCRIPT_DIR="$(cd "$(dirname "$0")" && pwd)"
source "$SCRIPT_DIR/../lib/lib.sh"

log_info "Załadowano bibliotekę"
```

Rys. 5. Przykład poprawnego załadowania biblioteki

```
albert@DESKTOP-3NTR2GI:~/skryptowe$ ./lab3/bin/app_bad.sh
./lab3/bin/app_bad.sh: line 3: ../lib/lib.sh: No such file or directory
./lab3/bin/app_bad.sh: line 5: log_info: command not found
albert@DESKTOP-3NTR2GI:~/skryptowe$ ./lab3/bin/app.sh
INFO: Załadowano bibliotekę
albert@DESKTOP-3NTR2GI:~/skryptowe$ cd /
/home/albert/skryptowe/lab3/bin/app.sh
INFO: Załadowano bibliotekę
albert@DESKTOP-3NTR2GI:/$ |
```

Rys. 6. Przykład uruchomienia skryptów ładujących biblioteki.

Rysunki 3–6 przedstawiają przykład podziału programu na bibliotekę funkcji oraz skrypt główny. Na rysunku 4 pokazano niepoprawny sposób dołączania biblioteki, który zależy od bieżącego katalogu roboczego, natomiast rysunek 5 prezentuje



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



rozwiązanie wykorzystujące katalog skryptu, dzięki czemu program działa poprawnie niezależnie od miejsca uruchomienia.

Aby zachować przewidywalność działania, istotne jest, by dołączany plik pełnił rolę biblioteki (zawierał definicje funkcji i ewentualnie stałe), a nie uruchamiał samodzielnie logiki programu. Przy organizacji katalogów często stosuje się rozdzielenie na katalog z funkcjami pomocniczymi (np. lib/) oraz katalog ze skryptami uruchamialnymi (np. bin/). W takim układzie kluczowe jest wyznaczenie ścieżki do biblioteki w sposób niezależny od aktualnego katalogu roboczego, tak aby dołączenie działało niezależnie od miejsca uruchomienia skryptu.

Konwencje zwracania wyniku: return oraz wynik na standardowym wyjściu

W Bash funkcje mogą przekazywać informację zwrotną na dwa podstawowe sposoby, które należy konsekwentnie rozróżniać. Po pierwsze, funkcja może sygnalizować powodzenie lub niepowodzenie przez kod zakończenia zwracany poleceniem return. Polecenie return może zwracać jedynie kod w zakresie 0 - 255, dlatego nie służy do zwracania wartości tekstowych ani dużych liczb. Jest to szczególnie użyteczne dla funkcji walidujących, które mają odpowiedzieć na pytanie typu „czy warunek jest spełniony”. Na rysunku 7 i 8 funkcja `is_integer` zwraca informację o poprawności danych w postaci kodu zakończenia, natomiast funkcja `to_upper` zwraca wynik przetwarzania przez standardowe wyjście.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
GNU nano 6.2 return_vs_stdout.sh *
#!/bin/bash

is_integer() {
    [[ "$1" =~ ^-?[0-9]+$ ]]
}

to_upper() {
    printf "%s" "$1" | tr '[:lower:]' '[:upper:]'
}

is_integer "123"
echo "Kod po is_integer 123: $?"

is_integer "abc"
echo "Kod po is_integer abc: $?"

result="$(to_upper "Jan Kowalski")"
echo "Wynik funkcji to_upper: $result"
```

Rys. 7. Przykład funkcji zwracającej status wykonania oraz funkcji zwracającej wynik przez standardowe wyjście.\

```
Skrypty
albert@DESKTOP-3NTR2GI:~$ nano return_vs_stdout.sh
albert@DESKTOP-3NTR2GI:~$ chmod +x return_vs_stdout.sh
albert@DESKTOP-3NTR2GI:~$ ./return_vs_stdout.sh
Kod po is_integer 123: 0
Kod po is_integer abc: 1
Wynik funkcji to_upper: JAN KOWALSKI
albert@DESKTOP-3NTR2GI:~$
```

Rys. 8. Przykład uruchomienia funkcji i obserwacji kodu zakończenia oraz wyniku zwróconego przez funkcję.

W poniższym wzorcu wynik warunku jest od razu kodem zakończenia funkcji: powodzenie oznacza „prawda”, a niepowodzenie „fałsz”.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
is_integer() {  
    [[ "$1" =~ ^-?[0-9]+$ ]]  
}
```

Po drugie, funkcja może zwracać dane poprzez standardowe wyjście, wypisując wynik, który następnie może zostać przechwycony do zmiennej za pomocą podstawienia polecenia `$(...)`. Takie podejście jest właściwe dla funkcji, których celem jest wyliczenie wartości (np. konwersja tekstu, obliczenie minimum i maksimum).

```
to_upper() {  
    printf "%s" "$1" | tr '[:lower:]' '[:upper:]'  
}  
  
result="$(to_upper "Jan Kowalski")"
```

Konwencja powinna być dobrana zgodnie z przeznaczeniem funkcji: funkcje walidujące i kontrolne zwracają status, natomiast funkcje obliczeniowe zwracają wynik jako tekst. Zachowanie takiego podziału zwiększa czytelność i pozwala budować przewidywalne, modularne skrypty.

Ćwiczenia samodzielne

1. Należy utworzyć skrypt `utils_demo.sh`, w którym zostaną zdefiniowane trzy funkcje: `log_info`, `log_warn` oraz `log_error`. Każda funkcja ma przyjmować jeden argument będący treścią komunikatu i wypisywać go w jednolitym formacie, np. z prefiksem `INFO:`, `WARN:` lub `ERROR:`. Wymagane jest, aby argument był poprawnie cytowany, a w skrypcie znalazły się co najmniej trzy wywołania tych funkcji prezentujące działanie.
2. Należy utworzyć skrypt `string_ops.sh`, w którym zostanie zdefiniowana funkcja `to_upper`. Funkcja ma przyjmować jeden argument tekstowy i zwracać jego



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



wersję pisaną wielkimi literami poprzez standardowe wyjście. Następnie wynik funkcji ma zostać przechwycony do zmiennej i wykorzystany w dalszym wypisywaniu komunikatu. Wymagane jest zaprezentowanie działania dla przykładu zawierającego spacje w argumencie.

3. Należy utworzyć skrypt `validators.sh`, w którym zostanie zdefiniowana funkcja `is_integer`. Funkcja ma przyjmować jeden argument i sprawdzać, czy jest on liczbą całkowitą (dopuszczalny znak minus na początku). Funkcja ma zwracać informację o wyniku poprzez kod zakończenia (`return 0` dla „tak”, `return 1` dla „nie”). W skrypcie należy wykonać testy dla co najmniej trzech wartości (poprawna liczba, tekst, liczba z minusem) i wypisać wynik w formie komunikatów. Dopuszcza się zarówno jawne użycie `return 0` lub `return 1`, jak również pozostawienie testu jako ostatniego polecenia funkcji, o ile funkcja zwraca poprawny kod zakończenia.
4. Należy utworzyć skrypt `scopes.sh`, w którym zostanie zdefiniowana zmienna globalna `counter=0` oraz funkcja `inc_counter`. Funkcja ma zwiększać licznik o 1. Następnie należy dopisać drugą funkcję `demo_local`, w której zostanie użyta zmienna o tej samej nazwie `counter`, ale zadeklarowana jako `local`, tak aby modyfikacje wewnątrz funkcji nie wpływały na zmienną globalną. Wymagane jest zaprezentowanie w terminalu, że po wywołaniu `demo_local` wartość globalna pozostała niezmienną.
5. Należy utworzyć skrypt `main_modular.sh`, który realizuje prosty scenariusz: pobiera dwa argumenty `imie` i `rok_urodzenia`, a następnie wypisuje podsumowanie zawierające imię oraz wiek obliczony na podstawie bieżącego roku. Wymagane jest rozdzielenie logiki na funkcje: `usage` (opis sposobu uruchomienia), `validate_args` (walidacja liczby argumentów i formatu roku), `calc_age` (obliczenie wieku i zwrócenie wyniku), `main` (wywołanie całości w poprawnej kolejności). Skrypt ma uruchamiać się poprzez wywołanie funkcji `main "$@"`. Bieżący rok może zostać pobrany przez `date +%Y`, a walidacja roku urodzenia powinna sprawdzać, że argument jest liczbą całkowitą. Jeżeli



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



rok urodzenia jest większy od bieżącego roku, skrypt powinien zakończyć działanie komunikatem błędu.

6. Należy utworzyć skrypt `joiner.sh`, w którym zostanie zdefiniowana funkcja `join_by`. Funkcja ma przyjmować jako pierwszy argument separator (np. `.`), a następnie dowolną liczbę kolejnych argumentów. Funkcja ma zwrócić na standardowe wyjście pojedynczy tekst, w którym argumenty zostały połączone separatorem. Wymagane jest przetestowanie działania dla co najmniej trzech argumentów, w tym jednego zawierającego spację. Po odczytaniu separatora (pierwszego argumentu) należy łączyć wyłącznie pozostałe argumenty przekazane do funkcji.
7. Należy utworzyć plik `lib.sh`, w którym zostaną umieszczone co najmniej cztery funkcje pomocnicze z wcześniejszych zadań (np. `log_info`, `log_error`, `is_integer`, `to_upper`). Następnie należy utworzyć skrypt `app.sh`, który dołącza `lib.sh` poprzez `source` i wykorzystuje te funkcje do wykonania krótkiego scenariusza demonstracyjnego. Wymagane jest, aby `app.sh` działał poprawnie po uruchomieniu z katalogu, w którym znajdują się oba pliki.
8. Należy utworzyć strukturę katalogów `lab3/` zawierającą podkatalog `lib/` na pliki z funkcjami oraz `bin/` na skrypty uruchamialne. Plik z funkcjami ma zostać umieszczony w `lab3/lib/`, a skrypt uruchamialny w `lab3/bin/`. W skrypcie uruchamialnym należy dołączyć plik biblioteki w sposób niewrażliwy na aktualny katalog roboczy, tzn. dołączenie ma działać niezależnie od tego, skąd uruchomiony zostanie skrypt. Wymagane jest zaprezentowanie co najmniej dwóch uruchomień: z katalogu `lab3/bin/` oraz z katalogu nadrzędnego.
9. Należy utworzyć skrypt `stats.sh`, w którym zostanie zdefiniowana funkcja `min_max`. Funkcja ma przyjmować listę liczb całkowitych jako argumenty i zwracać na standardowe wyjście dwie wartości w jednej linii: minimum i maksimum oddzielone spacją. Następnie w skrypcie należy przechwycić wynik funkcji do zmiennej i rozdzielić go na dwie zmienne (`min` i `max`), po czym wypisać podsumowanie. Wymagane jest przetestowanie na co najmniej



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



dwóch zestawach danych. Rozdzielenie wartości może zostać wykonane np. przez `read -r min max <<< "$result"`.

10. Należy przygotować zestaw funkcji w pliku biblioteki w taki sposób, aby obejmował co najmniej: funkcję logowania, funkcję walidującą argumenty oraz funkcję realizującą przekształcenie tekstu. Następnie należy utworzyć skrypt demonstracyjny, w którym logika działania jest w całości oparta o te funkcje, a kod główny ogranicza się do kolejności wywołań i przekazania parametrów. Wymagane jest zastosowanie jednej, konsekwentnej konwencji: funkcje zwracające dane mają zwracać je przez standardowe wyjście, natomiast funkcje walidujące mają zwracać status kodem return.

Pytania pomocnicze

1. Jak definiuje się funkcję w Bash i czym różnią się zapisy `fname() { ...; }` oraz `function fname { ...; }`?
2. W jaki sposób wywoływana jest funkcja i jak przekazywane są do niej argumenty?
3. Co oznaczają parametry `$1..$n`, `$#`, `"$@"` wewnątrz funkcji i czym różnią się od tych samych parametrów w skrypcie głównym?
4. Dlaczego zaleca się przekazywanie argumentów do funkcji w postaci `func "$@"` zamiast `func $@`?
5. Jaka jest różnica między zmienną globalną a lokalną w Bash i jaki jest cel stosowania `local`?
6. Co jest efektem ubocznym w kontekście funkcji i w jaki sposób `local` pomaga go ograniczać?
7. W jakich przypadkach funkcja powinna „zwracać” informację przez kod zakończenia (`return`), a w jakich przez standardowe wyjście (`STDOUT`)?
8. Jakie ograniczenie ma `return` w Bash i dlaczego nie służy do zwracania dużych wartości liczbowych ani tekstowych?
9. Jak przechwycić wynik funkcji wypisywany na `STDOUT` do zmiennej i dlaczego w takim przypadku zwykle stosuje się cytowanie?
10. Co robi polecenie `source` (lub `.`) i czym różni się od uruchomienia pliku jako osobnego procesu?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



11. Jakie zasady powinien spełniać plik biblioteki dołączany przez source, aby nie uruchamiał „logiki programu” przy imporcie?
12. Dlaczego dołączanie plików przez ścieżkę względną może być problematyczne i w jaki sposób można zapewnić niezależność od katalogu roboczego?
13. Na czym polega modularny podział skryptu na etapy (inicjalizacja, walidacja, logika główna, funkcje pomocnicze) i dlaczego main "\$@" jest użytecznym wzorcem?
14. W jaki sposób uzasadnia się zasadę „jedna funkcja = jedna odpowiedzialność” w skryptach powłoki?
15. Jakie są typowe zastosowania wspólnej biblioteki funkcji (np. logowanie, walidacja, kontrola zależności) w większej liczbie skryptów?

Podsumowanie

W trakcie trzeciego laboratorium wprowadzono modularny sposób organizacji skryptów Bash poprzez definiowanie i stosowanie funkcji. Utrwalono, że funkcje umożliwiają wydzielenie powtarzalnych operacji oraz etapów przetwarzania do niezależnych bloków kodu o jasno określonej odpowiedzialności, co ogranicza duplikację i ułatwia utrzymanie. Przeciwczono przekazywanie argumentów do funkcji i interpretację parametrów pozycyjnych w kontekście funkcji, w tym poprawne przekazywanie listy argumentów z użyciem "\$@". Omówiono znaczenie zmiennych lokalnych deklarowanych przez local jako podstawowego mechanizmu ograniczania efektów ubocznych i zapewnienia przewidywalności działania skryptu. Wprowadzono również wielokrotne wykorzystanie kodu poprzez dołączanie plików z funkcjami za pomocą source, przy założeniu rozdzielenia definicji funkcji od logiki uruchomieniowej. Całość domknęto przez ujednolicenie konwencji odpowiedzialności funkcji: funkcje walidujące sygnalizują wynik kodem zakończenia, natomiast funkcje obliczeniowe zwracają dane na standardowym wyjściu, co pozwala budować spójne i skalowalne narzędzia skryptowe.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Materiały do przedmiotu:

Programowanie skryptowe

Zajęcia laboratoryjne nr 4

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	3
Ćwiczenia samodzielne	10
Pytania pomocnicze	15
Podsumowanie	15
Literatura	16



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Laboratorium czwarte ma na celu opanowanie przetwarzania danych za pomocą strumieni oraz operacji na plikach w powłocie. W ramach zajęć realizowane jest stosowanie przekierowań (>, >>, <, 2>), potoków (|) oraz narzędzi umożliwiających budowę łańcuchów przetwarzania (np. cat, grep, wc, tee). Utrwalane jest także czytanie danych z plików i standardowego wejścia w sposób strumieniowy, tak aby skrypty mogły działać jako element większych pipeline'ów.

Cel ćwiczenia/laboratorium

- Opanowanie pracy na plikach i katalogach w skryptach, w tym poprawne rozróżnianie ścieżek względnych i bezwzględnych oraz kontrola kontekstu katalogu roboczego.
- Stosowanie przekierowań wejścia i wyjścia w celu zapisu wyników do plików, dopisywania danych oraz kierowania komunikatów diagnostycznych do osobnych strumieni.
- Budowanie łańcuchów przetwarzania danych z użyciem potoków, gdzie wynik jednego narzędzia stanowi wejście kolejnego etapu.
- Wykorzystanie strumieniowego przetwarzania danych (bez wczytywania całości do pamięci), w szczególności przy pracy na plikach tekstowych i logach.
- Realizacja scenariuszy obejmujących selekcję danych, agregację wyników oraz generowanie raportów na podstawie zawartości plików.
- Utrwalenie zasad projektowania skryptu jako elementu pipeline'u, tj. tak, aby mógł przyjmować dane ze standardowego wejścia i zwracać wynik na standardowe wyjście.

Instrukcja laboratoryjna/ćwiczeniowa

Standardowe strumienie: STDIN, STDOUT, STDERR

Proces uruchomiony w systemie Unix/Linux standardowo operuje na trzech strumieniach: standardowym wejściu (STDIN), standardowym wyjściu (STDOUT) oraz standardowym wyjściu diagnostycznym (STDERR). STDIN jest źródłem danych wejściowych (np. z klawiatury, pliku lub potoku), STDOUT służy do emitowania



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

danych wynikowych przeznaczonych do dalszego przetwarzania, natomiast STDERR jest przeznaczony na komunikaty diagnostyczne, ostrzeżenia i błędy. Rozdzielenie STDOUT i STDERR jest kluczowe w automatyzacji, ponieważ pozwala niezależnie przekierować dane wynikowe do pliku lub kolejnego etapu pipeline'u, zachowując jednocześnie widoczność błędów.

W praktyce poprawnie zaprojektowany skrypt powinien emitować raport lub dane na STDOUT, a komunikaty o błędach i nieprawidłowym użyciu na STDERR, tak aby nie zanieczyszczać danych wykorzystywanych dalej w potokach.

Ścieżki względne i bezwzględne oraz znaczenie bieżącego katalogu roboczego

W systemie Unix/Linux dostęp do plików i katalogów odbywa się za pomocą ścieżek. Mogą one być bezwzględne albo względne. Ścieżka bezwzględna zaczyna się od znaku / i wskazuje pełne położenie pliku w systemie plików, na przykład:

```
/home/user/lab4/input/notes.txt
```

Taka ścieżka zawsze oznacza to samo miejsce, niezależnie od tego, z jakiego katalogu zostało uruchomione polecenie lub skrypt. Z kolei ścieżka względna nie zaczyna się od / i jest interpretowana względem bieżącego katalogu roboczego, czyli katalogu, w którym aktualnie działa proces. Przykładem ścieżki względnej jest:

```
input/notes.txt
```

Jej znaczenie zależy więc od miejsca uruchomienia polecenia.

Bieżący katalog roboczy można sprawdzić poleceniem:

```
pwd
```

Ma to duże znaczenie praktyczne, ponieważ zmiana katalogu roboczego wpływa na interpretację wszystkich ścieżek względnych. Oznacza to, że to samo polecenie może w jednym przypadku działać poprawnie, a w innym zgłaszać błąd. Dobrze pokazuje to następujące porównanie:

```
cat input/notes.txt
```

```
cat /home/user/lab4/input/notes.txt
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Oba polecenia mogą odczytywać ten sam plik, ale pierwsze zadziała poprawnie tylko wtedy, gdy bieżącym katalogiem roboczym jest katalog lab4. Drugie polecenie korzysta ze ścieżki bezwzględnej, dlatego nie zależy od miejsca uruchomienia.

W praktyce skryptowej ścieżki względne są wygodne wtedy, gdy wiadomo, z jakiego katalogu działa skrypt, natomiast ścieżki bezwzględne są bardziej jednoznaczne i bezpieczne. Z tego powodu w wielu skryptach najpierw ustala się położenie samego pliku skryptu, a następnie na tej podstawie wyznacza katalog projektu. Pozwala to budować ścieżki w sposób kontrolowany i powtarzalny. Typowy przykład wygląda następująco:

```
SCRIPT_DIR="$(cd -- "$(dirname -- "${BASH_SOURCE[0]}")" && pwd)"  
LAB_DIR="$(cd -- "${SCRIPT_DIR}/.." && pwd)"
```

W tym fragmencie `${BASH_SOURCE[0]}` oznacza ścieżkę do aktualnie wykonywanego skryptu, `dirname` wyodrębnia katalog, w którym ten skrypt się znajduje, a `cd ... && pwd` zamienia tę informację na pełną ścieżkę bezwzględną. W rezultacie zmienna `SCRIPT_DIR` przechowuje ścieżkę do katalogu, w którym leży skrypt, natomiast `LAB_DIR` wskazuje katalog nadrzędny, czyli główny katalog projektu.

Jeżeli skrypt `make_report.sh` znajduje się w katalogu:

```
/home/student/lab4/bin
```

to po wykonaniu:

```
echo "$SCRIPT_DIR"
```

otrzymamy:

```
/home/student/lab4/bin
```

natomiast po wykonaniu:

```
echo "$LAB_DIR"
```

otrzymamy:

```
/home/student/lab4
```

Po ustaleniu katalogu projektu można przejść do niego jawnie poleceniem:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
cd "$LAB_DIR"
```

Od tego momentu ścieżki względne będą interpretowane względem katalogu lab4. Oznacza to, że zapis:

```
REPORT_FILE="output/report.txt"
```

jest ścieżką względną, która po wcześniejszym wykonaniu `cd "$LAB_DIR"` będzie oznaczała plik:

```
/home/student/lab4/output/report.txt
```

Można także od razu zbudować ścieżkę bezwzględną na podstawie wyznaczonego katalogu projektu:

```
ABS_REPORT_FILE="$LAB_DIR/output/report.txt"
```

Wtedy zmienna `ABS_REPORT_FILE` zawiera pełną ścieżkę do pliku i nie zależy od bieżącego katalogu roboczego. W praktyce oznacza to, że zapis:

```
REPORT_FILE="output/report.txt"
```

jest wygodny, jeśli wcześniej przejdziemy do katalogu projektu, natomiast zapis:

```
ABS_REPORT_FILE="$LAB_DIR/output/report.txt"
```

jest jego jednoznacznym odpowiednikiem bezwzględnym.

Przekierowania wejścia i wyjścia: >, >>, <, 2>, 2>&1

Powłoka Bash umożliwia przekierowanie strumieni do plików lub z plików. Operator `>` przekierowuje STDOUT do pliku, tworząc go lub nadpisując. Operator `>>` dopisuje dane do istniejącego pliku bez kasowania jego zawartości. Operator `<` przekierowuje plik na STDIN polecenia, dzięki czemu narzędzie czyta dane tak, jakby były podane z klawiatury.

Należy odróżnić przekierowanie strumienia od sposobu zapisu ścieżki do pliku. Operatory `>`, `>>`, `<`, `2>` oraz podobne określają, który strumień ma zostać przekierowany, natomiast sama ścieżka pliku podlega zwykłym zasadom interpretacji ścieżek przez powłokę. Oznacza to, że plik docelowy lub źródłowy może zostać wskazany zarówno ścieżką względną, jak i bezwzględną.

Przykłady:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- `date > output/report.txt`
- `date > /home/user/lab4/output/report.txt`
- `wc -l < input/numbers.txt`
- `wc -l < /home/user/lab4/input/numbers.txt`

Częstym źródłem błędów w pracy z przekierowaniami nie jest sam operator `>` lub `>>`, lecz błędnie założony katalog roboczy. Jeżeli polecenie zapisuje do `output/report.txt`, to plik zostanie utworzony w katalogu `output` liczonym względem bieżącego katalogu roboczego, a nie względem lokalizacji pliku skryptu. Z tego powodu skrypty przeznaczone do wielokrotnego uruchamiania powinny jawnie ustalać katalog pracy albo wyznaczać ścieżki na podstawie własnej lokalizacji.

W każdej z powyższych par operator przekierowania działa tak samo, a zmienia się jedynie sposób wskazania pliku. W przypadku ścieżki względnej wynik zależy od bieżącego katalogu roboczego, natomiast w przypadku ścieżki bezwzględnej położenie pliku jest określone jednoznacznie.

Przykłady:

- `echo "Naglowek" > output/report.txt`
- `date >> output/report.txt`
- `wc -l < input/numbers.txt`

Strumień `STDERR` może zostać zapisany osobno do pliku poprzez `2>`, natomiast zapis `2>&1` kieruje `STDERR` do tego samego miejsca, do którego aktualnie skierowany jest `STDOUT`. Działanie strumieni zostało zaprezentowane na rysunku 1. W przypadku łączenia strumieni istotna jest kolejność przekierowań, ponieważ powłoka interpretuje je od lewej do prawej.

Przykłady:

- `cat input/notes.txt input/nie_ma.txt > output/stdout.txt 2> output/stderr.txt`
- `cat input/notes.txt input/nie_ma.txt > output/all.txt 2>&1`

```
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ cat input/notes.txt input/nie_ma.txt > output/stdout.txt 2> output/stderr.txt
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ cat output/stdout.txt
Ala ma kota
To jest przykładowy plik
Koniec danych
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ cat output/stderr.txt
cat: input/nie_ma.txt: No such file or directory
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ |
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Rys. 1. Przykład przekierowania strumieni

Potoki: | i budowa łańcuchów przetwarzania

Potok (pipeline) pozwala połączyć programy w taki sposób, aby STDOUT jednego programu stało się STDIN programu następnego. Mechanizm ten umożliwia budowę strumieniowych łańcuchów przetwarzania, w których każdy etap wykonuje pojedynczą, jasno określoną operację (np. filtrację, selekcję, zliczenie).

Przykład:

```
cat input/app.log | grep "ERROR" | wc -l
```

lub równoważnie:

```
grep "ERROR" input/app.log | wc -l
```

W podejściu inżynierskim zaleca się projektowanie pipeline'u jako sekwencji prostych etapów. Dzięki temu każde narzędzie jest łatwe do testowania, a całość można rozwijać przez dodawanie kolejnych etapów bez modyfikowania wcześniejszych elementów.

tee jako rozgałęzianie strumienia do pliku i na ekran

Polecenie tee umożliwia zapis strumienia do pliku przy jednoczesnym wypisywaniu go na STDOUT, jego działanie zostało zaprezentowane na rysunku 2. Jest to użyteczne w sytuacjach, gdy dane mają zostać zachowane w pliku (np. do audytu lub analizy), a jednocześnie mają być widoczne w terminalu lub przekazane do dalszego etapu pipeline'u. Wariant z dopisywaniem realizuje opcja tee -a.

Przykład klasycznego użycia polecenia tee:

```
grep "WARN" input/app.log | tee output/warn_only.txt | head
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ cat > input/app.log <<'EOF'
INFO Start programu
WARN Brak pliku konfiguracyjnego
INFO Próba uruchomienia modułu
ERROR Nie udało się połączyć
WARN Użyto wartości domyślnej
INFO Koniec działania
EOF
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ grep "WARN" input/app.log | tee output/warn_only.txt
WARN Brak pliku konfiguracyjnego
WARN Użyto wartości domyślnej
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ cat output/warn_only.txt
WARN Brak pliku konfiguracyjnego
WARN Użyto wartości domyślnej
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ |
```

Rys. 2. Przykład rozdwojenia strumienia poleceniem tee.

Skrypt jako element pipeline'u: wejście ze STDIN, wynik na STDOUT

Skrypt powłoki może być zaprojektowany jako filtr działający strumieniowo, tj. przyjmujący dane ze standardowego wejścia i emitujący wynik na standardowe wyjście, co zostało zaprezentowane na rysunku 3. Takie podejście pozwala traktować skrypt jak standardowe narzędzie systemowe i włączać go w potoki bez zależności od konkretnych plików wejściowych.

Minimalny wzorzec filtra:

```
#!/bin/bash
while read -r line; do
case "$line" in
*ERROR*) printf "%s\n" "$line" ;;
esac
done
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

W poprawnie zaprojektowanym narzędziu strumieniowym:

- dane wynikowe trafiają na STDOUT,
- komunikaty diagnostyczne trafiają na STDERR,
- kod wyjścia sygnalizuje powodzenie lub błąd (np. 2 dla błędnego użycia).

Taki kontrakt umożliwia łączenie narzędzi w większe pipeline'y oraz automatyzację przetwarzania danych w sposób powtarzalny i kontrolowany.

```
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ cat > bin/only_errors.sh <<'EOF'
#!/bin/bash
grep '^ERROR'
EOF
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ chmod +x bin/only_errors.sh
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ cat input/app.log | bin/only_errors.sh
ERROR Nie udało się połączyć
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$ printf "INFO ok\nERROR błąd testowy\nWARN uwaga\n"
| bin/only_errors.sh
ERROR błąd testowy
albert@DESKTOP-3NTR2GI:~/skryptowe/lab4$
```

Rys. 3. Przykład użycia skryptu jako filtra w potoku danych. Skrypt przetwarza dane ze standardowego wejścia i zwraca wyłącznie linie zawierające komunikaty błędów.

Ćwiczenia samodzielne

Dopuszcza się wstępną realizację zadań w terminalu, natomiast wymaganym rezultatem jest skrypt Bash automatyzujący rozwiązanie.

1. Należy utworzyć katalog lab4 oraz w jego obrębie podkatalogi input, output, tmp oraz bin. Następnie należy utworzyć w input plik notes.txt zawierający co najmniej 12 linii dowolnego tekstu (mogą to być krótkie zdania). W ramach zadania należy wykonać zestaw poleceń, które potwierdzą aktualny katalog roboczy, a następnie przeprowadzić odczyt pliku notes.txt raz używając ścieżki względnej, a raz bezwzględnej. Wymagane jest, aby w terminalu zostało pokazane, że oba odczyty wskazują na ten sam plik.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dodatkowo należy przygotować skrypt `bin/env_paths.sh`, który wypisze bieżący katalog roboczy oraz zaprezentuje odczyt `notes.txt` w obu wariantach (względny i bezwzględny), tak aby odtwarzalność zadania nie zależała od ręcznie wpisywanych poleceń.

2. Należy przygotować w katalogu `output` plik `report.txt`, do którego zostaną zapisane wyniki trzech poleceń uruchomionych kolejno. Pierwsze polecenie ma utworzyć plik i zapisać w nim nagłówek raportu. Drugie polecenie ma dopisać bieżącą datę i godzinę. Trzecie polecenie ma dopisać listę plików znajdujących się w katalogu `input`. Wymagane jest użycie `>` do utworzenia/nadpisania raportu oraz `>>` do dopisywania kolejnych sekcji bez kasowania poprzednich.

Wymagane jest wykonanie powyższego jako skrypt `bin/make_report.sh`, uruchamiany w sposób powtarzalny, tak aby ponowne uruchomienie zawsze generowało aktualną wersję raportu.

3. Należy wykonać polecenie, które z jednej strony wypisze poprawny wynik, a z drugiej wygeneruje błąd (np. odczyt kilku ścieżek, z których jedna nie istnieje), przykład polecenia: `cat input/notes.txt input/nie_ma.txt` albo `ls input input/nieistnieje`. Następnie należy uruchomić to polecenie tak, aby standardowe wyjście zostało zapisane do `output/stdout.txt`, a wyjście diagnostyczne do `output/stderr.txt`. Wymagane jest sprawdzenie zawartości obu plików i potwierdzenie, że komunikat błędu znajduje się wyłącznie w pliku dla `STDERR`.

Wymagane jest przygotowanie skryptu `bin/split_streams.sh`, który realizuje to zadanie w sposób automatyczny: wykonuje polecenie, zapisuje oba strumienie do właściwych plików oraz na końcu wypisuje krótką informację, gdzie znajdują się wyniki.

4. Należy utworzyć w katalogu `input` plik `numbers.txt` zawierający co najmniej 20 liczb całkowitych, każdą w osobnej linii. Następnie należy zliczyć liczbę linii w pliku na dwa sposoby: (1) podając nazwę pliku jako argument narzędzia zliczającego, oraz (2) wykorzystując przekierowanie wejścia `<`. Wymagane



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



jest, aby oba sposoby dały identyczny wynik i zostały zaprezentowane w terminalu.

Wymagane jest przygotowanie skryptu `bin/count_lines_two_ways.sh`, który wykona oba warianty i wypisze oba wyniki w czytelnej formie.

5. Należy przygotować plik `input/app.log` zawierający co najmniej 25 linii, w tym linie z frazami `INFO`, `WARN`, `ERROR` (dowolna treść). Następnie należy wykonać polecenia oparte o potoki, które zliczą liczbę wierszy z `ERROR`, zliczą liczbę wierszy z `WARN` oraz wypiszą do `output/errors_only.txt` wyłącznie linie zawierające `ERROR`. Wymagane jest zastosowanie potoków z `grep` oraz `wc -l`.

Wymagane jest ujęcie powyższej procedury w skrypt `bin/log_filter_stats.sh`, który po uruchomieniu generuje plik `errors_only.txt` oraz wypisuje w terminalu zliczenia.

Przykład pliku `app.log`:

```
cat > input/app.log <<'EOF'

2026-03-01 08:00:00 INFO Start aplikacji

2026-03-01 08:00:01 INFO Wczytano konfiguracje z /etc/app/app.conf

2026-03-01 08:00:02 WARN Brak wpisu CACHE_SIZE, uzyto domyslnej wartosci=256

2026-03-01 08:00:03 INFO Ustawiono tryb pracy: prod

2026-03-01 08:00:04 INFO Polaczono z baza danych host=db01 port=5432

2026-03-01 08:00:05 ERROR Nie udalo sie otworzyc pliku /var/data/input.csv

2026-03-01 08:00:06 INFO Ponowienie operacji odczytu pliku

2026-03-01 08:00:07 WARN Wolna odpowiedz serwera api01 (timeout=2s)

2026-03-01 08:00:08 INFO Zapisano wynik posredni do /tmp/app.tmp

2026-03-01 08:00:09 ERROR Blad walidacji rekordu id=17 pole=email
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



2026-03-01 08:00:10 INFO Kontynuacja przetwarzania po walidacji

2026-03-01 08:00:11 WARN Niski poziom miejsca na dysku / (free=5%)

2026-03-01 08:00:12 INFO Wyslano zadanie do kolejki workerq

2026-03-01 08:00:13 ERROR Nieoczekiwany kod odpowiedzi HTTP 503 endpoint=/health

2026-03-01 08:00:14 INFO Proba ponowienia zapytania do endpoint=/health

2026-03-01 08:00:15 INFO Odpowiedz HTTP 200 endpoint=/health

2026-03-01 08:00:16 WARN Zwiekszona liczba retry=3 dla endpoint=/orders

2026-03-01 08:00:17 INFO Przetworzono batch=1 rekordow=100

2026-03-01 08:00:18 INFO Przetworzono batch=2 rekordow=100

2026-03-01 08:00:19 ERROR Nie mozna zapisac wyniku do /var/output/result.json (Permission denied)

2026-03-01 08:00:20 INFO Ustawiono katalog wyjsciowy na /tmp/out

2026-03-01 08:00:21 WARN Wykryto podejrzany format daty w rekordzie id=88

2026-03-01 08:00:22 INFO Przetworzono batch=3 rekordow=100

2026-03-01 08:00:23 INFO Zakonczono przetwarzanie zadania job_id=42

2026-03-01 08:00:24 ERROR Zakonczono z bledami: errors=4 warnings=5

EOF

6. Należy wykonać potok, który filtruje z input/app.log linie zawierające WARN, a następnie zapisuje je do output/warn_only.txt jednocześnie wypisując je na ekran. Wymagane jest użycie tee w miejscu, w którym strumień ma zostać rozdzielony na ekran i do pliku, a następnie wykonanie kontroli zawartości utworzonego pliku.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wymagane jest przygotowanie skryptu `bin/warn_tee.sh`, który realizuje filtrację i zapis, a następnie wykonuje kontrolne wyświetlenie kilku pierwszych linii pliku wynikowego (np. przez `head`).

7. Należy utworzyć skrypt `bin/only_errors.sh`, który przyjmuje dane ze standardowego wejścia i wypisuje na standardowe wyjście wyłącznie linie zawierające `ERROR`. Skrypt nie ma odczytywać pliku bezpośrednio po nazwie, tylko działać strumieniowo. Skrypt ma czytać dane wyłącznie ze standardowego wejścia (`STDIN`), np. w pętli `while read -r line; do ...; done`. Niedozwolone jest odwoływanie się do pliku logu przez nazwę ścieżki wewnątrz skryptu. Dopuszcza się implementację opartą o `grep "ERROR"` bez podawania nazwy pliku, tj. filtrując wyłącznie strumień z `STDIN`.

Następnie należy pokazać działanie skryptu jako elementu pipeline'u, np. w postaci: `odczyt logu | skrypt filtrujący | zliczenie linii`. Wymagane jest, aby skrypt działał poprawnie również wtedy, gdy źródłem danych jest inny potok, a nie plik.

Wymagane jest zaprezentowanie co najmniej dwóch uruchomień: jedno z pliku (np. `cat input/app.log | ...`) oraz jedno z generatora danych (np. `printf ... | ...`), aby potwierdzić strumieniowy charakter rozwiązania.

8. Należy utworzyć skrypt `bin/log_summary.sh`, który przyjmuje jako argument ścieżkę do pliku logu i wypisuje na standardowe wyjście krótkie podsumowanie obejmujące: liczbę wszystkich linii w pliku, liczbę linii `INFO`, `WARN` i `ERROR`. Skrypt ma wypisywać wyłącznie raport (bez dodatkowych komunikatów). W przypadku błędnego użycia (np. brak argumentu) lub braku pliku logu skrypt ma wypisać komunikat błędu na `STDERR` i zakończyć działanie kodem 2. Komunikaty diagnostyczne nie mogą pojawiać się na `STDOUT`, aby raport nadawał się do dalszego przetwarzania potokami.

Następnie należy uruchomić skrypt i przekierować jego standardowe wyjście do `output/summary.txt`. Wymagane jest, aby raport był możliwy do wykorzystania w dalszym przetwarzaniu (np. przez kolejne narzędzia w potoku).



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wymagane jest, aby raport był generowany przez skrypt i przekierowanie do pliku było wykonane na poziomie uruchomienia (tj. bez „ręcznego” dopisywania treści do pliku).

9. Należy utworzyć w katalogu input co najmniej trzy pliki logów, np. app1.log, app2.log, app3.log, w każdym po kilkanaście linii, z wystąpieniami ERROR. Następnie należy przygotować polecenie oparte o potoki, które zliczy łączną liczbę linii zawierających ERROR we wszystkich plikach. Wymagane jest, aby wynik był pojedynczą liczbą, a sposób realizacji wykorzystywał standardowe narzędzia powłoki i potokowanie. Wynik ma zawierać wyłącznie liczbę (bez prefiksów nazw plików i bez dodatkowych opisów); dopuszcza się użycie opcji `grep -h` lub takiej konstrukcji potoku, która usuwa nazwy plików z wyjścia.

Wymagane jest przygotowanie skryptu `bin/total_errors.sh`, który agreguje wynik dla wszystkich plików `.log` w katalogu input i wypisuje liczbę na `STDOUT` w postaci pojedynczej wartości.

10. Należy przygotować scenariusz, w którym wykonywany jest zestaw poleceń generujących raport z katalogu input: lista plików, liczba plików, a następnie łączna liczba linii zawierających ERROR we wszystkich plikach `.log`. Wynik ma zostać zapisany do `output/final_report.txt` w czytelnej formie, z nagłówkiem oraz datą wygenerowania raportu. Wymagane jest zastosowanie zarówno przekierowań (`>`, `>>`), jak i potoków (`|`) w ramach jednego spójnego rozwiązania.

Wymagane jest przygotowanie skryptu `bin/final_report.sh`, który po uruchomieniu generuje kompletny raport w `output/final_report.txt` i może być uruchamiany wielokrotnie bez ręcznych modyfikacji.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Pytania pomocnicze

1. Jakie role pełnią strumienie STDIN, STDOUT i STDERR w procesach systemu Unix/Linux i dlaczego ich rozdzielenie jest istotne w automatyzacji?
2. Czym różni się przekierowanie `>` od `>>` i w jakich sytuacjach dopisywanie jest bezpieczniejsze niż nadpisywanie?
3. Jak działa przekierowanie wejścia `<` i czym różni się od przekazania nazwy pliku jako argumentu polecenia (np. `wc -l plik` vs `wc -l < plik`)?
4. W jaki sposób zapisać standardowe wyjście i wyjście diagnostyczne do dwóch osobnych plików oraz jak zweryfikować, że komunikaty błędów nie trafiły do pliku przeznaczonego dla STDOUT?
5. Co oznacza zapis `2>&1` i jaki jest jego praktyczny cel podczas uruchamiania poleceń w skryptach?
6. Na czym polega potok `|` i w jaki sposób wyjście jednego polecenia staje się wejściem następnego etapu przetwarzania?
7. Dlaczego w podejściu pipeline'owym zaleca się budowanie łańcucha z narzędzi realizujących pojedyncze, wąskie zadanie (np. filtracja, selekcja, zliczanie)?
8. Do czego służy polecenie `tee` i czym różni się od zwykłego przekierowania `>` w kontekście pracy w potoku?
9. Jakie wymagania powinien spełniać skrypt zaprojektowany jako filtr strumieniowy, aby mógł być używany w pipeline (wejście ze STDIN, wynik na STDOUT)?
10. Dlaczego komunikaty diagnostyczne skryptu nie powinny trafiać na STDOUT, jeżeli wynik ma być dalej przetwarzany przez inne narzędzia w potoku?
11. W jaki sposób można wykonać analizę wielu plików logów i uzyskać wynik w postaci pojedynczej liczby, bez dopisywania nazw plików do wyniku?
12. Jakie elementy powinien zawierać raport generowany w skrypcie w sposób powtarzalny (nagłówki, data/godzina, sekcje), aby mógł być uruchamiany wielokrotnie bez ręcznych modyfikacji?

Podsumowanie

W trakcie czwartego laboratorium utrwalono przetwarzanie danych w powłoce Bash z wykorzystaniem plików oraz strumieni wejścia i wyjścia. Omówiono rolę standardowych strumieni procesu (STDIN, STDOUT, STDERR) oraz konsekwencje



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

ich poprawnego rozdzielania w kontekście automatyzacji i diagnostyki. Przećwiczono przekierowania umożliwiające zapis wyników do plików, dopisywanie kolejnych sekcji raportu oraz wczytywanie danych z plików przez standardowe wejście.

Wprowadzono budowę łańcuchów przetwarzania z użyciem potoków, w których kolejne narzędzia realizują wąskie etapy analizy (filtracja i zliczanie). Zastosowano polecenie tee do rozgałęziania strumienia w celu jednoczesnego zapisu do pliku i obserwacji wyników w terminalu. Utrwalono również zasady projektowania skryptów w stylu narzędzi pipeline'owych: przyjmowanie danych ze STDIN, emitowanie wyniku na STDOUT oraz kierowanie komunikatów diagnostycznych na STDERR.

Rezultatem laboratorium jest umiejętność przygotowania powtarzalnych skryptów generujących raporty oraz filtrów strumieniowych, które mogą być łączone w większe pipeline'y przetwarzające dane tekstowe i logi.

Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:

Programowanie skryptowe

Zajęcia laboratoryjne nr 5

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	3
Ćwiczenia samodzielne	9
Pytania pomocnicze	12
Podsumowanie	12
Literatura	13



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Laboratorium piąte ma na celu wprowadzenie struktur danych dostępnych w Bash, w szczególności tablic indeksowanych i asocjacyjnych, oraz praktycznych wzorców iteracji po kolekcjach. W ramach zajęć realizowane jest inicjalizowanie kolekcji, dodawanie i usuwanie elementów, odczyt rozmiaru oraz przetwarzanie list danych w sposób deterministyczny. Efektem ma być uporządkowane zarządzanie zestawami danych wejściowych (np. listami plików, identyfikatorów, parametrów) bez konieczności natychmiastowego sięgania po narzędzia zewnętrzne.

Cel ćwiczenia/laboratorium

- Zrozumienie roli kolekcji w Bash jako sposobu organizowania zestawów wartości wykorzystywanych wielokrotnie w skrypcie.
- Definiowanie i użycie tablic indeksowanych oraz operacje podstawowe: inicjalizacja, odczyt elementów, modyfikacja, usuwanie oraz wyznaczanie liczby elementów.
- Wprowadzenie tablic asocjacyjnych jako struktury mapującej klucz → wartość oraz stosowanie ich do przechowywania prostych konfiguracji i wyników pośrednich.
- Iteracja po kolekcjach w sposób deterministyczny oraz budowa scenariuszy przetwarzania list (np. list plików, list parametrów, zestawów nazw).
- Integracja kolekcji z konstrukcjami sterującymi (warunki, pętle) w celu tworzenia czytelnej i skalowalnej logiki skryptu.
- Utrwalenie zasad bezpiecznego cytowania przy pracy z elementami kolekcji, w szczególności gdy przechowywane wartości mogą zawierać spacje lub znaki specjalne.

Instrukcja laboratoryjna/ćwiczeniowa

Kolekcje w Bash

W skryptach Bash często zachodzi potrzeba przechowywania wielu wartości i wykonywania na nich powtarzalnych operacji. Typowe zastosowania obejmują listy



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

plików do przetworzenia, zestawy parametrów uruchomieniowych, identyfikatory zadań, nazwy usług oraz mapowanie kluczy na wartości konfiguracyjne. Kolekcje (tablice) umożliwiają utrzymanie takich danych w uporządkowanej postaci w obrębie skryptu oraz wykonywanie iteracji w sposób czytelny i powtarzalny, bez konieczności natychmiastowego angażowania narzędzi zewnętrznych.

Tablice indeksowane

Tablica indeksowana przechowuje elementy pod indeksami liczbowymi (0, 1, 2, ...). Inicjalizacja odbywa się przez nawiasy okrągłe, a dostęp do elementów oraz właściwości tablicy realizowany jest przez podstawienia z nawiasami klamrowymi.

Przykłady podstawowe zaprezentowano na rysunku numer 1:

```
files=("input/app.log" "input/app1.log" "input/app2.log")
```

```
echo "Pierwszy plik: ${files[0]}"
```

```
files[1]="input/changed.log"
```

```
unset 'files[2]'
```

```
echo "Liczba elementow: ${#files[@]}"
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
student@student-VirtualBox:~/Desktop/lab5$ files=("input/app.log" "input/app1.log" "input/app2.log")
student@student-VirtualBox:~/Desktop/lab5$ echo "Pierwszy plik: ${files[0]}"
Pierwszy plik: input/app.log
student@student-VirtualBox:~/Desktop/lab5$ echo "Liczba elementow: ${#files[@]}"
Liczba elementow: 3
student@student-VirtualBox:~/Desktop/lab5$ echo "Drugi plik: ${files[1]}"
Drugi plik: input/app1.log
student@student-VirtualBox:~/Desktop/lab5$ echo "Trzeci plik: ${files[2]}"
Trzeci plik: input/app2.log
student@student-VirtualBox:~/Desktop/lab5$ files[1]="input/changed.log"
student@student-VirtualBox:~/Desktop/lab5$ echo "Drugi plik: ${files[1]}"
Drugi plik: input/changed.log
student@student-VirtualBox:~/Desktop/lab5$ unset 'files[2]'
student@student-VirtualBox:~/Desktop/lab5$ echo "Liczba elementow: ${#files[@]}"
Liczba elementow: 2
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 1. Przykład tablicy indeksowanej

Najczęściej używane operacje:

- `${arr[i]}` – odczyt elementu o indeksie `i`,
- `${#arr[@]}` – liczba elementów tablicy,
- `arr+=("nowy")` – dopisanie elementu na koniec tablicy,
- `unset 'arr[i]'` – usunięcie elementu; po usunięciu może wystąpić nieciągłość indeksów (nie wszystkie wartości `0..n` muszą być obsadzone).

W praktyce należy odróżniać:

- iterację po wartościach (typowe przetwarzanie danych),
- iterację po indeksach (np. gdy potrzebne są numery elementów lub gdy tablica ma nieciągłe indeksy po `unset`).

W praktyce preferuje się `${arr[@]}` (szczególnie w cudzysłowie `"${arr[@]}"`), ponieważ zachowuje granice elementów, podczas gdy `${arr[*]}` w cudzysłowie łączy elementy w jeden łańcuch.

Skrypt prezentujący iteracje po wartościach:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
cat > iter_values.sh <<'EOF'  
#!/bin/bash  
arr=("alpha" "beta value" "gamma" "delta item")  
for x in "${arr[@]}": do  
    echo "$x"  
done  
EOF
```

Należy nadać mu prawa do uruchomienia, które zostało zaprezentowane na rysunku 2:

```
chmod +x iter_values.sh  
./iter_values.sh
```



```
student@student-VirtualBox: ~/Desktop/lab5$ bash iter_values.sh  
alpha  
beta value  
gamma  
delta item  
student@student-VirtualBox: ~/Desktop/lab5$
```

Rys. 2. Przykład iterowania po wartościach.

Skrypt prezentujący iterację po indeksach (zachowuje informację o indeksie i działa także przy nieciągłych indeksach):

```
cat > iter_indexes.sh <<'EOF'  
#!/bin/bash  
arr=("alpha" "beta" "gamma value" "delta" "epsilon item")  
unset 'arr[1]'
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
unset 'arr[3]'
```

```
for i in "${!arr[@]}"; do
```

```
  printf "%s -> %s\n" "$i" "${arr[$i]}"
```

```
done
```

```
EOF
```

Należy nadać mu prawa do uruchomienia, które zostało zaprezentowane na rysunku 3, widać tam że elementy na indeksach 1 oraz 3 zostały usunięte:

```
chmod +x iter_indexes.sh
```

```
./iter_indexes.sh
```

```
student@student-VirtualBox:~/Desktop/lab5$ ./iter_indexes.sh
0 -> alpha
2 -> gamma value
4 -> epsilon item
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 3. Przykład iterowania po indeksach

Jeżeli numer indeksu znajduje się w zmiennej, na przykład *i*, to odwołanie do elementu ma postać `${arr[$i]}`. Gdy wartość ma zostać przekazana dalej jako argument, należy ją cytować, np. `"${arr[$i]}"`.

Wczytywanie wierszy pliku do tablicy indeksowanej

W praktyce często zachodzi potrzeba wczytania danych z pliku w modelu „jeden wiersz = jeden element tablicy”. Poniższe dwa warianty są równoważne funkcjonalnie i zachowują wiersze zawierające spacje jako pojedyncze elementy tablicy.

Wariant A wykorzystujący `readarray` / `mapfile`, które mogą być niedostępne w bardzo starych wersjach `bash`. W `mapfile` opcja `-t` usuwa znak końca linii (`\n`) z każdego elementu tablicy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dane wejściowe:

```
cat > input/items.txt <<'EOF'
```

```
alpha
```

```
beta value
```

```
gamma
```

```
delta item
```

```
epsilon
```

```
zeta long name
```

```
eta
```

```
theta value
```

```
EOF
```

Skrypt służący do wczytania elementów do tablicy, wymaga istnienia katalogu input w katalogu bieżącym:

```
cat > readarray_demo.sh <<'EOF'
```

```
#!/bin/bash
```

```
items=()
```

```
readarray -t items < input/items.txt
```

```
printf "Liczba elementów: %s\n" "${#items[@]}"
```

```
printf "Pierwszy: %s\n" "${items[0]}"
```

```
printf "Drugi: %s\n" "${items[1]}"
```

```
printf "Ostatni: %s\n" "${items[${#items[@]}-1]}"
```

```
EOF
```

Należy nadać mu uprawnienia do uruchomienia, które zostało zaprezentowane na rysunku numer 4.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

`chmod +x readarray_demo.sh`

`./readarray_demo.sh`

```
student@student-VirtualBox:~/Desktop/lab5$ cat input/items.txt
alpha
beta value
gamma
delta item
epsilon
zeta long name
eta
theta value
student@student-VirtualBox:~/Desktop/lab5$ ./readarray_demo.sh
Liczba elementów: 8
Pierwszy: alpha
Drugi: beta value
Ostatni: theta value
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 4. Wywołanie skryptu wczytującego dane do tablicy

Wariant B wykorzystujący pętlę `while read -r` został zaprezentowany na rysunku 5. Opcja `-r` powoduje, że znak `\` (backslash) nie jest interpretowany jako znak ucieczki podczas wczytywania tekstu.

```
cat > while_read_demo.sh <<'EOF'
#!/bin/bash
items=()
while read -r line; do
    items+=("$line")
done < input/items.txt

printf "Liczba elementów: %s\n" "${#items[@]}"
printf "Pierwszy: %s\n" "${items[0]}"
printf "Drugi: %s\n" "${items[1]}"
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

```
printf "Ostatni: %s\n" "${items[${#items[@]}-1]}"  
EOF
```

```
student@student-VirtualBox:~/Desktop/lab5$ cat while_read_demo.sh  
#!/bin/bash  
items=()  
while read -r line; do  
    items+=("$line")  
done < input/items.txt  
  
printf "Liczba elementów: %s\n" "${#items[@]}"  
printf "Pierwszy: %s\n" "${items[0]}"  
printf "Drugi: %s\n" "${items[1]}"  
printf "Ostatni: %s\n" "${items[${#items[@]}-1]}"  
student@student-VirtualBox:~/Desktop/lab5$ ./while_read_demo.sh  
Liczba elementów: 8  
Pierwszy: alpha  
Drugi: beta value  
Ostatni: theta value  
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 5. Skrypt wczytujący dane do tablicy za pomocą pętli while

Bezpieczne cytowanie w tablicach indeksowanych

Elementy tablicy mogą zawierać spacje i znaki specjalne. Z tego powodu w praktyce stosuje się zasady:

- iteracja po elementach: for x in "\${arr[@]}" (z cudzysłowem),
- przekazywanie elementu jako argumentu: some_cmd "\${arr[i]}",
- unikanie zapisu for x in \${arr[@]} bez cudzysłowów, ponieważ prowadzi do niezamierzonego dzielenia argumentów (word splitting).



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Przykład skryptu realizującego bezpieczne cytowanie został zaprezentowany poniżej oraz na rysunku numer 6.

```
cat > safe_quoting_demo.sh <<'EOF'
#!/bin/bash
arr=("alpha" "beta value" "gamma item" "delta")
echo 'Bez cudzysłówów:'
for x in ${arr[@]}; do
    printf "<%=s>\n" "$x"
done

echo
echo 'Z cudzysłowami:'
for x in "${arr[@]}"; do
    printf "<%=s>\n" "$x"
done
EOF
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

```
student@student-VirtualBox:~/Desktop/lab5$ cat safe_quoting_demo.sh
#!/bin/bash
arr=("alpha" "beta value" "gamma item" "delta")
echo 'Bez cudzysłowów:'
for x in ${arr[@]}; do
    printf "<%s>\n" "$x"
done

echo
echo 'Z cudzysłowami:'
for x in "${arr[@]}"; do
    printf "<%s>\n" "$x"
done
student@student-VirtualBox:~/Desktop/lab5$ bash safe_quoting_demo.sh
Bez cudzysłowów:
<alpha>
<beta>
<value>
<gamma>
<item>
<delta>

Z cudzysłowami:
<alpha>
<beta value>
<gamma item>
<delta>
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 6. Skrypt realizujący bezpieczne cytowanie.

Tablice asocjacyjne

Tablice asocjacyjne są obsługiwane w Bash, dlatego skrypty wykorzystujące `declare -A` powinny być uruchamiane w Bash (np. z shebangiem `#!/bin/bash`). Tablica asocjacyjna mapuje klucz tekstowy na wartość i wymaga deklaracji, na przykład:

`declare -A map`

Jest użyteczna m.in. do:

- przechowywania prostych konfiguracji (klucz → wartość),
- liczników i agregacji wyników (np. INFO/WARN/ERROR → liczba),



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- struktur typu „seen” do eliminacji duplikatów.

Przykład tablicy asocjacyjnej został przedstawiony na rysunku numer 7:

```
cat > assoc_array_demo.sh <<'EOF'  
#!/bin/bash  
declare -A counters  
counters[INFO]=0  
counters[WARN]=0  
counters[ERROR]=0  
echo "Stan początkowy:"  
echo "INFO=${counters[INFO]}"  
echo "WARN=${counters[WARN]}"  
echo "ERROR=${counters[ERROR]}"  
counters[ERROR]=$((counters[ERROR] + 1))  
counters[ERROR]=$((counters[ERROR] + 1))  
echo  
echo "Po zwiększeniu licznika ERROR:"  
echo "ERROR=${counters[ERROR]}"  
unset 'counters[WARN]'  
echo  
echo "Po unset WARN:"  
echo "INFO=${counters[INFO]}"  
echo "WARN=${counters[WARN]}"  
echo "ERROR=${counters[ERROR]}"  
EOF
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
student@student-VirtualBox:~/Desktop/lab5$ bash assoc_array_demo.sh
Stan początkowy:
INFO=0
WARN=0
ERROR=0

Po zwiększeniu licznika ERROR:
ERROR=2

Po unset WARN:
INFO=0
WARN=
ERROR=2
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 7. Przykład tablicy asocjacyjnej.

Najczęściej używane konstrukcje:

- odczyt wartości: "\${map[\$k]}",
- iteracja po kluczach: for k in "\${!map[@]}"; do ...; done,
- poprawne cytowanie kluczy i wartości, jeżeli mogą zawierać znaki specjalne.

Porządek iteracji i raportowanie deterministyczne

Tablica indeksowana zachowuje porządek wynikający z indeksów (z uwzględnieniem faktu, że indeksy mogą być nieciągłe po usunięciu elementów). Tablica asocjacyjna nie gwarantuje stabilnego porządku kluczy, dlatego raportowanie wyników w deterministycznej kolejności powinno opierać się na z góry ustalonej liście kluczy.

Przykład raportowania w stałej kolejności został zaprezentowany na rysunku numer 8:

```
cat > deterministic_report.sh <<'EOF'

#!/bin/bash
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
declare -A counters
counters[ERROR]=2
counters[INFO]=5
counters[WARN]=1
levels=("INFO" "WARN" "ERROR")
for lvl in "${levels[@]}"; do
    echo "$lvl: ${counters[$lvl]}"
done
EOF
```

```
student@student-VirtualBox:~/Desktop/lab5$ cat deterministic_report.sh
#!/bin/bash
declare -A counters
counters[ERROR]=2
counters[INFO]=5
counters[WARN]=1

levels=("INFO" "WARN" "ERROR")

for lvl in "${levels[@]}"; do
    echo "$lvl: ${counters[$lvl]}"
done
student@student-VirtualBox:~/Desktop/lab5$ ./deterministic_report.sh
INFO: 5
WARN: 1
ERROR: 2
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 8. Przykład raportowania w stałej kolejności.

W zadaniach laboratoryjnych nie należy zakładać, że iteracja po "\${!map[@]}" zwróci klucze w pożądanej kolejności raportu. Jeżeli wynik ma mieć stałą strukturę, kolejność należy wymusić osobną listą kluczy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



To podejście jest zalecane wszędzie tam, gdzie raport ma mieć stałą strukturę niezależną od sposobu przechowywania kluczy przez Bash.

Ćwiczenie praktyczne

W tej części zostanie przygotowany kompletny przykład skryptu Bash, który:

1. wczyta wiersze z pliku do tablicy indeksowanej,
2. wypisze podstawowe informacje o tablicy,
3. zbuduje raport numerowany,
4. sklasyfikuje elementy do kategorii SHORT, MEDIUM, LONG,
5. wypisze raport końcowy w stałej, deterministycznej kolejności.

Ćwiczenie należy realizować krok po kroku, rozwijając jeden skrypt: `bin/load_items.sh`.

Krok 1. Przygotowanie katalogu roboczego i danych wejściowych

Najpierw należy utworzyć katalog roboczy `lab5/`, podkatalog `input/` oraz podkatalog `bin/`:

```
mkdir -p lab5/input lab5/bin  
cd lab5
```

Następnie należy przygotować plik `input/items.txt`. Warto umieścić w nim również wiersze zawierające spacje, aby można było sprawdzić poprawność wczytywania danych do tablicy.

```
cat > input/items.txt <<'EOF'  
  
alpha  
  
beta value  
  
gamma  
  
delta item  
  
epsilon
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



zeta long name

eta

theta value

iota

kappa item

EOF

Zawartość pliku można sprawdzić poleceniem co zostało zaprezentowane na rysunku numer 9:

cat input/items.txt

```
student@student-VirtualBox:~/Desktop/lab5$ cat input/items.txt
alpha
beta value
gamma
delta item
epsilon
zeta long name
eta
theta value
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 8. Dane wejściowe dla przykładu

Krok 2. Utworzenie skryptu bin/load_items.sh

Należy utworzyć plik skryptu:

nano bin/load_items.sh

Do pliku należy wkleić poniższy kod:

#!/bin/bash

set -u

INPUT_FILE="input/items.txt"



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

```
if [[ ! -f "$INPUT_FILE" ]]; then  
echo "Błąd: brak pliku $INPUT_FILE" >&2  
exit 1  
fi  
  
items=()  
  
if builtin command -v readarray >/dev/null 2>&1; then  
readarray -t items < "$INPUT_FILE"  
  
else  
while read -r line; do  
items+=("$line")  
done < "$INPUT_FILE"  
fi  
  
count=${#items[@]}  
  
if (( count == 0 )); then  
echo "Plik wejściowy jest pusty." >&2  
exit 1  
fi  
  
echo "=== Podstawowe informacje o tablicy ==="  
echo "Liczba elementów: $count"  
echo "Pierwszy element: ${items[0]}"  
  
if (( count >= 3 )); then  
echo "Trzeci element: ${items[2]}"  
  
else  
echo "Trzeci element: brak"
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



fi

last_index=\$((count - 1))

echo "Ostatni element: \${items[\$last_index]}"

echo

echo "=== Raport numerowany ==="

n=1

for item in "\${items[@]}": do

echo "\$n. \$item"

((n++))

done

echo

echo "=== Klasyfikacja długości napisów ==="

declare -A counters

counters[SHORT]=0

counters[MEDIUM]=0

counters[LONG]=0

for item in "\${items[@]}": do

len=\${#item}

if ((len >= 1 && len <= 4)); then

((counters[SHORT]++))

elif ((len >= 5 && len <= 7)); then

((counters[MEDIUM]++))

else

((counters[LONG]++))

fi

done



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



echo

echo "=== Raport końcowy ==="

categories=("SHORT" "MEDIUM" "LONG")

for category in "\${categories[@]}"; do

echo "\$category: \${counters[\$category]}"

done

Po zapisaniu pliku należy nadać mu uprawnienia wykonywania:

```
chmod +x bin/load_items.sh
```

Krok 3. Uruchomienie skryptu

Skrypt należy uruchomić poleceniem:

```
./bin/load_items.sh
```

Przykładowy wynik działania został przedstawiony na rysunku 9.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
student@student-VirtualBox:~/Desktop/lab5$ nano bin/load_items.sh
student@student-VirtualBox:~/Desktop/lab5$ bash bin/load_items.sh
=== Podstawowe informacje o tablicy ===
Liczba elementów: 8
Pierwszy element: alpha
Trzeci element: gamma
Ostatni element: theta value

=== Raport numerowany ===
1. alpha
2. beta value
3. gamma
4. delta item
5. epsilon
6. zeta long name
7. eta
8. theta value

=== Klasyfikacja długości napisów ===

=== Raport końcowy ===
SHORT: 1
MEDIUM: 3
LONG: 4
student@student-VirtualBox:~/Desktop/lab5$
```

Rys. 9. Uruchomienie skryptu i wynik działania

Krok 4. Omówienie działania skryptu

Skrypt rozpoczyna pracę od sprawdzenia, czy istnieje plik input/items.txt. Jeżeli plik nie istnieje, wypisywany jest komunikat błędu na standardowe wyjście błędów, a działanie programu zostaje zakończone.

Następnie tworzoną strukturą danych jest tablica indeksowana items. Jeżeli środowisko obsługuje polecenie readarray, plik zostaje wczytany bezpośrednio do tablicy. W przeciwnym razie używana jest pętla while read -r, która dodaje kolejne wiersze do tablicy. W obu przypadkach zachowany jest model: jeden wiersz pliku = jeden element tablicy.

Po wczytaniu danych skrypt wypisuje:

1. liczbę elementów tablicy,
2. pierwszy element,
3. trzeci element,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



4. ostatni element.

W dalszej części wykonywana jest iteracja po "\${items[@]}", co pozwala bezpiecznie przetwarzać również te elementy, które zawierają spacje. Na tej podstawie budowany jest raport numerowany.

Kolejnym etapem jest utworzenie tablicy asocjacyjnej counters, która przechowuje liczniki trzech kategorii:

SHORT dla napisów o długości od 1 do 4 znaków,

MEDIUM dla napisów o długości od 5 do 7 znaków,

LONG dla napisów o długości co najmniej 8 znaków.

Każdy element tablicy items jest analizowany pod względem długości, a następnie zwiększany jest odpowiedni licznik.

Na końcu raport jest wypisywany w stałej kolejności SHORT, MEDIUM, LONG. Zastosowanie osobnej tablicy categories gwarantuje deterministyczny porządek wyniku niezależnie od sposobu przechowywania kluczy przez Bash.

Krok 5. Wniosek z ćwiczenia

Wykonany przykład pokazuje, że Bash umożliwia nie tylko prostą automatyzację pojedynczych poleceń, ale również organizowanie danych w postaci tablic oraz ich dalsze przetwarzanie. Tablice indeksowane pozwalają przechowywać uporządkowane listy wartości, natomiast tablice asocjacyjne umożliwiają budowanie prostych map oraz liczników. W połączeniu z poprawnym cytowaniem i kontrolowanym raportowaniem daje to podstawę do pisania bardziej czytelnych i skalowalnych skryptów.

Ćwiczenia samodzielne

Dopuszcza się wstępną realizację zadań w terminalu, natomiast wymagany rezultatem jest skrypt Bash automatyzujący rozwiązanie. W przypadku błędnego użycia (zła liczba argumentów / zły typ danych) skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

Ćwiczenie numer 0

Utworzenie katalogu roboczego



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Utworzenie środowiska:

`mkdir -p lab5/input`

`mkdir -p lab5/bin`

`cd lab5`

Utworzenie pliku input/items.txt

`cat > input/items.txt <<'EOF'`

`alpha`

`beta value`

`gamma`

`delta item`

`epsilon`

`zeta long name`

`eta`

`theta value`

`iota`

`kappa item`

`EOF`

Utworzenie pliku input/words.txt

`cat > input/words.txt <<'EOF'`

`error warn info`

`error error warn`

`info test`

`ERROR warn`

`error_info info`

`warn info error`



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



EOF

Utworzenie pliku input/app.log

cat > input/app.log <<'EOF'

2026-03-01 08:00:00 INFO user=anna msg=Start aplikacji

2026-03-01 08:00:01 WARN user=anna msg=Brak pliku konfiguracyjnego

2026-03-01 08:00:02 ERROR user=jan msg=Nie udało się otworzyć pliku

2026-03-01 08:00:03 INFO user=jan msg=Ponowiono próbę

2026-03-01 08:00:04 error user=ola msg=Małe litery

2026-03-01 08:00:05 WARN user=jan msg=Timeout

2026-03-01 08:00:06 ERROR user=anna msg=HTTP=500

2026-03-01 08:00:07 INFO user=jan msg=Koniec pracy

EOF

Utworzenie katalogu input/files_demo

mkdir -p input/files_demo

touch input/files_demo/app1.log

touch input/files_demo/app2.log

touch input/files_demo/notes.txt

touch input/files_demo/data.csv

touch input/files_demo/.hidden_file

mkdir -p input/files_demo/nested

touch input/files_demo/nested/inner.log

mkdir -p input/empty_demo



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Ćwiczenie numer 1

Należy utworzyć skrypt `bin/array_basics.sh`, który inicjalizuje tablicę indeksowaną co najmniej pięcioma wartościami, wypisuje liczbę elementów, wypisuje pierwszy i ostatni element, następnie usuwa jeden element poleceniem `unset` oraz ponownie wypisuje liczbę elementów. Wymagane jest pokazanie, że usunięcie elementu może spowodować nieciągłość indeksów tablicy.

Ćwiczenie numer 2

Należy utworzyć skrypt `bin/args_to_array.sh`, który przyjmuje dowolną liczbę argumentów i zapisuje je do tablicy indeksowanej. Skrypt ma wypisać każdy argument w osobnej linii z numeracją. Wymagane jest poprawne użycie `"$@"` oraz iteracji po `"${arr[@]}"`, tak aby argumenty zawierające spacje były traktowane jako pojedyncze elementy.

Ćwiczenie numer 3

Należy utworzyć skrypt `bin/files_inventory.sh`, który przyjmuje jako argument ścieżkę do katalogu.

Skrypt ma zbudować tablicę indeksowaną z nazwami wszystkich nieukrytych plików regularnych znajdujących się bezpośrednio w tym katalogu (bez przeszukiwania podkatalogów).

Następnie ma wypisać:

liczbę znalezionych plików,

listę nazw plików, po jednej nazwie w każdej linii.

W przypadku braku argumentu lub wskazania ścieżki, która nie jest katalogiem, skrypt ma wypisać błąd na `STDERR` i zakończyć działanie kodem 2.

Ćwiczenie numer 4

Należy utworzyć skrypt `bin/assoc_config.sh`, który tworzy tablicę asocjacyjną mapującą co najmniej cztery klucze na wartości, na przykład: `HOST`, `PORT`, `MODE`, `TIMEOUT`.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Skrypt ma działać w dwóch trybach:

przy uruchomieniu bez argumentów ma wypisać wszystkie pary klucz → wartość w ustalonej kolejności: HOST, PORT, MODE, TIMEOUT,

przy uruchomieniu z jednym argumentem ma wypisać wyłącznie wartość odpowiadającą wskazanemu kluczowi.

W przypadku nieznanego klucza albo błędnej liczby argumentów skrypt ma wypisać komunikat błędu na STDERR i zakończyć działanie kodem 2.

Ćwiczenie numer 5

Należy utworzyć skrypt bin/count_words.sh, który przyjmuje jako argument ścieżkę do pliku tekstowego i zlicza wystąpienia wyłącznie trzech słów kontrolnych: error, warn, info.

Liczniki mają być przechowywane w tablicy asocjacyjnej, gdzie klucz oznacza słowo, a wartość liczbę jego wystąpień.

Za „słowo” uznaje się token rozdzielony białymi znakami; nie jest wymagane usuwanie interpunkcji ani normalizacja wielkości liter.

Skrypt ma wypisać raport na STDOUT dokładnie w kolejności:

error: <licznik>

warn: <licznik>

info: <licznik>

W przypadku błędnego użycia lub braku pliku skrypt ma wypisać komunikat na STDERR i zakończyć działanie kodem 2.

Ćwiczenie numer 7

Należy utworzyć skrypt bin/dedupe_list.sh, który przyjmuje listę argumentów i usuwa duplikaty, wykorzystując tablicę asocjacyjną jako strukturę „seen”. Skrypt ma wypisać na STDOUT listę unikalnych elementów w kolejności pierwszego wystąpienia.

Skrypt wymaga co najmniej jednego argumentu. W przypadku uruchomienia bez argumentów ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie dodatkowe A

Należy utworzyć skrypt `bin/array_indexes.sh`, który tworzy tablicę indeksowaną zawierającą co najmniej sześć elementów, a następnie usuwa dwa wybrane elementy poleceniem `unset` w taki sposób, aby indeksy tablicy nie tworzyły ciągłej sekwencji. Następnie skrypt ma wypisać pary indeks $\rightarrow$ wartość, iterując po dostępnych indeksach tablicy z użyciem konstrukcji `${!arr[@]}` oraz odwołując się do wartości przez `${arr[$i]}` z poprawnym cytowaniem. Wymagane jest wykazanie w wyniku działania skryptu, że po usunięciu elementów indeksy mogą być nieciągłe oraz że iteracja po indeksach nie zakłada kolejnych wartości 0, 1, 2, ... bez przerw.

Zadanie dodatkowe B

Należy utworzyć skrypt `bin/read_lines.sh`, który przyjmuje jako argument ścieżkę do pliku tekstowego i wczytuje jego wiersze do tablicy indeksowanej w modelu „jeden wiersz = jeden element tablicy”. Następnie skrypt ma wypisać liczbę wczytanych wierszy oraz trzy pierwsze elementy tablicy. Wymagane jest, aby wiersze zawierające spacje były przechowywane jako pojedyncze elementy tablicy (bez niezamierzonego dzielenia na argumenty). Dopuszcza się implementację opartą o `readarray/mapfile` lub o pętlę `while read -r`, pod warunkiem zachowania poprawnego cytowania.

Pytania pomocnicze

1. Jaką rolę pełnią kolekcje (tablice) w Bash i w jakich scenariuszach są szczególnie użyteczne?
2. Jak inicjalizuje się tablicę indeksowaną i jak odczytuje się jej element o zadany indeksie?
3. Jak uzyskać liczbę elementów tablicy indeksowanej i dlaczego `${#arr[@]}` różni się od `${#arr[0]}`?
4. Co robi `unset 'arr[i]'` w tablicy indeksowanej i jaki efekt może mieć to na indeksy elementów?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



5. Jak zadeklarować tablicę asocjacyjną i jak wygląda zapis przypisania wartości do klucza?
6. Jak iterować po wszystkich elementach tablicy indeksowanej z zachowaniem poprawnego cytowania?
7. Jaka jest różnica między iteracją po `"${arr[@]}"` i `${arr[@]}` w przypadku wartości zawierających spacje?
8. Czy tablica asocjacyjna gwarantuje porządek kluczy? Jak zapewnić deterministyczną kolejność raportowania wyników?
9. W jakich przypadkach funkcja lub skrypt powinny emitować dane wynikowe na STDOUT, a komunikaty błędów na STDERR?
10. W jaki sposób tablica asocjacyjna może być użyta jako licznik lub struktura do eliminacji duplikatów?

Podsumowanie

W trakcie piątego laboratorium wprowadzono struktury danych dostępne w Bash, ze szczególnym uwzględnieniem tablic indeksowanych i asocjacyjnych, jako narzędzi porządkujących pracę na zestawach wartości w skryptach. Omówiono podstawowe operacje na tablicach indeksowanych, obejmujące inicjalizację, odczyt, modyfikację, usuwanie elementów oraz wyznaczanie liczby elementów. Przedstawiono tablice asocjacyjne jako mechanizm mapowania klucz→wartość, użyteczny m.in. do przechowywania konfiguracji, liczników oraz wyników pośrednich. Utrwalono iterację po kolekcjach z zachowaniem poprawnego cytowania, w szczególności przy pracy z wartościami zawierającymi spacje i znaki specjalne. Zwrócono uwagę na kwestię deterministycznego porządku raportowania wyników, zwłaszcza w przypadku tablic asocjacyjnych, oraz na projektowanie skryptu w sposób umożliwiający czytelne przetwarzanie list danych. Rezultatem laboratorium jest umiejętność budowy skryptów, które przechowują i przetwarzają zbiory wartości w sposób przewidywalny, bezpieczny i łatwy do rozwoju.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Materiały do przedmiotu:

Programowanie skryptowe

Zajęcia laboratoryjne nr 6

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	3
Ćwiczenia samodzielne	11
Pytania pomocnicze	13
Podsumowanie	14
Literatura	14



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Laboratorium szóste ma na celu opanowanie dopasowań tekstowych w skryptach z rozróżnieniem szablonów powłoki (globbing) i wyrażeń regularnych. W ramach zajęć realizowane jest wykorzystanie grep, a także wprowadzenie przekształceń tekstu z użyciem narzędzi takich jak sed lub awk w zakresie niezbędnym do filtrowania, walidacji i ekstrakcji informacji. Efektem ma być umiejętność budowania reguł dopasowania dla typowych danych tekstowych (logi, pliki konfiguracyjne, komunikaty) oraz świadomy dobór narzędzi w zależności od realizowanego zadania.

Cel ćwiczenia/laboratorium

- Rozróżnienie dopasowań realizowanych przez szablony powłoki (globbing) od dopasowań opartych o wyrażenia regularne oraz świadomy dobór techniki do problemu.
- Wykorzystanie dopasowań do filtrowania, wyszukiwania i selekcji danych tekstowych w typowych plikach (logi, konfiguracje, dane w formie tekstowej).
- Budowa i testowanie podstawowych wyrażeń regularnych: klasy znaków, kotwice, powtórzenia, grupowanie oraz alternatywy w zakresie wymaganym w zadaniach laboratoryjnych.
- Stosowanie wyrażeń regularnych do walidacji formatów danych (np. identyfikatory, proste wzorce, struktury wierszy), a także do ekstrakcji informacji z większego tekstu.
- Realizacja prostych transformacji tekstu w oparciu o dopasowania (np. zamiany, usuwanie fragmentów, wyciąganie pól), z naciskiem na przewidywalność i jednoznaczność reguł.
- Ocena jakości dopasowań pod kątem przypadków brzegowych oraz unikanie typowych błędów, takich jak zbyt ogólne wzorce lub niezamierzone dopasowania.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Instrukcja laboratoryjna/ćwiczeniowa

Zakres i cel zagadnień

Dopasowania tekstowe stanowią jeden z kluczowych mechanizmów pracy w skryptach powłoki, ponieważ umożliwiają selekcję danych, walidację formatu oraz ekstrakcję istotnych informacji z większych zbiorów tekstu. W praktyce przekłada się to na możliwość filtrowania logów, wyszukiwania ustawień w plikach konfiguracyjnych, identyfikowania wierszy spełniających określone reguły oraz budowy prostych raportów opartych o dane tekstowe.

Globbering: szablony powłoki

Globbering to mechanizm dopasowania nazw plików i ścieżek realizowany przez powłokę jeszcze przed uruchomieniem programu, czyli na etapie interpretacji polecenia. Najczęściej stosowane symbole to * (dowolny ciąg znaków), ? (dokładnie jeden znak) oraz [...] (jeden znak z podanego zbioru), np. *.log wybiera wszystkie pliki z rozszerzeniem .log, a app?.log dopasowuje pliki typu app1.log, app2.log itd. Istotne jest rozróżnienie, że globbing nie jest wyrażeniem regularnym: dotyczy nazw plików, a nie zawartości tekstu. W konsekwencji zapis "*.log" (w cudzysłowie) nie zostanie rozwinięty przez powłokę do listy plików i zostanie przekazany do programu dosłownie jako tekst *.log, co jest częstym źródłem błędów podczas pracy z dopasowaniami.

Warto również pamiętać, że jeżeli wzorzec globbingu nie dopasuje żadnego pliku, Bash domyślnie może pozostawić wzorzec w postaci dosłownej (np. *.log). Z tego powodu przy projektowaniu skryptów nie należy zakładać, że każde dopasowanie zwróci co najmniej jeden wynik. W zadaniach wymagających odporności na brak dopasowań należy jawnie obsłużyć taki przypadek, na przykład przez sprawdzenie wyników dopasowania albo odpowiednią konfigurację opcji powłoki.

Wyrażenia regularne: idea i elementy budulcowe



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wyrażenia regularne (regex) stanowią formalny opis wzorca tekstowego, wykorzystywany przez narzędzia przetwarzające zawartość (np. grep, sed, awk) do wyszukiwania, walidacji i ekstrakcji danych. W ramach laboratorium wystarczające jest opanowanie podstawowych elementów: klas znaków (np. [0-9], [A-Za-z], [[:digit:]], [[:space:]]), kotwic początku i końca wiersza (^, \$) służących do dopasowania całej linii, powtórzeń (*, +, ?, {m,n}) kontrolujących liczbę wystąpień fragmentu wzorca, a także grupowania nawiasami (...) i alternatywy | w zakresie niezbędnym do budowy prostych reguł dopasowania. W praktyce stosuje się również wykluczanie dopasowań, np. przez filtrowanie odwrotne grep -v, aby usunąć wiersze spełniające określony wzorzec. Jakość wzorca zależy od jego precyzji: wzorce zbyt ogólne prowadzą do dopasowań niezamierzonych, natomiast wzorce celowe (z kotwicami i jednoznacznymi klasami znaków) ograniczają liczbę fałszywych trafień i uodparniają rozwiązanie na przypadki brzegowe.

grep jako podstawowe narzędzie filtrowania

Program grep służy do selekcji wierszy tekstu spełniających warunek dopasowania i stanowi podstawowe narzędzie budowy filtrów w potokach. Zachowanie grep jest sterowane opcjami programu, np. -n (wypisanie numerów linii), -i (ignorowanie wielkości liter), -v (odwrócenie dopasowania, czyli wybór wierszy niepasujących) oraz -c (zliczenie liczby dopasowanych wierszy), przy czym do rozwiązywania zadań nie jest wymagane wyczerpujące omówienie wszystkich opcji. Domyślnie grep interpretuje wzorce jako podstawowe wyrażenia regularne, natomiast tryb rozszerzony (grep -E) jest stosowany wtedy, gdy potrzebne są wygodniejsze konstrukcje, w szczególności alternatywy | oraz operatory + i ? bez konieczności stosowania dodatkowego escapingu. grep może pracować zarówno na pliku przekazanym jako argument, jak i strumieniowo (STDIN → STDOUT) jako etap pipeline'u, co pozwala łączyć go z innymi narzędziami przetwarzania tekstu w jednym, powtarzalnym łańcuchu operacji.

Kody wyjścia programu grep

Program grep zwraca kod zakończenia zależny od wyniku działania:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- 0 - znaleziono co najmniej jedno dopasowanie,
- 1 - nie znaleziono dopasowań (nie musi to oznaczać błędu wykonania),
- 2 - wystąpił błąd użycia albo problem z plikiem wejściowym.

W praktyce należy pamiętać, że kod 1 w grep często oznacza poprawne zakończenie analizy bez trafień, a nie awarię programu.

Regex podstawowy a rozszerzony

W praktyce należy odróżniać podstawowe wyrażenia regularne od rozszerzonych. Domyślny tryb grep wykorzystuje regex podstawowy (BRE), natomiast grep -E wykorzystuje regex rozszerzony (ERE). W laboratorium szóstym wszystkie przykłady korzystające z operatorów takich jak +, ?, |, {m,n} oraz z grupowania nawiasami (...) należy rozumieć jako wzorce przeznaczone do użycia w trybie rozszerzonym, tj. z grep -E, albo w innych narzędziach obsługujących analogiczne konstrukcje. Dzięki temu zapis wzorców pozostaje krótszy i czytelniejszy.

Walidacja formatu danych za pomocą regex

Wyrażenia regularne są często wykorzystywane do walidacji, czyli weryfikacji, czy dany ciąg znaków lub cała linia tekstu spełnia wymagany format. W odróżnieniu od filtrowania „gdzieś w linii”, walidacja opiera się na dopasowaniu całości, dlatego kluczowe znaczenie mają kotwice początku i końca wiersza (^ oraz \$). Wzorzec zapisany jako ^...\$ oznacza, że sprawdzany tekst ma w pełni odpowiadać regule, a nie zawierać ją jedynie jako fragment. Poniższe przykłady należy rozumieć jako wzorce przeznaczone przede wszystkim do użycia z grep -E albo innym narzędziem obsługującym regex rozszerzony.

W kontekście laboratoriów typowe są m.in. proste identyfikatory zbudowane z liter i cyfr, np. ^[A-Za-z][A-Za-z0-9]*\$ (wymusza rozpoczęcie od litery, a następnie dopuszcza litery i cyfry). Często waliduje się także datę lub znacznik czasu w ustalonym formacie, np. ^[0-9]{4}-[0-9]{2}-[0-9]{2}\$ dla daty YYYY-MM-DD lub ^[0-9]{4}-[0-9]{2}-[0-9]{2}[:,space:][0-9]{2}:[0-9]{2}:[0-9]{2}\$ dla formatu YYYY-MM-DD HH:MM:SS. Innym typowym przypadkiem jest linia logu zawierająca poziom komunikatu, np. ^(INFO|WARN|ERROR)[[:space:]]+.*\$, gdzie alternatywa wymusza jeden z trzech dozwolonych poziomów, a dalsza część linii może zawierać treść komunikatu.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Transformacje tekstu z użyciem sed

Narzędzie sed służy do deterministycznych transformacji strumienia tekstu, wykonywanych w modelu „wejście wiersz po wierszu → wyjście wiersz po wierszu”. Jest przydatne wtedy, gdy konieczne jest ujednolicenie formatu danych, usunięcie fragmentu wiersza, zamiana określonej sekwencji lub przygotowanie tekstu do dalszej analizy. W praktyce sed nie jest narzędziem do implementacji „logiki programu”, lecz do przekształcania danych w potoku.

Minimalny zestaw operacji obejmuje przede wszystkim zamianę z użyciem polecenia `s/old/new/`, gdzie `old` może być wzorcem (także regex), a `new` tekstem zastępującym. Przykładowo, zamiana stałego fragmentu może przyjąć postać `sed 's/ERROR/ERR/g'`, a normalizacja separatorów lub usuwanie zbędnych znaków może być realizowana analogicznie. Częstym zastosowaniem jest również „oczyszczanie” wierszy przez usuwanie prefiksu, np. usunięcie stałego nagłówka albo wycięcie części do pierwszego separatora. W tym kontekście regex w sed pozwala standaryzować dane przed kolejnymi etapami pipeline’u, tak aby dalsze filtrowanie i ekstrakcja były jednoznaczne i powtarzalne.

Ekstrakcja pól i proste agregacje: awk w zakresie minimalnym

Narzędzie awk umożliwia przetwarzanie danych w formie wierszowej z podziałem na pola (kolumny). Typowym zastosowaniem na tym etapie jest wycinanie wybranych pól, formatowanie wyjścia oraz proste zliczanie lub warunkowe raportowanie. W porównaniu do łączenia wielu etapów `grep | sed`, awk bywa korzystne wtedy, gdy logika selekcji zależy od wartości konkretnych pól, a nie wyłącznie od dopasowania wzorca w całej linii.

Podstawą pracy w awk jest rozumienie separatora pól oraz odwołań do pól w postaci `$1`, `$2`, ... (odpowiednio pierwsze, drugie pole itd.). Separator może być domyślny (białe znaki) lub ustawiany *explicite*, co pozwala dopasować analizę do formatu danych. Na poziomie minimalnym wystarczające jest wykorzystanie awk do wypisania wskazanych kolumn, ewentualnie z własnym formatowaniem, oraz do prostych agregacji, takich jak licznik zwiększany warunkowo, np. gdy określone pole ma daną wartość. Taki model pozwala realizować raporty typu „zlicz wiersze spełniające warunek w konkretnej kolumnie” bez rozbudowanej sekwencji wielu narzędzi.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Numer pola zależy bezpośrednio od formatu analizowanego wiersza. Przykładowo, jeżeli linia logu ma postać 2026-03-01 08:00:05 ERROR user=jan msg=..., to:

- \$1 oznacza datę,
- \$2 oznacza godzinę,
- \$3 oznacza poziom komunikatu.

Z tego powodu przed użyciem awk należy zawsze świadomie ustalić, jak wygląda struktura wejściowych danych tekstowych.

Typowe błędy i ryzyka dopasowań

W praktyce najczęstszym źródłem błędów jest mylenie dwóch różnych mechanizmów dopasowania: globbingu i wyrażeń regularnych. Globbing jest wykonywany przez powłokę i dotyczy nazw plików (np. *.log), natomiast regex jest interpretowany przez narzędzia przetwarzające tekst (np. grep, sed, awk) i dotyczy zawartości danych. Konsekwencją pomylenia tych pojęć bywa konstruowanie poleceń, które „wyglądają poprawnie”, ale działają na innym obiekcie niż zakładano (np. próba użycia globbingu do dopasowania fragmentu linii w pliku).

Drugim typowym ryzykiem jest brak kotwic ^ i \$ w sytuacjach, w których intencją jest walidacja całej linii lub całego ciągu. Bez kotwic wzorzec może dopasować jedynie fragment tekstu, co prowadzi do niezamierzonych akceptacji danych, które tylko częściowo spełniają regułę. Podobny problem dotyczy wzorców zbyt szerokich, które generują „fałszywe trafienia”, czyli dopasowania formalnie poprawne, ale niezgodne z intencją (np. dopasowanie ERROR we fragmencie NOERROR albo dopasowanie dowolnych cyfr bez kontroli formatu).

Istotnym źródłem problemów jest również niepoprawne cytowanie argumentów zawierających znaki specjalne. Brak cudzysłowów może spowodować, że powłoka wykona word splitting lub globbing, a w konsekwencji wzorzec albo ścieżka zostaną przekształcone przed uruchomieniem narzędzia. Dotyczy to zwłaszcza sytuacji, gdy wzorzec zawiera znaki takie jak *, ?, [], (), |, \$ lub gdy analizowane ścieżki zawierają spacje. W praktyce przyjmuje się zasadę, że wzorce regex oraz argumenty wejściowe powinny być cytowane, o ile nie ma świadomego powodu, aby powłoka wykonała dodatkowe rozwinięcia.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Osobnym przypadkiem są wzorce, w których dany token ma zostać dopasowany jako samodzielne słowo, a nie jako fragment większego ciągu. Przykładowo, przy analizie logów dopasowanie ERROR może nie być wystarczająco precyzyjne, jeżeli w danych mogą wystąpić napisy typu ERRORS, SUPERERROR albo ERROR_1. W takich sytuacjach należy stosować mechanizmy ograniczające dopasowanie do samodzielnego tokenu, na przykład odpowiednio zaprojektowany regex albo opcję `grep -w`, jeżeli jej semantyka jest zgodna z celem zadania.

Praktyczne przykłady

Poniższe przykłady należy wykonywać na wspólnym, jednoznacznie zdefiniowanym zestawie danych testowych. Dzięki temu każdy przykład odnosi się do tych samych plików wejściowych i może zostać odtworzony bez konieczności samodzielnego wymyślania struktury katalogów lub zawartości plików.

Przykłady 1–2 dotyczą nazw plików znajdujących się w katalogu roboczym. Przykłady 3–5, 7, 9–12 odnoszą się do pliku `input/app.log`. Przykład 6 odnosi się do pliku `input/dates.txt`, natomiast przykład 8 do pliku `input/prefixed.log`.

Należy pamiętać, że globbing działa na nazwach plików i jest wykonywany przez powłokę jeszcze przed uruchomieniem programu. Wyrażenia regularne są natomiast interpretowane przez narzędzia takie jak `grep`, `sed` i `awk` i dotyczą zawartości tekstu.

Przygotowanie przykładowych danych

Wszystkie poniższe przykłady i ćwiczenia należy wykonywać na danych przygotowanych w Zadaniu 0.

Przykład 1. Globbing – wybór plików po rozszerzeniu

Polecenia:

```
ls *.log
```

```
ls "*.log"
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
student@student-VirtualBox:~/Desktop/lab6$ cd input
student@student-VirtualBox:~/Desktop/lab6/input$ ls *.log
app10.log app1.log app2.log app.log prefixed.log system.log
student@student-VirtualBox:~/Desktop/lab6/input$ ls "*.log"
ls: cannot access '*.log': No such file or directory
student@student-VirtualBox:~/Desktop/lab6/input$
```

Rys. 1. Globbing i wpływ cytowania wzorca

Interpretacja:

Zaprezentowane na rysunku 1 polecenie `ls *.log` powoduje rozwinięcie wzorca `*.log` przez powłokę do listy wszystkich plików z rozszerzeniem `.log` w bieżącym katalogu. Z kolei zapis `ls "*.log"` nie zostanie rozwinięty przez powłokę i zostanie przekazany do programu dosłownie jako tekst `*.log`. Jeżeli nie istnieje plik o takiej nazwie, polecenie zakończy się błędem. Przykład pokazuje, że cytowanie wpływa na działanie globbingu.

Przykład 2. Globbing – dopasowanie dokładnie jednego znaku

Polecenie:

`ls app?.log`

Interpretacja:

Polecenie `ls app?.log` dopasowuje nazwy, w których po ciągu `app` występuje dokładnie jeden znak, a następnie rozszerzenie `.log`. Oznacza to, że pasują pliki `app1.log` i `app2.log`, natomiast nie pasuje `app10.log`, ponieważ między `app` a `.log` występują tam dwa znaki. Przykład pokazuje, że globbing ma własną semantykę i nie należy utożsamiać go z wyrażeniami regularnymi.

Przykład 3. grep – filtrowanie wierszy zawierających wzorec

Polecenie:

`grep "ERROR" input/app.log`

Interpretacja:

Polecenie wypisuje wyłącznie te wiersze pliku `input/app.log`, które zawierają ciąg znaków `ERROR`. Jest to podstawowy mechanizm selekcji danych tekstowych i jeden z najczęściej wykorzystywanych elementów potoków przetwarzania.

Przykład 4. grep – odwrócenie dopasowania



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Polecenie:

```
grep -v "INFO" input/app.log
```

Interpretacja:

Polecenie wypisuje wszystkie wiersze, które nie zawierają ciągu INFO. Opcja -v odwraca wynik dopasowania i jest użyteczna wtedy, gdy celem jest usunięcie określonej klasy wpisów zamiast wyszukania wpisów pozytywnych.

Przykład 5. grep – zliczenie liczby dopasowanych wierszy

Polecenie:

```
grep -c "WARN" input/app.log
```

Interpretacja:

Polecenie zwraca liczbę wierszy zawierających ciąg WARN. Jest to prosty przykład agregacji danych i praktyczny sposób przygotowania raportu liczbowego.

Przykład 6. Walidacja formatu przy użyciu grep -E

Polecenie:

```
grep -E '^[0-9]{4}-[0-9]{2}-[0-9]{2}$' input/dates.txt
```



```
student@student-VirtualBox:~/Desktop/lab6$ cat input/dates.txt
2026-03-01
2026-12-31
2026-3-1
01-03-2026
abc
student@student-VirtualBox:~/Desktop/lab6$ grep -E '^[0-9]{4}-[0-9]{2}-[0-9]{2}$'
input/dates.txt
2026-03-01
2026-12-31
student@student-VirtualBox:~/Desktop/lab6$
```

Rys. 2. Walidacja formatu daty przy użyciu grep -E

Interpretacja:

Polecenie wypisuje wyłącznie te wiersze, które mają postać daty w formacie YYYY-MM-DD. Zastosowanie kotwic ^ i \$ oznacza, że dopasowaniu musi



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



odpowiadać cały wiersz, a nie tylko jego fragment. W przygotowanych danych poprawne są linie 2026-03-01 i 2026-12-31, natomiast pozostałe powinny zostać odrzucone.

Przykład 7. sed – zamiana fragmentu tekstu

Polecenie:

```
sed 's/ERROR/ERR/g' input/app.log
```

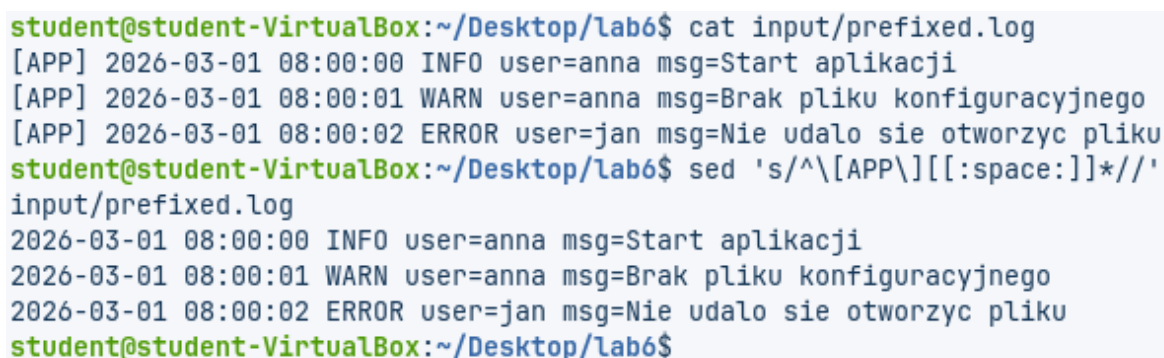
Interpretacja:

Polecenie zamienia każdą sekwencję ERROR na ERR, pozostawiając resztę wiersza bez zmian. Opcja g oznacza zamianę wszystkich wystąpień wzorca w danym wierszu. Jest to minimalny przykład pokazujący użycie sed do prostych transformacji tekstu.

Przykład 8. sed – usuwanie stałego prefiksu

Jeżeli wiersze rozpoczynają się od stałego prefiksu, można go usunąć poleceniem:

```
sed 's/^[APP]\[:space:]*//' input/prefixed.log
```



```
student@student-VirtualBox:~/Desktop/lab6$ cat input/prefixed.log
[APP] 2026-03-01 08:00:00 INFO user=anna msg=Start aplikacji
[APP] 2026-03-01 08:00:01 WARN user=anna msg=Brak pliku konfiguracyjnego
[APP] 2026-03-01 08:00:02 ERROR user=jan msg=Nie udało się otworzyć pliku
student@student-VirtualBox:~/Desktop/lab6$ sed 's/^[APP]\[:space:]*//'
input/prefixed.log
2026-03-01 08:00:00 INFO user=anna msg=Start aplikacji
2026-03-01 08:00:01 WARN user=anna msg=Brak pliku konfiguracyjnego
2026-03-01 08:00:02 ERROR user=jan msg=Nie udało się otworzyć pliku
student@student-VirtualBox:~/Desktop/lab6$
```

Rys. 3. Usuwanie prefiksu z linii logu za pomocą sed

Interpretacja:

Polecenie usuwa z początku każdego wiersza stały prefiks [APP] oraz ewentualne białe znaki następujące bezpośrednio po nim. Przykład pokazuje praktyczne użycie sed do oczyszczania danych przed dalszą analizą. Wzorec ^ oznacza początek wiersza, a [\[:space:]\]* dopuszcza dowolną liczbę spacji lub tabulatorów po prefiksie.

Przykład 9. awk – wypisanie wybranego pola

Polecenie:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
awk '{print $1}' input/app.log
```

Interpretacja

Polecenie wypisuje pierwsze pole z każdej linii, zakładając domyślny separator pól oparty na białych znakach. Jeżeli log zaczyna się od daty lub znacznika czasu, polecenie to pozwala szybko wyodrębnić tę informację. Przykład ten pokazuje podstawowy model działania awk: wiersz wejściowy jest dzielony na pola, a następnie wybrane pola mogą być wypisywane lub dalej przetwarzane.

Przykład 10. awk – warunkowe zliczanie wierszy

Polecenie:

```
awk '$3 == "ERROR" {count++} END {print count}' input/app.log
```

Interpretacja:

Polecenie zlicza wiersze, w których trzecie pole ma wartość ERROR, a następnie wypisuje wynik końcowy. Jest to prosty przykład pokazujący, że awk może służyć nie tylko do ekstrakcji pól, ale także do budowy prostych agregacji warunkowych.

Przykład 11. Potok – filtracja, transformacja i agregacja

Polecenie:

```
grep "ERROR" input/app.log | sed 's/ERROR/ERR/g' | wc -l
```

Interpretacja:

Polecenie realizuje wieloetapowe przetwarzanie danych. W pierwszym etapie wybierane są wyłącznie wiersze zawierające ERROR, w drugim etapie normalizowany jest zapis poziomego komunikatu, a w trzecim etapie zliczana jest liczba wynikowych wierszy. Przykład pokazuje ideę projektowania rozwiązania jako sekwencji prostych etapów, w których każde narzędzie odpowiada za jeden jasno określony krok.

Przykład 12. Potok – ekstrakcja i raportowanie

Polecenie:

```
grep -E 'ERROR|WARN' input/app.log | awk '{print $3}' | sort | uniq -c
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
student@student-VirtualBox:~/Desktop/lab6$ grep -E 'ERROR|WARN' input/app.  
log | awk '{print $3}' | sort | uniq -c  
2 ERROR  
2 WARN  
student@student-VirtualBox:~/Desktop/lab6$
```

Rys. 4. Potok filtrujący i zliczający poziomy logów ERROR oraz WARN

Interpretacja:

Polecenie wybiera wiersze zawierające poziomy ERROR lub WARN, następnie wyodrębnia pole poziomego logu, a na końcu zlicza liczbę wystąpień każdej wartości. Jest to dobry przykład pokazujący, że potok może łączyć filtrowanie, ekstrakcję oraz prostą agregację w jednym ciągu przetwarzania.

Ćwiczenia samodzielne

Dopuszcza się wstępną realizację w terminalu, natomiast wymagany rezultat jest skrypt Bash automatyzujący rozwiązanie. W przypadku błędnego użycia (zła liczba argumentów / zły typ danych) skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

Zadanie 0. Przygotowanie środowiska i plików testowych

Należy wykonać niniejsze zadanie **jednorazowo na początku laboratorium**, przed realizacją kolejnych zadań. Celem jest przygotowanie wspólnego zestawu plików testowych, do których będą odwoływały się dalsze przykłady i zadania. Dzięki temu wszystkie kolejne polecenia będą wykonywane na tych samych danych wejściowych.

W pierwszym kroku należy utworzyć katalog roboczy lab6 oraz podstawową strukturę podkatalogów:

```
mkdir -p lab6/input/glob_demo  
mkdir -p lab6/input/select_demo  
mkdir -p lab6/input/empty_demo  
mkdir -p lab6/output  
mkdir -p lab6/bin  
cd lab6
```

Następnie należy przygotować zestaw plików wykorzystywanych do porównań globbingu i regex.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
touch input/glob_demo/app1.log
touch input/glob_demo/app2.log
touch input/glob_demo/app10.log
touch input/glob_demo/appA.log
touch input/glob_demo/app_.log
touch input/glob_demo/system.log
touch input/glob_demo/notes.txt
```

Następnie należy utworzyć plik input/app.log, który będzie wykorzystywany w zadaniach dotyczących filtrowania, ekstrakcji pól, agregacji oraz pipeline'ów:

```
cat > input/app.log <<'EOF'
2026-03-01 08:00:00 INFO user=anna msg=Start aplikacji
2026-03-01 08:00:01 WARN user=anna msg=Brak pliku konfiguracyjnego HTTP=404
2026-03-01 08:00:02 ERROR user=jan msg=Nie udało się otworzyć pliku HTTP=503
2026-03-01 08:00:03 INFO user=jan msg=Ponowiono próbę
2026-03-01 08:00:04 error user=ola msg=Test małych liter
2026-03-01 08:00:05 WARN user=jan msg=HTTP=404 timeout
2026-03-01 08:00:06 ERROR user=anna msg=HTTP=500
2026-03-01 08:00:07 INFO user=jan msg=Koniec pracy
2026-03-01 08:00:08 ERROR msg=Brak usera HTTP=401
EOF
```

W tym pliku należy przyjąć następującą interpretację pól:

- pierwsze pole oznacza datę,
- drugie pole oznacza godzinę,
- trzecie pole oznacza poziom logu (INFO, WARN, ERROR lub inne).

Następnie należy utworzyć plik input/prefixed.log, wykorzystywany do przykładów związanych z usuwaniem prefiksu przez sed:

```
cat > input/prefixed.log <<'EOF'
[APP] 2026-03-01 08:00:00 INFO user=anna msg=Start aplikacji
[APP] 2026-03-01 08:00:01 WARN user=anna msg=Brak pliku konfiguracyjnego
[APP] 2026-03-01 08:00:02 ERROR user=jan msg=Nie udało się otworzyć pliku
EOF
```

Następnie należy utworzyć plik input/dates.txt, wykorzystywany do walidacji dat:

```
cat > input/dates.txt <<'EOF'
2026-03-01
2026-12-31
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



2026-3-1
01-03-2026
abc
EOF

Następnie należy utworzyć plik input/records.txt, wykorzystywany do walidacji całych wierszy:

```
cat > input/records.txt <<'EOF'
ID=123;USER=jan1;ROLE=ADMIN
ID=77;USER=anna;ROLE=USER
ID=AB;USER=ola;ROLE=ADMIN
ID=17;USER=jan_k;ROLE=USER
ID=5;USER=adam;ROLE=ROOT
EOF
```

Następnie należy utworzyć plik input/raw_log.log, wykorzystywany do normalizacji tekstu:

```
printf '2026-03-01\t08:10:00\tWARN\tuser=anna\tmsg=Start aplikacji \n2026-03-01 08:10:01
INFO user=jan msg=OK\t\n2026-03-01 08:10:02 WARN user=ola msg=Błąd formatu \n' >
input/raw_log.log
```

Następnie należy utworzyć plik input/error_cases.txt, wykorzystywany do zadania z dokładnym dopasowaniem tokenu ERROR:

```
cat > input/error_cases.txt <<'EOF'
ERROR
ERROR timeout
WARN ERROR WARN
ERRORS detected
SUPERERROR case
ERROR_1
INFO no problem
EOF
```

Na końcu należy przygotować katalog wykorzystywany do zadania z wyborem plików po rozszerzeniu:

```
touch input/select_demo/a.log
touch input/select_demo/b.log
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



touch input/select_demo/c.txt
touch input/select_demo/readme.md

Ćwiczenie numer 1

Należy utworzyć skrypt `bin/glob_vs_regex.sh`, który na podstawie katalogu `input/glob_demo/` przygotowanego w Zadaniu 0 wypisuje dwa zestawy wyników:

1. listę plików pasujących do wzorca globbingowego `app*.log`,
2. listę plików pasujących do wyrażenia regularnego zastosowanego do listy nazw plików, tak aby dopasowane zostały wyłącznie nazwy w postaci `app + liczby + .log`.

Raport ma zawierać dwie jednoznacznie oznaczone sekcje:

GLOBBING:

REGEX:

Każda dopasowana nazwa pliku ma zostać wypisana w osobnej linii.

Ćwiczenie numer 2

Należy utworzyć skrypt `bin/grep_filters.sh`, który przyjmuje jako argument ścieżkę do pliku tekstowego. Do testów należy wykorzystać plik `input/records.txt` przygotowany w Zadaniu 0.

Skrypt ma wygenerować raport w następującej postaci:

ROLE_OK_COUNT: <liczba>

USER_WITH_DIGIT_COUNT: <liczba>

INVALID_ROLE_LINES:

<numer_linii>:<treść_wiersza>

...

Wymagania:

1. ROLE_OK_COUNT oznacza liczbę wierszy, w których pole ROLE ma wartość ADMIN albo USER.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

2. `USER_WITH_DIGIT_COUNT` oznacza liczbę wierszy, w których wartość pola `USER` zawiera co najmniej jedną cyfrę.
3. W sekcji `INVALID_ROLE_LINES` należy wypisać wszystkie wiersze, w których pole `ROLE` ma wartość inną niż `ADMIN` albo `USER`, wraz z numerami linii.
4. W rozwiązaniu należy użyć `grep`, w tym co najmniej raz `grep -E` oraz co najmniej raz `grep -n` albo `grep -v`.
5. Skrypt ma wypisywać wyłącznie raport na `STDOUT`. Komunikaty błędów mają trafiać na `STDERR`.

Ćwiczenie numer 3

Należy utworzyć skrypt `bin/validate_lines.sh`, który przyjmuje jako argument ścieżkę do pliku tekstowego. Do testów należy wykorzystać plik `input/records.txt` przygotowany w Zadaniu 0. Skrypt ma wypisać na `STDOUT` wyłącznie te wiersze, które spełniają format:

`ID=<liczby>;USER=<litery_i_cyfry>;ROLE=<ADMIN|USER>`

Przyjmuje się, że:

- ID zawiera co najmniej jedną cyfrę,
- USER zawiera co najmniej jeden znak z zakresu A–Z, a–z, 0–9,
- ROLE może przyjmować wyłącznie wartość `ADMIN` albo `USER`.

Wymagane jest zastosowanie wyrażenia regularnego obejmującego cały wiersz, z użyciem kotwic początku i końca (^, \$). Wiersze odrzucone nie powinny być wypisywane.

Ćwiczenie numer 4

Należy utworzyć skrypt `bin/extract_field.sh`, który przyjmuje jako argument ścieżkę do pliku logu. Do testów należy wykorzystać plik `input/app.log` przygotowany w Zadaniu 0. Skrypt ma wypisać listę wartości pola `user=` występującego w wierszach logu, po jednej wartości w każdej linii.

Skrypt ma wypisywać wyłącznie nazwy użytkowników, bez prefiksu `user=`. Jeżeli wiersz nie zawiera pola `user=`, ma zostać pominięty. Przyjmuje się, że w pojedynczym wierszu występuje co najwyżej jedno pole `user=<wartość>`.

Wymagane jest użycie `sed` albo `awk` do ekstrakcji danych.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Ćwiczenie numer 5

Należy utworzyć skrypt `bin/normalize_log.sh`, który przyjmuje dokładnie dwa argumenty:

- ścieżkę do pliku wejściowego,
- ścieżkę do pliku wyjściowego.

Do testów należy wykorzystać plik `input/raw_log.log` przygotowany w Zadaniu 0. Skrypt ma wykonać następujące transformacje:

1. zamianę wielu spacji lub tabulatorów na pojedynczą spację,
2. usunięcie końcowych spacji i tabulatorów w wierszach,
3. zamianę tokenów `WARN` na `WARNING`. Przyjmuje się, że zamianie podlega samodzielny token `WARN`, a nie fragment większego słowa.

Wymagane jest użycie `sed` oraz zapis wyniku do wskazanego pliku wyjściowego.

Ćwiczenie numer 6

Należy utworzyć skrypt `bin/strict_error_match.sh`, który przyjmuje jako argument ścieżkę do pliku tekstowego. Do testów należy wykorzystać plik `input/error_cases.txt` przygotowany w Zadaniu 0.

Skrypt ma:

- wypisać liczbę linii zawierających token `ERROR` jako osobne słowo,
- następnie wypisać maksymalnie trzy pierwsze linie spełniające ten warunek.

Skrypt nie może zaliczać linii zawierających formy takie jak `ERRORS`, `SUPERERROR`, `ERROR_1`.

Raport ma mieć postać:

- `COUNT: <liczba>`
- `MATCHES:`
- następnie do trzech linii spełniających warunek.

Wymagane jest użycie wyrażenia regularnego ograniczającego dopasowanie do samodzielnego tokenu.

Ćwiczenie numer 7



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Należy utworzyć skrypt `bin/validate_ip.sh`, który przyjmuje dokładnie jeden argument i sprawdza, czy ma on postać adresu IPv4 w formacie czterech oktetów oddzielonych kropkami.

W tym zadaniu wystarczy walidacja składniowa, to znaczy:

- argument ma zawierać dokładnie cztery części oddzielone kropkami,
- każda część ma składać się z 1 do 3 cyfr.

Pełna walidacja zakresu 0–255 nie jest wymagana.

W przypadku niepoprawnego formatu skrypt ma wypisać komunikat na `STDERR` i zakończyć działanie kodem 2.

Ćwiczenie numer 8

Należy utworzyć skrypt `bin/select_files.sh`, który przyjmuje dwa argumenty:

- ścieżkę do katalogu,
- rozszerzenie plików, na przykład `log`.

Do testów należy wykorzystać katalogi `input/select_demo/` oraz `input/empty_demo/` przygotowane w Zadaniu 0.

Skrypt ma wypisać na `STDOUT` listę nazw plików znajdujących się bezpośrednio w podanym katalogu i pasujących do wzorca `*.<rozszerzenie>`, po jednej nazwie w każdej linii.

W przypadku poprawnego formatu skrypt ma wypisać `VALID` na `STDOUT` i zakończyć działanie kodem 0.

Jeżeli katalog nie zawiera żadnych dopasowań, skrypt nie może wypisywać dosłownego wzorca. W takim przypadku ma wypisać komunikat na `STDERR` i zakończyć działanie kodem 2.

Należy zaprezentować działanie skryptu co najmniej w dwóch przypadkach:

- dla katalogu `input/select_demo/`, w którym istnieją dopasowania,
- dla katalogu `input/empty_demo/`, w którym dopasowań brak.

Ćwiczenie numer 9



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Należy utworzyć skrypt `bin/log_pipeline.sh`, który przyjmuje jako argument ścieżkę do pliku logu. Do testów należy wykorzystać plik `input/app.log` przygotowany w Zadaniu 0.

Skrypt ma wygenerować raport w następującej postaci:

HTTP_CODES:

HTTP=<kod>

HTTP=<kod>

...

HTTP_UNIQUE_COUNT: <liczba>

HTTP_COUNTS:

HTTP=<kod> COUNT=<liczba>

HTTP=<kod> COUNT=<liczba>

...

Wymagania:

1. W sekcji HTTP_CODES należy wypisać unikalne kody HTTP występujące w pliku, po jednym kodzie w każdej linii, w porządku rosnącym.
2. HTTP_UNIQUE_COUNT oznacza liczbę różnych kodów HTTP występujących w pliku.
3. W sekcji HTTP_COUNTS należy wypisać każdy kod HTTP wraz z liczbą jego wystąpień, również w porządku rosnącym.
4. Dopuszcza się założenie, że wiersze zawierają fragment `HTTP=<liczby>`.
5. Wymagane jest użycie potoków oraz co najmniej jednego z narzędzi `grep -E`, `sed` lub `awk` do ekstrakcji danych.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



6. Skrypt ma wypisywać wyłącznie raport na STDOUT. Komunikaty błędów, takie jak brak pliku wejściowego, mają trafiać na STDERR.

Pytania pomocnicze

1. Na czym polega różnica między szablonami powłoki (globbing) a wyrażeniami regularnymi i kiedy każde z nich ma zastosowanie?
2. Jakie znaczenie mają znaki *, ? oraz nawiasy [...] w globbingu i dlaczego nie są one równoważne operatorom regex?
3. Czym różni się grep od grep -E i jakie konstrukcje stają się dostępne w trybie rozszerzonych wyrażań regularnych?
4. Do czego służą kotwice ^ i \$ w wyrażeniach regularnych i dlaczego są istotne w walidacji całych wierszy?
5. Czym są klasy znaków, np. [0-9], [A-Za-z], [[:digit:]], [[:space:]], i czym różni się zapis „klasy POSIX” od klasy znaków w nawiasach []?
6. Jak działa pojęcie „granicy słowa” w regex (np. dopasowanie tokenu jako osobnego słowa) i jakie błędy powoduje brak takiego ograniczenia?
7. Na czym polega różnica między dopasowaniem „dowolnego fragmentu” a dopasowaniem „całej linii” oraz jakie są konsekwencje w przetwarzaniu logów?
8. Jakie znaczenie mają kwantyfikatory (np. *, +, ?, {m,n}) oraz alternatywa |, i jak wpływają na szerokość dopasowania?
9. Dlaczego nieostrożne wzorce mogą prowadzić do „niezamierzonych dopasowań” i jak ograniczać je przez kotwice, klasy znaków oraz bardziej precyzyjne warunki?
10. Jakie są typowe opcje grep przydatne w analizie logów (np. -i, -n, -v, -c) i jak wpływają na wynik?
11. Do czego służy sed w kontekście transformacji tekstu i jakie są typowe operacje realizowane poleceniem s///?
12. Do czego służy awk w kontekście ekstrakcji pól i agregacji oraz kiedy jest praktyczniejszy niż łączenie wielu wywołań grep i sed?
13. Dlaczego w zadaniach wymagane jest raportowanie w deterministycznej kolejności i jak zapewnić stały porządek wyników w raporcie?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Podsumowanie

W ramach laboratorium szóstego utrwalono techniki dopasowań tekstowych w skryptach powłoki, ze szczególnym rozróżnieniem szablonów powłoki oraz wyrażeń regularnych stosowanych w narzędziach takich jak grep. Przećwiczono budowę i weryfikację podstawowych wzorców dopasowania wykorzystywanych w filtrowaniu, walidacji i selekcji danych tekstowych, w tym stosowanie kotwic, klas znaków, powtórzeń, grupowania i alternatyw. Wprowadzono również transformacje tekstu z użyciem sed oraz ekstrakcję i agregację informacji z wykorzystaniem awk, w zakresie umożliwiającym tworzenie prostych, przewidywalnych łańcuchów przetwarzania danych. Zwrócono uwagę na znaczenie precyzji reguł dopasowania, analizę przypadków brzegowych oraz ryzyka wynikające ze zbyt szerokich dopasowań. Rezultatem laboratorium jest umiejętność świadomego doboru techniki dopasowania i narzędzia do problemu.

Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Materiały do przedmiotu:

Programowanie skryptowe

Zajęcia laboratoryjne nr 7

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	3
Ćwiczenia samodzielne	12
Pytania pomocnicze	25
Podsumowanie	26
Literatura	26



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Laboratorium siódme ma na celu nauczanie integracji skryptów z narzędziami zewnętrznymi oraz zarządzania zależnościami środowiskowymi. W ramach zajęć realizowane jest sprawdzanie dostępności wymaganych programów (command -v), identyfikowanie wersji, instalowanie brakujących narzędzi w systemie opartym o Debian (apt) oraz konfigurowanie środowiska uruchomieniowego (zmienne środowiskowe, pliki konfiguracyjne, parametry). Efektem ma być skrypt, który jednoznacznie deklaruje zależności i potrafi przeprowadzić podstawową autoweryfikację środowiska.

Cel ćwiczenia/laboratorium

- Uporządkowanie pojęcia zależności skryptu od narzędzi zewnętrznych oraz określanie minimalnych wymagań środowiskowych niezbędnych do poprawnego uruchomienia.
- Wprowadzenie procedur weryfikacji dostępności wymaganych programów oraz ich wersji, tak aby możliwe było jednoznaczne wykrywanie braków w środowisku.
- Konfiguracja środowiska uruchomieniowego skryptu w sposób kontrolowany i powtarzalny, z uwzględnieniem zmiennych środowiskowych oraz plików konfiguracyjnych.
- Realizacja prostego mechanizmu inicjalizacji środowiska (np. przygotowanie katalogów roboczych, wskazanie ścieżek, ustawienia parametrów), umożliwiającego uruchamianie skryptu w różnych warunkach.
- Zapewnienie podstawowej przenośności skryptu między maszynami i systemami z tej samej rodziny, poprzez eliminowanie zależności niejawnych i stosowanie jednoznacznych komunikatów o wymaganiach.

Instrukcja laboratoryjna/ćwiczeniowa

Zależności skryptu i wymagania środowiskowe



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Skrypt Bash bardzo rzadko działa całkowicie samodzielnie. W praktyce korzysta on z programów zewnętrznych, plików konfiguracyjnych, katalogów roboczych, zmiennych środowiskowych oraz określonej struktury projektu. Wszystkie te elementy stanowią zależności skryptu. Jeżeli którykolwiek z nich nie jest dostępny albo ma niepoprawną postać, wykonanie skryptu może zakończyć się błędem lub dawać wyniki niezgodne z oczekiwaniami.

Poprawnie zaprojektowany skrypt powinien możliwie wcześnie wykrywać brakujące zależności i jednoznacznie informować o wymaganiach środowiskowych. Dzięki temu użytkownik nie uruchamia logiki głównej w niepoprawnym środowisku, lecz od razu otrzymuje jasny komunikat wskazujący przyczynę problemu.

W tym laboratorium zależnościami będą w szczególności:

- interpreter bash,
- podstawowe narzędzia tekstowe, takie jak grep, sed i awk,
- narzędzia środowiska kontenerowego, takie jak docker i docker compose,
- pliki konfiguracyjne w katalogu config/,
- katalogi robocze projektu, np. reports/, logs/, data/.

Sprawdzanie dostępności narzędzi zewnętrznych

Podstawowym sposobem sprawdzenia, czy dane narzędzie jest dostępne w systemie, jest użycie polecenia `command -v`. Rozwiązanie to jest preferowane, ponieważ umożliwia ustalenie, czy wskazana komenda jest widoczna w bieżącym środowisku uruchomieniowym.

Przykład:

```
if command -v docker >/dev/null 2>&1; then
    echo "docker jest dostępny"
else
    echo "Brak docker" >&2
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zaletą takiego podejścia jest prostota oraz możliwość wykorzystania tej samej logiki dla wielu narzędzi. Zamiast zakładać, że dany program jest obecny, skrypt powinien sprawdzić to jawnie przed wykonaniem operacji wymagającej tego programu.

W praktyce laboratoryjnej warto przyjąć zasadę, że:

- brak argumentu lub błędne użycie kończy się kodem 2,
- brak wymaganego narzędzia kończy się kodem 1,
- poprawne wykonanie kończy się kodem 0.

Weryfikacja wersji programów

Sama obecność programu nie zawsze jest wystarczająca. W niektórych przypadkach istotna jest również jego wersja, ponieważ składnia poleceń lub zakres obsługiwanych opcji programu może różnić się między wydaniem.

Informację o wersji najczęściej uzyskuje się za pomocą jednej z opcji:

--version,

version,

rzadziej -V.

Przykłady:

- `bash --version`
- `grep --version`
- `docker --version`
- `docker compose version`



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W zadaniach laboratoryjnych raportowanie wersji ma dwa cele. Po pierwsze pozwala potwierdzić, że środowisko spełnia minimalne wymagania. Po drugie umożliwia porównanie środowisk między różnymi maszynami, co jest ważne przy analizie przyczyn błędów lub różnic w działaniu skryptu.

Instalacja brakujących narzędzi

Jeżeli wymagane narzędzie nie jest dostępne, użytkownik powinien otrzymać informację o brakującej zależności oraz, opcjonalnie, propozycję sposobu jej instalacji. W systemach opartych o Debian podstawowym menedżerem pakietów jest apt.

Przykłady:

- `sudo apt update`
- `sudo apt install -y curl`
- `sudo apt install -y grep sed gawk`

Warto podkreślić, że instalacja zależności nie powinna być wykonywana automatycznie bez wyraźnej zgody użytkownika. Skrypt może zaproponować odpowiednią komendę lub wykonać instalację dopiero po użyciu jawnego przełącznika, na przykład `--install`. Takie podejście zmniejsza ryzyko nieoczekiwanych zmian w systemie.

W przypadku narzędzi bardziej złożonych, takich jak Docker, skrypt laboratoryjny powinien raczej sygnalizować brak i podać komunikat diagnostyczny niż samodzielnie próbować przeprowadzać pełną instalację.

Konfiguracja skryptu i źródła ustawień

Skrypt, który korzysta z narzędzi zewnętrznych i wielu parametrów, nie powinien mieć wszystkich ustawień wpisanych bezpośrednio w kodzie głównym. Znacznie



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



lepszym rozwiązaniem jest oddzielenie logiki od konfiguracji. W praktyce można używać kilku źródeł ustawień:

- wartości domyślnych zapisanych w kodzie,
- pliku konfiguracyjnego w postaci KEY=VALUE,
- zmiennych środowiskowych,
- argumentów uruchomieniowych.

Takie podejście zwiększa czytelność rozwiązania oraz umożliwia uruchamianie tego samego skryptu w różnych warunkach bez konieczności każdorazowej modyfikacji kodu.

Przykładowe ustawienia:

APP_NAME="lab7-demo"

COMPOSE_FILE="compose.yaml"

ENV_FILE="config/default.env"

OUTPUT_DIR="reports"

PROJECT_NAME="demo_app"

Pliki .env i zmienne środowiskowe

Plik .env jest prostym sposobem przechowywania ustawień konfiguracyjnych w postaci par KEY=VALUE. Może on zawierać ścieżki, nazwy projektu, porty, znaczniki wersji obrazu lub inne parametry potrzebne do uruchomienia środowiska.

Przykład pliku:

PROJECT_NAME=demo_app

APP_PORT=8080

IMAGE_TAG=latest



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

`COMPOSE_FILE=compose.yaml`

Zmienne środowiskowe różnią się od ustawień zapisanych w pliku tym, że istnieją w bieżącym środowisku procesu. Mogą być ustawiane tymczasowo na czas jednego uruchomienia:

```
APP_PORT=9090 bin/show_config.sh
```

lub eksportowane na dłużej:

```
export APP_PORT=9090
```

Dzięki temu możliwe jest wygodne nadpisywanie ustawień bez zmiany plików konfiguracyjnych.

Priorytet źródeł konfiguracji

Jeżeli ten sam parametr może pochodzić z kilku źródeł, konieczne jest ustalenie jasnego priorytetu. Brak takiej reguły prowadzi do niejednoznaczności i utrudnia diagnozowanie problemów.

W laboratorium siódmym warto przyjąć następujący porządek:

- argumenty uruchomieniowe,
- zmienne środowiskowe,
- plik konfiguracyjny,
- wartości domyślne zapisane w kodzie.

Takie rozwiązanie jest czytelne i zgodne z praktyką wielu narzędzi systemowych. Użytkownik może uruchomić skrypt z konfiguracją domyślną, nadpisać część ustawień przez środowisko, a w razie potrzeby jednorazowo wymusić konkretną wartość przy pomocy argumentu.

Autoweryfikacja środowiska



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zanim skrypt zacznie wykonywać logikę główną, powinien sprawdzić, czy środowisko jest gotowe do pracy. Taki etap nazywa się często autoweryfikacją środowiska lub preflight check.

Autoweryfikacja może obejmować:

- sprawdzenie obecności wymaganych programów,
- sprawdzenie wersji wybranych narzędzi,
- weryfikację istnienia plików konfiguracyjnych,
- sprawdzenie istnienia wymaganych katalogów,
- kontrolę poprawności podstawowych parametrów.

Dzięki temu błędy są wykrywane wcześniej, a użytkownik otrzymuje krótką listę braków zamiast niejasnego komunikatu po uruchomieniu połowy skryptu.

Przykładowy wynik działania etapu preflight:

OK: bash (5.2.21)

OK: grep (3.8)

OK: docker (27.0.0)

MISSING: docker compose

OK: config/default.env

MISSING: compose.yaml

Jeżeli brakuje elementu krytycznego, skrypt powinien zakończyć działanie kodem błędu i nie przechodzić do dalszych etapów.

Inicjalizacja środowiska projektu

Po sprawdzeniu zależności skrypt może przejść do inicjalizacji środowiska projektu. Chodzi tu o przygotowanie minimalnej struktury niezbędnej do dalszej pracy, na przykład:

- utworzenie katalogów reports/, logs/, data/,
- sprawdzenie obecności pliku konfiguracji,
- zapis raportu startowego,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- wyliczenie finalnych wartości ustawień.

Taki etap ma na celu przygotowanie środowiska w sposób kontrolowany i powtarzalny.

Przykład:

```
mkdir -p reports logs data
```

Inicjalizacja środowiska stanowi naturalny krok pomiędzy prostą weryfikacją zależności a uruchomieniem właściwej logiki aplikacji.

Profile środowiskowe

W praktyce ten sam skrypt może być uruchamiany w różnych wariantach środowiska, mogą to być na przykład:

- środowisko deweloperskie,
- środowisko testowe,
- środowisko produkcyjne.

Każdy profil może korzystać z innego pliku konfiguracyjnego i innych wartości parametrów. Dzięki temu jedna logika skryptu może być używana w wielu warunkach, a różnice są wyrażone wyłącznie przez konfigurację.

Przykładowa struktura:

```
config/
```

```
|— default.env
```

```
|— dev.env
```

```
|— test.env
```

```
|— prod.env
```

Taki mechanizm jest szczególnie przydatny wtedy, gdy różni się:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- nazwa projektu,
- port aplikacji,
- ścieżka do pliku Compose,
- katalog raportów,
- znacznik wersji obrazu.

Docker i Docker Compose jako przykład zależności zewnętrznej

Docker jest przykładem bardziej złożonej zależności zewnętrznej. Dzięki temu można pokazać, że skrypt nie tylko sprawdza pojedynczą komendę, ale także kontroluje gotowość całego środowiska uruchomieniowego.

W praktyce skrypt może weryfikować:

- dostępność docker,
- dostępność docker compose,
- obecność pliku compose.yaml,
- obecność pliku .env,
- istnienie katalogów wykorzystywanych przez projekt.

Przykładowe polecenia:

```
docker --version
```

```
docker compose version
```

```
docker compose -f compose.yaml config
```

Szczególnie istotne jest polecenie `docker compose ... config`, które pozwala sprawdzić poprawność konfiguracji bez pełnego uruchamiania środowiska. Dzięki temu student może potraktować Docker jako konkretny przykład narzędzia zewnętrznego wymagającego:

- sprawdzenia dostępności,
- sprawdzenia wersji,
- sprawdzenia plików konfiguracyjnych,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- raportowania błędów środowiskowych.

Raportowanie stanu środowiska

Oprócz komunikatów wyświetlanych na bieżąco warto tworzyć również raporty zapisywane do plików. Raport taki może zawierać:

- datę uruchomienia,
- finalne ustawienia konfiguracji,
- listę wykrytych narzędzi,
- wersje programów,
- status sprawdzenia plików i katalogów.

Przykład prostego raportu:

```
DATE=2026-03-11 14:30:00  
PROJECT_NAME=demo_app  
COMPOSE_FILE=compose.yaml  
ENV_FILE=config/default.env  
DOCKER=OK  
DOCKER_COMPOSE=OK  
REPORTS_DIR=OK  
LOGS_DIR=OK  
DATA_DIR=OK
```

Raporty są szczególnie przydatne przy porównywaniu wyników między stanowiskami oraz przy późniejszym diagnozowaniu problemów.

Komunikaty diagnostyczne i przenośność skryptu

Skrypt powinien możliwie jasno rozróżniać:

- dane wynikowe wypisywane na STDOUT,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- komunikaty diagnostyczne wypisywane na STDERR.

Taki podział pozwala łączyć skrypt z innymi narzędziami, przekierowywać wyniki do plików oraz łatwiej analizować błędy. Równie ważne jest unikanie zależności niejawnych, takich jak:

- ukryte założenia o istnieniu katalogu roboczego,
- brak sprawdzenia plików wejściowych,
- brak informacji o wymaganych komendach,
- nadpisywanie ustawień bez jasnych reguł.

Im bardziej jawnie skrypt określa swoje wymagania, tym łatwiej przenieść go na inną maszynę z tej samej rodziny systemowej i przewidzieć jego zachowanie.

Ćwiczenia samodzielne

Dopuszcza się wstępną realizację w terminalu, natomiast wymaganym rezultatem jest skrypt Bash automatyzujący rozwiązanie. W przypadku błędnego użycia skryptu (np. niewłaściwa liczba argumentów lub niepoprawny format danych wejściowych) należy wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

Zadanie 0. Przygotowanie struktury projektu i plików konfiguracyjnych

Należy utworzyć katalog lab7/ z następującą strukturą podkatalogów:

lab7/

|— bin/

|— lib/

|— config/

|— reports/

|— data/

|— logs/

W katalogu config/ należy przygotować plik default.env zawierający podstawową konfigurację projektu w formacie KEY=VALUE.

Przykładowa zawartość pliku config/default.env:

APP_NAME=lab7-demo



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
PROJECT_NAME=demo_app
OUTPUT_DIR=reports
LOG_DIR=logs
DATA_DIR=data
COMPOSE_FILE=compose.yaml
ENV_FILE=config/compose.env
```

Należy przyjąć, że plik konfiguracyjny ma prostą postać: jedna linia oznacza jedną parę KEY=VALUE, bez spacji wokół znaku =.

W tym laboratorium wystarczy obsługiwać następujące klucze:

```
APP_NAME
PROJECT_NAME
OUTPUT_DIR
LOG_DIR
DATA_DIR
COMPOSE_FILE
ENV_FILE
```

Jeżeli w pliku pojawią się inne klucze, skrypty mogą je zignorować.

To zadanie przygotowuje wspólne środowisko pracy dla kolejnych ćwiczeń.

Zadanie 1. Ładowanie ustawień z pliku konfiguracyjnego

Należy utworzyć skrypt: bin/env_file_loader.sh

Skrypt ma ładować ustawienia z pliku konfiguracyjnego wskazanego argumentem:

--env=ścieżka_do_pliku_env

Przykład uruchomienia:

./bin/env_file_loader.sh --env=config/default.env

Skrypt ma:

- sprawdzić, czy podano argument --env=...,
- sprawdzić, czy wskazany plik istnieje i jest zwykłym plikiem,
- odczytać z niego obsługiwane ustawienia,
- wypisać na STDOUT wybrane wartości w formacie KEY=VALUE.

Wynik należy wypisać w stałej kolejności:

```
APP_NAME=...
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
PROJECT_NAME=...  
OUTPUT_DIR=...  
LOG_DIR=...  
DATA_DIR=...  
COMPOSE_FILE=...  
ENV_FILE=...
```

Jeżeli w pliku brakuje któregoś z powyższych kluczy, skrypt powinien wypisać dla niego wartość:

KEY=UNSET

Na przykład:

```
APP_NAME=lab7-demo  
PROJECT_NAME=demo_app  
OUTPUT_DIR=reports  
LOG_DIR=logs  
DATA_DIR=data  
COMPOSE_FILE=compose.yaml  
ENV_FILE=config/compose.env
```

W przypadku błędnego użycia, na przykład braku argumentu `--env=...`, podania nieznanego argumentu albo argumentu bez wartości, skrypt ma wypisać komunikat użycia na `STDERR` i zakończyć działanie kodem 2.

Przykład komunikatu użycia:

Usage: ./bin/env_file_loader.sh --env=FILE

Jeżeli wskazany plik nie istnieje, skrypt ma wypisać czytelny komunikat na `STDERR` i zakończyć działanie kodem 1.

To zadanie ma sprawdzić wyłącznie ładowanie ustawień z pliku. Priorytet argumentów, zmiennych środowiskowych, pliku konfiguracyjnego i wartości domyślnych zostanie rozwinięty w kolejnym zadaniu.

Zadanie 2. Finalna konfiguracja projektu i priorytet źródeł ustawień

Należy utworzyć skrypt: `bin/show_config.sh`



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Skrypt ma korzystać ze struktury projektu i pliku config/default.env przygotowanych w zadaniu 0.

Skrypt ma wypisać na STDOUT finalne wartości następujących ustawień:

APP_NAME=...

PROJECT_NAME=...

OUTPUT_DIR=...

LOG_DIR=...

DATA_DIR=...

Wynik należy wypisać zawsze w powyższej kolejności.

Skrypt ma obsługiwać cztery źródła konfiguracji. Ich priorytet, od najwyższego do najniższego, jest następujący:

1. argumenty uruchomieniowe,
2. zmienne środowiskowe,
3. plik config/default.env,
4. wartości domyślne zapisane bezpośrednio w skrypcie.

Przez wartości domyślne zapisane bezpośrednio w skrypcie należy rozumieć początkowe wartości zmiennych ustawione w pliku bin/show_config.sh, na przykład:

APP_NAME="default-app"

PROJECT_NAME="default-project"

OUTPUT_DIR="reports"

LOG_DIR="logs"

DATA_DIR="data"

Wartości te mają zostać użyte tylko wtedy, gdy dana zmienna nie została ustawiona w żadnym źródle o wyższym priorytecie.

Skrypt ma obsługiwać argumenty uruchomieniowe w postaci:

--app-name=...

--project-name=...



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



--output-dir=...

--log-dir=...

--data-dir=...

Przykładowe uruchomienia:

./bin/show_config.sh

PROJECT_NAME=test_env ./bin/show_config.sh

PROJECT_NAME=test_env ./bin/show_config.sh --project-name=test_arg

--output-dir=out

Dla ostatniego przykładu wartość PROJECT_NAME powinna pochodzić z argumentu **--project-name**, ponieważ argument uruchomieniowy ma wyższy priorytet niż zmienna środowiskowa.

Jeżeli w pliku config/default.env pojawią się inne klucze niż obsługiwane przez skrypt, można je zignorować.

W przypadku błędnego użycia, na przykład podania nieznanego argumentu albo argumentu bez wartości, skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

Przykład komunikatu użycia:

**Usage: ./bin/show_config.sh [--app-name=NAME] [--project-name=NAME]
[--output-dir=DIR] [--log-dir=DIR] [--data-dir=DIR]**

Zadanie 3. Sprawdzanie dostępności pojedynczego programu

Należy utworzyć skrypt bin/require_cmd.sh, który przyjmuje jako argument nazwę programu i sprawdza jego dostępność w systemie za pomocą polecenia command -v. Jeżeli wskazany program jest dostępny, skrypt ma wypisać na STDOUT ścieżkę do programu i zakończyć działanie kodem 0.

Jeżeli wskazany program nie jest dostępny, skrypt ma wypisać na STDERR komunikat o brakującej zależności i zakończyć działanie kodem 1.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W przypadku braku argumentu lub podania niewłaściwej liczby argumentów skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

Jako przykładowe narzędzia należy przetestować co najmniej:

- a. bash
- b. docker
- c. jedno wybrane narzędzie tekstowe spośród: grep, sed, awk

Brak programu docker na stanowisku nie oznacza błędnego rozwiązania, jeżeli skrypt poprawnie wykrywa jego brak, wypisuje komunikat diagnostyczny na STDERR i kończy działanie kodem 1.

Przykładowe uruchomienia:

```
./bin/require_cmd.sh bash
```

```
./bin/require_cmd.sh docker
```

```
./bin/require_cmd.sh grep
```

Zadanie 4. Biblioteka zależności i podstawowy preflight

Należy utworzyć plik biblioteki lib/deps.sh.

W pliku należy zdefiniować co najmniej dwie funkcje:

- a. require_cmd
- b. get_version

Funkcja require_cmd ma sprawdzać dostępność programu za pomocą command -v.

Funkcja get_version ma próbować odczytać wersję programu w postaci tekstowej.

Jeżeli nie uda się jednoznacznie ustalić wersji programu, funkcja get_version może zwrócić wartość UNKNOWN.

Następnie należy utworzyć skrypt bin/preflight.sh, który dołącza bibliotekę lib/deps.sh za pomocą source i wykonuje podstawową autoweryfikację środowiska projektu.

Skrypt ma sprawdzać:

- bash
- grep
- sed
- awk
- config/default.env



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- logs/
- data/

Raport preflight ma być wypisany na STDOUT w stałej kolejności:

- bash
- grep
- sed
- awk
- config/default.env
- logs/
- data/

Przykładowy format raportu:

OK: bash (5.2.21)

OK: grep (3.8)

OK: sed (4.9)

OK: awk (5.2.1)

OK: config/default.env

OK: logs/

OK: data/

Jeżeli dany element nie jest dostępny, należy wypisać linię w formacie:

MISSING: <nazwa_elementu>

Za wymagania krytyczne uznaje się:

- dostępność bash, grep, sed i awk,
- istnienie pliku config/default.env,
- istnienie katalogów logs/ oraz data/.

Jeżeli brakuje co najmniej jednego wymagania krytycznego, skrypt ma zakończyć działanie kodem 1.

W przypadku błędnego użycia skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dla chętnych można rozszerzyć skrypt bin/preflight.sh o przełącznik --report=ścieżka, który zapisze wykryte wersje narzędzi do pliku raportowego. Raport powinien zawierać co najmniej DATE, BASH_VERSION, GREP_VERSION, SED_VERSION oraz AWK_VERSION.

Zadanie 5. Walidacja konfiguracji Docker Compose

Należy utworzyć skrypt: bin/compose_config_check.sh

Skrypt ma sprawdzać, czy konfiguracja projektu kontenerowego jest kompletna i poprawna składniowo. Skrypt nie ma uruchamiać kontenerów. Jego zadaniem jest wyłącznie sprawdzenie zależności, plików konfiguracyjnych oraz poprawności konfiguracji za pomocą polecenia:

```
docker compose config
```

W katalogu lab7/ należy przygotować plik compose.yaml:

```
services:
  app:
    image: "nginx:${IMAGE_TAG}"
    ports:
      - "${APP_PORT}:80"
```

W katalogu config/ należy przygotować plik compose.env:

```
APP_PORT=8080
IMAGE_TAG=latest
```

W pliku config/default.env należy dodać lub uzupełnić ustawienia:

```
COMPOSE_FILE=compose.yaml
ENV_FILE=config/compose.env
PROJECT_NAME=demo_app
```

Skrypt ma obsługiwać opcjonalne argumenty uruchomieniowe:

```
--compose=ścieżka_do_pliku_compose
--env=ścieżka_do_pliku_env
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Jeżeli argumenty nie zostaną podane, skrypt ma odczytać wartości COMPOSE_FILE oraz ENV_FILE z pliku config/default.env.

Jeżeli dana wartość nie występuje ani w argumentach, ani w pliku config/default.env, należy użyć wartości domyślnych:

COMPOSE_FILE=compose.yaml

ENV_FILE=config/compose.env

Skrypt ma wykonać kolejno następujące kroki:

- a. sprawdzić dostępność polecenia docker,
- b. sprawdzić dostępność polecenia docker compose,
- c. sprawdzić istnienie pliku wskazanego przez COMPOSE_FILE,
- d. sprawdzić istnienie pliku wskazanego przez ENV_FILE,
- e. upewnić się, że katalog reports/ istnieje,
- f. sprawdzić poprawność konfiguracji i zapisać rozwiniętą konfigurację do pliku:
reports/compose_resolved.yaml

Do sprawdzenia konfiguracji należy użyć polecenia na przykład:

```
docker compose -p "$PROJECT_NAME" -f "$COMPOSE_FILE" --env-file  
"$ENV_FILE" config > reports/compose_resolved.yaml
```

W przypadku poprawnej konfiguracji skrypt ma wypisać na STDOUT raport:

COMPOSE_VALID=YES

COMPOSE_FILE=<użyty_plik_compose>

ENV_FILE=<użyty_plik_env>

RESOLVED_CONFIG=reports/compose_resolved.yaml



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Jeżeli docker albo docker compose nie są dostępne na stanowisku, poprawnym zachowaniem skryptu jest wypisanie komunikatu diagnostycznego na STDERR i zakończenie działania kodem 1.

W przypadku błędnego użycia, na przykład podania nieznanego argumentu albo argumentu bez wartości, skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

Przykład poprawnego uruchomienia:

```
./bin/compose_config_check.sh
```

Przykład uruchomienia z argumentami:

```
./bin/compose_config_check.sh --compose=compose.yaml --env=config/compose.env
```

Zadanie 6. Zadanie dodatkowe A Inicjalizacja środowiska projektu

Należy utworzyć skrypt bin/bootstrap.sh, który realizuje pełną inicjalizację środowiska uruchomieniowego dla projektu w katalogu lab7/.

W rozwiązaniu należy wykorzystać funkcje lub skrypty przygotowane we wcześniejszych zadaniach, w szczególności lib/deps.sh oraz logikę ładowania konfiguracji. Nie należy implementować całej logiki od zera.

Skrypt ma wykonać kolejno następujące kroki:

- wczytać konfigurację z pliku config/default.env,
- nadpisać ustawienia przez zmienne środowiskowe oraz argumenty uruchomieniowe,
- wykonać autoweryfikację wymaganych narzędzi z użyciem funkcji z lib/deps.sh,
- utworzyć brakujące katalogi robocze, w szczególności reports/, logs/, data/ oraz katalog wskazany przez OUTPUT_DIR,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- e. sprawdzić obecność pliku wskazanego przez COMPOSE_FILE oraz pliku wskazanego przez ENV_FILE,
- f. wygenerować raport środowiska do pliku reports/bootstrap_report.txt

Raport ma zawierać co najmniej:

DATE=...
APP_NAME=...
PROJECT_NAME=...
OUTPUT_DIR=...
COMPOSE_FILE=...
ENV_FILE=...
DEPENDENCIES_STATUS=...

Skrypt ma obsługiwać co najmniej tryb --strict

W trybie --strict brak dowolnej zależności krytycznej ma zakończyć działanie skryptu kodem 1.

W przypadku błędnego użycia skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

Zadanie 7. Zadanie dodatkowe B Profile środowiskowe

Należy utworzyć skrypt bin/profile_runner.sh, który przyjmuje nazwę profilu środowiska oraz opcjonalne argumenty nadpisujące wybrane ustawienia.

Skrypt ma obsługiwać profile:

- a. dev
- b. test
- c. prod

Dla każdego profilu należy przygotować osobny plik konfiguracyjny w katalogu config/.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykładowe pliki:

```
config/dev.env  
PROJECT_NAME=demo_dev  
APP_PORT=8080  
IMAGE_TAG=dev
```

```
config/test.env  
PROJECT_NAME=demo_test  
APP_PORT=8081  
IMAGE_TAG=test
```

```
config/prod.env  
PROJECT_NAME=demo_prod  
APP_PORT=80  
IMAGE_TAG=latest
```

Skrypt ma:

- wczytać plik konfiguracyjny odpowiadający wybranemu profilowi,
- uwzględnić zmienne środowiskowe jako źródło nadpisujące konfigurację,
- uwzględnić argumenty uruchomieniowe jako źródło o najwyższym priorytecie,
- uruchomić bin/preflight.sh,
- wypisać finalne ustawienia środowiska w czytelnej, deterministycznej formie,
- wygenerować raport do pliku:
- reports/profile_<nazwa>.txt

Raport powinien zawierać informację, który profil został użyty, który plik konfiguracyjny został wczytany oraz który plik Compose został wskazany.

W przypadku podania nieznanego profilu skrypt ma wypisać komunikat błędu na STDERR i zakończyć działanie kodem 2.

Zadanie 8. Zadanie dodatkowe C Indeks dostępnych skryptów



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Należy utworzyć skrypt `bin/help_index.sh`, który wypisuje na STDOUT listę dostępnych skryptów znajdujących się w katalogu `bin/` wraz z jednolinijkowym opisem każdego z nich.

Opis ma być pobierany z nagłówka danego skryptu, z linii zaczynającej się od:

DESC:

Przykład nagłówka w skrypcie:

DESC: sprawdza dostępność wskazanego programu

Wynik działania `bin/help_index.sh` ma być deterministyczny, czyli skrypty powinny być wypisane w stałej kolejności, na przykład alfabetycznie według nazwy pliku.

Przykładowy format wyniku:

- a. `bootstrap.sh` - inicjalizuje środowisko projektu
- b. `preflight.sh` - sprawdza podstawowe zależności środowiska
- c. `require_cmd.sh` - sprawdza dostępność wskazanego programu

Jeżeli dany skrypt nie zawiera linii `# DESC:`, skrypt `bin/help_index.sh` powinien wypisać przy nim opis: brak opisu

Zadanie 9. Zadanie dodatkowe D Propozycja instalacji brakujących narzędzi

Należy utworzyć skrypt `bin/install_missing.sh`, który sprawdza dostępność podstawowych narzędzi tekstowych, a następnie wypisuje proponowaną komendę instalacji dla systemu opartego o Debian.

Skrypt ma sprawdzać co najmniej następujące programy:

- a. `grep`
- b. `sed`
- c. `awk`
- d. `curl`



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W zadaniu należy obsłużyć co najmniej następujące mapowanie programów na pakiety:

grep -> grep

sed -> sed

awk -> gawk

curl -> curl

Domyślnie skrypt nie wykonuje instalacji. Ma jedynie wypisać na STDOUT proponowaną komendę instalacji brakujących pakietów, na przykład:

sudo apt update && sudo apt install -y gawk curl

Instalacja może zostać wykonana wyłącznie po podaniu przełącznika:
--install

Jeżeli skrypt uruchomiono z przełącznikiem --install, może zainstalować tylko te narzędzia, które zostały jednoznacznie przypisane do standardowych pakietów Debiana.

Dopuszcza się uwzględnienie programu docker wyłącznie informacyjnie. Skrypt może zgłosić jego brak, ale nie powinien podejmować automatycznej instalacji Dockera.

Jeżeli apt nie jest dostępny, skrypt ma wypisać komunikat błędu na STDERR i zakończyć działanie kodem 1.

W przypadku błędnego użycia skrypt ma wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Pytania pomocnicze

1. Co rozumie się przez zależności skryptu i jakie elementy środowiska mogą być takimi zależnościami?
2. Dlaczego poprawnie zaprojektowany skrypt powinien jawnie deklarować swoje wymagania środowiskowe?
3. Do czego służy command -v i dlaczego jest preferowany przy sprawdzaniu dostępności narzędzi?
4. W jaki sposób można uzyskać informację o wersji programu i dlaczego wersja może mieć znaczenie dla działania skryptu?
5. Jaką rolę pełni apt w systemach opartych o Debian i dlaczego instalacja zależności nie powinna być wykonywana automatycznie bez wyraźnej zgody użytkownika?
6. Czym różni się zmienna środowiskowa od ustawienia zapisanego w pliku .env?
7. Jakie korzyści daje przeniesienie ustawień konfiguracyjnych do pliku KEY=VALUE?
8. Jakie problemy może powodować brak jednoznacznie określonego priorytetu źródeł konfiguracji?
9. Jaki priorytet źródeł konfiguracji jest najczytelniejszy w skryptach laboratoryjnych?
10. Dlaczego warto wydzielać funkcje pomocnicze do osobnego pliku biblioteki?
11. Czym różni się biblioteka funkcji dołączana przez source od zwykłego skryptu uruchamianego jako osobny proces?
12. Na czym polega autoweryfikacja środowiska i jakie elementy powinna sprawdzać?
13. Dlaczego raporty diagnostyczne i komunikaty o brakujących zależnościach powinny trafiać na STDERR?
14. W jaki sposób eliminować zależności niejawne, aby zwiększyć przenośność skryptu między maszynami?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Podsumowanie

W ramach laboratorium siódmego omówiono zagadnienia związane z integracją skryptów Bash z narzędziami zewnętrznymi oraz z kontrolowanym przygotowaniem środowiska uruchomieniowego. Przedstawiono pojęcie zależności skryptu i wskazano, że mogą one obejmować zarówno programy zewnętrzne, jak i pliki konfiguracyjne, katalogi robocze oraz zmienne środowiskowe. Utrwalono mechanizmy weryfikacji dostępności komend i ich wersji, a także podstawy pracy z menedżerem pakietów apt w systemach opartych o Debian. Omówiono rolę zmiennych środowiskowych i plików .env jako źródeł konfiguracji oraz znaczenie jednoznacznego określenia priorytetów nadpisywania ustawień. Zwrócono również uwagę na potrzebę budowania etapu autoweryfikacji środowiska przed wykonaniem logiki głównej skryptu. Rezultatem laboratorium jest umiejętność przygotowania skryptu, który jawnie określa zależności, potrafi raportować braki środowiskowe oraz działa w sposób bardziej przenośny i przewidywalny między różnymi maszynami tej samej rodziny systemowej.

Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:

Programowanie skryptowe

Zajęcia laboratoryjne nr 8

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści	
Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	3
Instrukcja laboratoryjna/ćwiczeniowa	3
Ćwiczenia samodzielne	10
Pytania pomocnicze	13
Podsumowanie	14
Literatura	14



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Laboratorium ósme ma na celu opanowanie technik diagnostycznych umożliwiających wykrywanie i usuwanie błędów w skryptach Bash. W ramach zajęć realizowane jest stosowanie trybów uruchomieniowych i przełączników debugowania (np. `bash -n`, `set -x`, `set -v`), analiza przebiegu wykonania, lokalizowanie błędów związanych z cytowaniem, zmiennymi i przepływem sterowania oraz wykorzystanie narzędzi wspierających jakość (np. `shellcheck`). Efektem ma być metodyczne podejście do identyfikacji przyczyn błędów i weryfikacji poprawek.

Cel ćwiczenia/laboratorium

- Rozpoznawanie i rozróżnianie podstawowych typów błędów występujących w skryptach Bash, w szczególności błędów składniowych, wykonawczych oraz logicznych.
- Opanowanie podstawowych technik diagnostycznych służących do analizy działania skryptu, w tym wykorzystania `bash -n`, `set -x`, `set -v` oraz interpretacji kodów zakończenia poleceń i skryptów.
- Stosowanie trybów zaostrej obsługi błędów, takich jak `set -u`, `set -e` oraz `set -o pipefail`, w celu szybszego wykrywania problemów związanych ze zmiennymi, poleceniami i potokami.
- Kształtowanie umiejętności metodycznego debugowania skryptów poprzez odtworzenie błędu, analizę jego przyczyny, dobranie właściwego narzędzia diagnostycznego oraz weryfikację poprawki.
- Wprowadzenie do korzystania z narzędzi wspomagających jakość skryptów, w szczególności `shellcheck`, oraz świadome rozdzielanie danych wynikowych od komunikatów diagnostycznych.
- Wypracowanie nawyku ponownego testowania i wykonywania prostych testów regresji po poprawie błędu, tak aby potwierdzić skuteczność naprawy i ograniczyć ryzyko wprowadzenia nowych nieprawidłowości.

Instrukcja laboratoryjna/ćwiczeniowa

Istota debugowania w skryptach Bash



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Debugowanie skryptu powłoki polega na metodycznym ustalaniu, dlaczego program nie działa zgodnie z oczekiwaniami. W praktyce nie wystarczy stwierdzenie, że skrypt „nie działa”. Należy ustalić, czy problem dotyczy składni, wykonania, czy logiki programu. Błąd składniowy uniemożliwia poprawne zinterpretowanie kodu przez powłokę. Błąd wykonania pojawia się wtedy, gdy składnia jest poprawna, lecz podczas działania występuje problem środowiskowy lub operacyjny, na przykład brak pliku, brak programu lub odwołanie do niezdefiniowanej zmiennej. Błąd logiczny jest najtrudniejszy do wykrycia, ponieważ skrypt uruchamia się poprawnie, ale zwraca niewłaściwy wynik.

W laboratorium stosowany jest powtarzalny schemat pracy:

1. Odtworzenie błędu na konkretnym wejściu.
2. Sprawdzenie składni przez `bash -n`.
3. Uruchomienie kontrolne i zapis `STDOUT` oraz `STDERR`.
4. Śledzenie wykonania przez `bash -x` lub `set -x`.
5. Analiza kodu zakończenia.
6. Uruchomienie ShellCheck, jeżeli problem dotyczy jakości kodu.
7. Wprowadzenie poprawki.
8. Retest na tym samym wejściu.
9. Porównanie wyniku z oczekiwanym rezultatem.

Kolejność ma znaczenie. Skrypt z błędem składniowym nie powinien być dalej analizowany jako przypadek logiczny. Najpierw należy doprowadzić go do poprawnej składni, a dopiero potem badać zachowanie podczas wykonania.

W pracy ze skryptami Bash istotne jest rozdzielenie diagnozowania od przypadkowego modyfikowania kodu. Najpierw należy odtworzyć błąd, następnie zawęzić jego źródło, dobrać odpowiednie narzędzie diagnostyczne, a dopiero potem wprowadzać poprawki. Dzięki temu proces naprawy staje się przewidywalny i możliwy do powtórzenia.

Sprawdzanie poprawności składni

Pierwszym krokiem przy analizie problemu powinno być sprawdzenie składni skryptu. W Bash służy do tego polecenie `bash -n`, które analizuje kod bez jego wykonywania.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Pozwala to wykryć typowe błędy, takie jak brak fi, brak done, niezamknięty cudzysłów, niepoprawny zapis warunku lub uszkodzoną strukturę bloku sterującego.

Zaletą bash -n jest to, że umożliwia bardzo szybkie odróżnienie błędów składniowych od błędów pojawiających się dopiero podczas wykonania. Jeżeli skrypt nie przechodzi tego etapu, dalsza analiza nie ma jeszcze sensu, ponieważ program nie jest poprawnym kodem Bash.

Przykład:

```
bash -n broken/broken_syntax.sh
```

Jeżeli składnia jest poprawna, polecenie zwykle nie wypisuje żadnego komunikatu i kończy się kodem 0. W przypadku błędu zwracany jest komunikat diagnostyczny wskazujący miejsce problemu.

Śledzenie przebiegu wykonania

Gdy składnia skryptu jest poprawna, kolejnym krokiem jest analiza przebiegu wykonania. Najważniejszym mechanizmem jest tu tryb śledzenia set -x lub równoważne uruchomienie skryptu przez bash -x. Mechanizm ten powoduje wypisywanie kolejnych poleceń po rozwinięciu podstawień, zmiennych i argumentów. Dzięki temu można zobaczyć, jakie wartości mają zmienne w momencie wykonania oraz które gałęzie warunków lub iteracje pętli zostały faktycznie uruchomione.

Tryb set -v działa inaczej: pokazuje linie skryptu w momencie ich odczytu, jeszcze przed pełnym rozwinięciem argumentów i podstawień. W praktyce set -v jest użyteczny przy analizie struktury wykonywanego kodu, natomiast set -x lepiej nadaje się do obserwacji rzeczywistego przebiegu programu.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykłady:

```
bash -x broken/broken_logic.sh
```

```
set -v
```

```
set -x
```

W bardziej uporządkowanych rozwiązaniach ślad debugowania powinien być zapisywany do pliku raportowego albo przekierowywany na standardowe wyjście błędów, tak aby nie mieszał się z właściwymi danymi programu.

Zaostrzona obsługa błędów

W Bash dostępne są przełączniki, które pomagają szybciej ujawniać problemy ukryte w kodzie. Szczególnie ważne są:

- `set -u` — wykrywanie odwołań do niezdefiniowanych zmiennych,
- `set -e` — przerywanie działania po błędzie polecenia,
- `set -o pipefail` — wykrywanie błędów wewnątrz potoków.

W praktyce często używa się łącznej postaci:

```
set -euo pipefail
```

Tryb ten nie naprawia błędów samodzielnie, ale powoduje, że wiele problemów staje się natychmiast widocznych. Jest to szczególnie ważne wtedy, gdy skrypt korzysta z wielu poleceń zewnętrznych, danych wejściowych lub zmiennych środowiskowych.

Należy jednak pamiętać, że tryb restrykcyjny wymaga bardziej świadomego pisania kodu. Programista musi wiedzieć, kiedy błąd powinien rzeczywiście przerwać wykonanie, a kiedy jest stanem dopuszczalnym i należy go obsłużyć jawnie.

ShellCheck jako narzędzie wspomagające jakość skryptów



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



ShellCheck jest narzędziem statycznej analizy kodu Bash. Pozwala wykrywać liczne typowe problemy jeszcze przed uruchomieniem programu, między innymi:

- brak cytowania zmiennych,
- nieużywane zmienne,
- ryzykowne konstrukcje warunkowe,
- nieczytelne lub niezalecane wzorce składniowe.

Przykład użycia:

```
shellcheck broken/problematic.sh
```

ShellCheck nie zastępuje testów i debugowania wykonania, ale stanowi bardzo ważny etap kontroli jakości. W praktyce dobrze jest traktować go jako narzędzie wstępnej weryfikacji, uruchamiane przed właściwą analizą działania skryptu.

Metodyka diagnozowania problemu

Poprawne debugowanie powinno mieć uporządkowaną postać. W typowym scenariuszu zaleca się następującą kolejność:

najpierw sprawdzenie składni, następnie reprodukcję błędu, potem uruchomienie śledzenia wykonania, analizę kodu zakończenia i dopiero w dalszej kolejności użycie narzędzi wspierających, takich jak ShellCheck. Po poprawieniu błędu należy ponownie uruchomić test i potwierdzić, że skrypt działa poprawnie oraz że poprawka nie wprowadziła nowych problemów.

Takie podejście jest szczególnie ważne w skryptach rozwijanych iteracyjnie, które z czasem stają się bardziej złożone i zaczynają zależeć od danych wejściowych, plików, programów zewnętrznych lub konfiguracji środowiskowej.

Przykłady praktyczne

Przykład 1. Sprawdzenie błędu składniowego

Poniższy skrypt zawiera celowo wprowadzony błąd składniowy:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
#!/bin/bash  
  
if [ "$1" = "test" ]; then  
  
    echo "OK"
```

Polecenie diagnostyczne:

```
bash -n broken/broken_syntax.sh
```

Wynik działania powinien wskazać, że konstrukcja warunkowa nie została poprawnie zamknięta. Po dopisaniu brakującego fi test należy wykonać ponownie.

Przykład 2. Śledzenie błędu logicznego

Poniższy fragment zawiera problem związany z niepoprawnym cytowaniem zmiennej:

```
#!/bin/bash  
  
file_path="/tmp/Moje pliki/dane.txt"  
  
cat $file_path
```

Uruchomienie z trybem śledzenia:

```
bash -x broken/broken_logic.sh
```

Na podstawie raportu można zauważyć, że ścieżka została rozbита na kilka argumentów. Wersja poprawna powinna używać postaci:

```
cat "$file_path"
```

Przykład 3. Porównanie set -v i set -x

Dla prostego skryptu można porównać oba tryby:

```
bash -v sample.sh
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
bash -x sample.sh
```

W pierwszym przypadku widoczne są kolejne linie kodu podczas odczytu, natomiast w drugim — polecenia po rozwinięciu zmiennych i podstawień. Dzięki temu można dobrać narzędzie diagnostyczne do rodzaju analizowanego problemu.

Przykład 4. Wykrywanie problemu przez `set -u`

Poniższy skrypt odwołuje się do niezdefiniowanej zmiennej:

```
#!/bin/bash  
  
echo "Wartość: $UNDEFINED_VAR"
```

Uruchomienie zwykłe:

```
bash broken/unset_var.sh
```

Uruchomienie z trybem restrykcyjnym:

```
bash -c 'set -u; source broken/unset_var.sh'
```

W drugim przypadku Bash natychmiast zgłosi problem związany z brakiem definicji zmiennej.

Przykład 5. Analiza kodu zakończenia

Przykładowe skrypty:

```
#!/bin/bash  
  
echo "Test poprawny"  
  
exit 0  
  
#!/bin/bash  
  
echo "Test błędny" >&2
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



exit 5

Uruchomienie:

```
./broken/ok.sh
```

```
echo $?
```

```
./broken/fail.sh
```

```
echo $?
```

W ten sposób można łatwo sprawdzić, że poprawne wykonanie kończy się kodem 0, natomiast błąd zwraca kod różny od zera.

Przykład 6. ShellCheck

Dla skryptu zawierającego typowe błędy:

```
#!/bin/bash  
  
name=$1  
  
if [ $name = admin ]; then  
    echo Hello $name  
  
fi
```

uruchomienie:

```
shellcheck broken/problematic.sh
```

powinno wskazać między innymi brak cytowania zmiennych. Na tej podstawie można poprawić kod jeszcze przed jego wykonaniem.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Ćwiczenia samodzielne

Dopuszcza się wstępną realizację zadań w terminalu, natomiast wymaganym rezultatem jest skrypt Bash automatyzujący rozwiązanie. W przypadku błędnego użycia skryptu, np. niewłaściwej liczby argumentów lub niepoprawnego formatu danych wejściowych, należy wypisać komunikat użycia na STDERR i zakończyć działanie kodem 2.

Organizacja katalogu pracy

Do laboratorium dołączony jest katalog startowy lab8_start/. Student pracuje na gotowych przypadkach błędów. Zadania polegają na diagnozie, poprawie i ponownym sprawdzeniu wskazanych przypadków. Nie należy tworzyć własnych błędnych skryptów w zadaniach obowiązkowych.

Zalecana struktura pracy:

```
lab8_start/  
├── cases.tsv  
├── bin/  
├── broken/  
├── fixed/  
├── data/  
├── expected/  
└── reports/
```

Katalog broken/ zawiera błędne skrypty. Katalog fixed/ służy do zapisywania poprawionych wersji. Katalog data/ zawiera dane testowe. Katalog expected/ zawiera oczekiwane wyniki. Katalog reports/ służy do zapisywania wyników diagnostyki.

Zadanie 1. Klasyfikacja przypadków błędów

Przeanalizuj plik cases.tsv oraz skrypty z katalogu broken/. Plik cases.tsv zawiera identyfikatory przypadków, nazwy skryptów oraz typ przypadku diagnostycznego wskazany przez prowadzącego. Przygotuj tabelę reports/error_cases.tsv w formacie: case_id, file, symptom, error_type, first_tool. Dopuszczalne typy błędów: SYNTAX, RUNTIME, LOGIC, QUALITY, UNKNOWN. Dopuszczalne narzędzia pierwszego wyboru: bash -n, bash -x, strict mode, shellcheck, manual run.

Zadanie 2. Kontrola składni przez bash -n

Utwórz skrypt bin/check_syntax_all.sh sprawdzający składnię wszystkich plików *.sh z katalogu broken/. Wynik zapisz w reports/syntax_report.tsv w formacie: file, status.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Status: OK albo SYNTAX_ERROR. Dla przypadków składniowych przygotuj poprawione wersje w fixed/ i wygeneruj reports/syntax_report_fixed.tsv.

Zadanie 3. Reprodukacja błędów wykonania

Dla przypadków wykonawczych uruchom wskazane skrypty na danych z katalogu data/. Za przypadki wykonawcze należy uznać wpisy oznaczone w cases.tsv jako RUNTIME. Zapisz STDOUT i STDERR do osobnych plików reports/<case_id>_stdout.txt oraz reports/<case_id>_stderr.txt. Przygotuj reports/runtime_reproduction.tsv w formacie: case_id, command, stdout_file, stderr_file, result. Wartości result: REPRODUCED albo NOT_REPRODUCED.

Zadanie 4. Śledzenie wykonania przez bash -x

Utwórz skrypt bin/trace_case.sh przyjmujący identyfikator przypadku i uruchamiający odpowiadający mu skrypt w trybie bash -x. Ślad wykonania zapisz do reports/<case_id>_trace.txt. Przygotuj reports/trace_summary.tsv w formacie: case_id, trace_file, observed_problem.

Zadanie 5. Diagnostyka błędów logicznych

Dla przypadków logicznych porównaj wynik rzeczywisty z wynikiem oczekiwanym z katalogu expected/. Przygotuj reports/logic_diagnosis.tsv w formacie: case_id, actual_output, expected_output, diagnosis. Następnie przygotuj poprawione wersje skryptów w katalogu fixed/.

Zadanie 6. Analiza trybu restrykcyjnego

Utwórz skrypt bin/strict_mode_check.sh uruchamiający wskazane przypadki w trybie standardowym oraz w trybie set -euo pipefail. Skrypt strict_mode_check.sh powinien analizować przypadki oznaczone w cases.tsv jako RUNTIME lub LOGIC, dla których tryb restrykcyjny może ujawnić dodatkowe problemy. Przygotuj reports/strict_mode_report.tsv w formacie: case_id, normal_result, strict_result, diagnosis. Raport ma wskazywać, które problemy ujawniają się dopiero w trybie restrykcyjnym.

Zadanie 7. Analiza ShellCheck



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Utwórz skrypt `bin/shellcheck_all.sh` uruchamiający ShellCheck dla skryptów z katalogu `broken/`. Raport zbiorczy zapisz w `reports/shellcheck_report.tsv` w formacie: `file, status, issues_count`. Status: OK, WARN albo NOT_AVAILABLE. Pełne wyniki zapisz jako `reports/shellcheck_<nazwa_pliku>.txt`.

Zadanie 8. Raport diagnostyczny dla pojedynczego przypadku

Utwórz skrypt `bin/debug_case.sh` przyjmujący identyfikator przypadku i generujący `reports/debug_case_<case_id>.txt`. Raport ma zawierać: `case_id, file, syntax_status, runtime_status, trace_file, shellcheck_status, error_type, probable_cause, fixed_file, retest_status`.

Zadanie 9. Retest poprawionych skryptów

Utwórz skrypt `bin/retest_fixed.sh` uruchamiający poprawione wersje z katalogu `fixed/` i porównujący ich wyniki z plikami w `expected/`. Przygotuj `reports/retest_report.tsv` w formacie: `case_id, fixed_file, expected_file, result`. Wartości result: PASS, FAIL, MISSING_FIXED, MISSING_EXPECTED.

Zadanie 10. Końcowe podsumowanie diagnostyki

Przygotuj raport `reports/lab8_debug_summary.md`. Raport ma zawierać sekcje: Syntax errors, Runtime errors, Logic errors, ShellCheck findings, Fixed scripts, Retest results, Conclusions. Raport powinien odwoływać się do plików utworzonych w poprzednich zadaniach.

Zadanie dodatkowe A. Selektowne śledzenie wykonania

Przygotuj skrypt `bin/selective_trace_demo.sh` pokazujący użycie `set -x` tylko dla wybranego fragmentu kodu. Wynik zapisz w `reports/selective_trace.txt`. Porównaj go ze śladem uzyskanym przez uruchomienie całego skryptu przez `bash -x`.

Zadanie dodatkowe B. Prosty test regresji

Dla jednego poprawionego przypadku przygotuj skrypt `bin/regression_check.sh` uruchamiający co najmniej dwa testy wejścia/wyjścia. Wynik zapisz w `reports/regression_report.tsv` w formacie: `test_id, command, expected_file, actual_file, result`.

Zadanie dodatkowe C. Porównanie `set -v` i `set -x`

Dla jednego wskazanego skryptu porównaj wynik uruchomienia z `set -v` oraz `set -x`. Przygotuj `reports/modes_comparison.txt` z dwiema sekcjami: `set -v` oraz `set -x`, a następnie krótko wyjaśnij różnicę między trybami.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Pytania pomocnicze

1. Czym różni się błąd składniowy od błędu wykonania oraz od błędu logicznego w skrypcie Bash?
2. Do czego służy polecenie `bash -n` i jakie problemy można nim wykryć?
3. Jaka jest różnica między `set -v` a `set -x`?
4. W jakich sytuacjach użycie `set -x` jest szczególnie przydatne?
5. Co oznacza opcja `set -u` i dlaczego pomaga wykrywać błędy związane ze zmiennymi?
6. Jakie znaczenie mają `set -e` oraz `set -o pipefail` w kontekście diagnostyki?
7. Dlaczego poprawne cytowanie zmiennych jest tak ważne podczas debugowania skryptów?
8. W jaki sposób można odczytać kod zakończenia ostatniego polecenia i dlaczego ma to znaczenie diagnostyczne?
9. Jaką rolę pełni `shellcheck` i jakie typy problemów potrafi wykrywać?
10. Dlaczego analiza błędu powinna mieć charakter metodyczny, a nie polegać na przypadkowym modyfikowaniu skryptu?
11. W jaki sposób rozdzielać dane wynikowe od diagnostyki podczas uruchamiania skryptów?
12. Na czym polega reprodukcja błędu i dlaczego jest pierwszym krokiem poprawnego debugowania?
13. Dlaczego po naprawie błędu warto wykonać ponowną weryfikację i test regresji?
14. W jakich przypadkach selektywne włączanie `set -x` jest lepsze niż śledzenie całego skryptu?
15. Jakie są typowe symptomy błędów związanych z niepoprawnym użyciem "\$@", testów warunkowych lub porównań liczbowych?

Podsumowanie

W ramach laboratorium ósmego omówiono metodyczne podejście do diagnozowania i usuwania błędów w skryptach Bash. Przedstawiono różnice między błędami



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

składniowymi, wykonawczymi i logicznymi oraz pokazano, że każdy z tych typów wymaga nieco innej metody analizy. Utrwalono zastosowanie podstawowych mechanizmów debugowania, takich jak `bash -n`, `set -v`, `set -x`, a także trybów zaostrożonej obsługi błędów opartych na `set -u`, `set -e` oraz `pipefail`. Wprowadzono także narzędzie `shellcheck` jako element wspomagający jakość skryptów i pozwalający wykrywać typowe problemy jeszcze przed uruchomieniem kodu. Rezultatem laboratorium jest umiejętność prowadzenia uporządkowanej analizy błędów: od odtworzenia problemu, przez wskazanie przyczyny, aż po weryfikację poprawki i potwierdzenie, że naprawa nie wprowadziła nowych nieprawidłowości.

Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Materiały do przedmiotu:

Programowanie skryptowe

Zajęcia laboratoryjne nr 9

Autor:

Albert Rachwał



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Spis treści

Wprowadzenie, tematyka zajęć.	3
Cel ćwiczenia/laboratorium	4
Instrukcja laboratoryjna/ćwiczeniowa	6
Ćwiczenia samodzielne	22
Pytania pomocnicze	40
Podsumowanie	42
Literatura	43



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Laboratorium dziewiąte realizuje temat „Obsługa błędów i raportowanie problemów” w kontekście skryptów operacyjnych uruchamianych bez nadzoru. Dotychczasowe laboratoria wprowadzały podstawowe mechanizmy potrzebne do tworzenia skryptów Bash: uruchamianie skryptów, argumenty, walidację, funkcje, pracę z plikami, potoki, analizę tekstu, konfigurację oraz sprawdzanie zależności środowiskowych. W tym laboratorium mechanizmy te są traktowane jako wymagany fundament i zostają użyte do zbudowania skryptu, który wykonuje rzeczywiste zadanie administracyjne w sposób kontrolowany, powtarzalny i odporny na błędy.

Skrypt operacyjny różni się od prostego skryptu demonstracyjnego. Nie zakłada, że użytkownik obserwuje każdy etap działania i ręcznie reaguje na problemy. Może być uruchamiany cyklicznie, na przykład przez cron, albo jako element większego procesu automatyzacji. Z tego powodu musi działać bez interakcji z użytkownikiem, korzystać z jednoznacznej konfiguracji, stosować przewidywalne kody zakończenia, zapisywać log, generować raport końcowy, obsługiwać błędy poleceń zewnętrznych oraz sprzątać zasoby tymczasowe niezależnie od sposobu zakończenia pracy.

Głównym scenariuszem laboratorium jest przygotowanie skryptu operacyjnego wykonującego zadanie utrzymaniowe, takie jak archiwizacja lub backup katalogu projektu. Proces ten obejmuje wczytanie konfiguracji, wykonanie preflightu, zabezpieczenie przed równoległym uruchomieniem, sprawdzenie zależnej usługi z użyciem timeout i retry, utworzenie katalogu tymczasowego, wykonanie właściwej operacji, weryfikację artefaktu, wygenerowanie manifestu, zapis logu oraz przygotowanie raportu końcowego.

Szczególny nacisk położony jest na sytuacje awaryjne. Skrypt może napotkać brak konfiguracji, niedostępne narzędzie, aktywną blokadę poprzedniego uruchomienia, niepowodzenie polecenia zewnętrznego, przekroczenie czasu oczekiwania, chwilową niedostępność usługi albo przerwanie przez użytkownika. Każdy z tych przypadków powinien być obsługiwany w sposób jednoznaczny: komunikatem diagnostycznym, wpisem w logu, odpowiednim kodem zakończenia oraz — jeżeli to konieczne — procedurą sprzątającą.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W laboratorium wykorzystywane są mechanizmy typowe dla praktycznych skryptów administracyjnych i DevOps: konfiguracja w pliku .env lub KEY=VALUE, flock albo lockfile do ochrony przed równoległym uruchomieniem, timeout do ograniczenia czasu oczekiwania, funkcja retry do ponawiania chwilowo nieudanych operacji, funkcja run_checked do kontrolowanego uruchamiania poleceń zewnętrznych, mktemp do tworzenia zasobów tymczasowych, trap do sprzątania przy zakończeniu lub przerwaniu pracy, atomowy zapis wyniku, logowanie ze znacznikiem czasu, pomocniczy skrypt Pythona do przetwarzania danych oraz testy smoke.

Ważnym elementem jest atomowy zapis artefaktu wynikowego. Plik backupu, manifest albo raport końcowy nie powinny być zapisywane od razu jako finalny wynik, jeżeli proces może zakończyć się w połowie. Najpierw powinny powstać w lokalizacji tymczasowej lub pod nazwą roboczą, a dopiero po poprawnym zakończeniu wszystkich wymaganych etapów powinny zostać przeniesione do lokalizacji docelowej. Dzięki temu katalog wynikowy nie zawiera częściowo zapisanych plików, które mogłyby zostać błędnie uznane za poprawny rezultat.

Efektem laboratorium ma być skrypt klasy roboczej: bezinterakcyjny, konfigurowalny, zabezpieczony przed równoległym uruchomieniem, odporny na podstawowe awarie, zapisujący log i raport, sprzątający zasoby tymczasowe oraz możliwy do uruchamiania ręcznego albo cyklicznego. Laboratorium domyka cykl zajęć, pokazując przejście od pojedynczych mechanizmów języka Bash do narzędzia, które zachowuje się jak element rzeczywistego procesu operacyjnego.

Cel ćwiczenia/laboratorium

Po wykonaniu laboratorium student powinien umieć:

- zaprojektować bezinterakcyjny skrypt operacyjny wykonujący wieloetapowe zadanie administracyjne lub utrzymaniowe;
- przygotować strukturę katalogów procesu operacyjnego obejmującą konfigurację, dane wejściowe, wyniki, pliki tymczasowe, logi, raporty i testy;
- wczytać konfigurację z pliku .env lub pliku w formacie KEY=VALUE oraz sprawdzić kompletność wymaganych ustawień;
- przygotować tryb --dry-run, który pokazuje plan działania bez tworzenia finalnych artefaktów;



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- wykonać preflight narzędzi, katalogów i usług wymaganych przez konkretny proces operacyjny;
- sprawdzać dostępność narzędzi zewnętrznych wymaganych w laboratorium, takich jak tar, gzip, find, du, date, sha256sum, mktemp, flock, timeout, curl i python3;
- stosować flock albo mechanizm lockfile do ochrony przed równoległym uruchomieniem tego samego skryptu;
- stosować timeout do ograniczenia maksymalnego czasu oczekiwania na polecenie zewnętrzne lub odpowiedź usługi;
- zaimplementować funkcję retry, która ponawia chwilowo nieudaną operację określoną liczbę razy;
- przygotować funkcję run_checked, która uruchamia polecenie zewnętrzne, zapisuje jego kod zakończenia i raportuje błąd w kontrolowany sposób;
- stosować mktemp do bezpiecznego tworzenia katalogów lub plików tymczasowych;
- stosować trap do uruchamiania procedury sprzątającej przy zakończeniu działania skryptu;
- obsługiwać sygnały SIGINT i SIGTERM, usuwać zasoby tymczasowe i kończyć działanie przewidywalnym kodem, w szczególności kodem 130 po przerwaniu przez użytkownika;
- projektować funkcję cleanup, która usuwa zasoby tymczasowe, ale nie kasuje poprawnie utworzonych wyników końcowych;
- tworzyć artefakt wynikowy, na przykład backup katalogu, w sposób atomowy: najpierw jako wynik tymczasowy, a dopiero po sukcesie jako plik finalny;
- wygenerować manifest artefaktu zawierający nazwę pliku, rozmiar, liczbę elementów, sumę kontrolną i status weryfikacji;
- sprawdzić poprawność utworzonego archiwum za pomocą tar -tzf oraz sha256sum;
- zapisywać przebieg działania skryptu do pliku logu z użyciem znaczników czasu;
- rozróżniać w logu komunikaty informacyjne, ostrzeżenia i błędy;
- przygotować raport końcowy operacji zawierający status wykonania, ścieżki do artefaktów, rozmiary, sumy kontrolne, czas wykonania, lokalizację logu i komunikat końcowy;
- stosować jawny kontrakt kodów wyjścia, rozróżniający między innymi sukces, błędne użycie, brak konfiguracji, błąd preflightu, aktywny lock, błąd operacji zewnętrznej, błąd raportowania oraz przerwanie sygnałem;



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- wykorzystać pomocniczy skrypt Pythona z argparse do przetworzenia manifestu lub raportu wygenerowanego przez Bash;
- przygotować testy smoke sprawdzające najważniejsze tryby działania skryptu;
- przygotować przykładowy wpis cron uruchamiający skrypt cyklicznie z przekierowaniem logów;
- wyjaśnić różnicę między skryptem uruchamianym ręcznie a skryptem operacyjnym przeznaczonym do pracy bez nadzoru.

Instrukcja laboratoryjna/ćwiczeniowa

Od obsługi błędów do skryptu operacyjnego

Skrypt operacyjny to narzędzie przeznaczone do wykonania konkretnego zadania utrzymaniowego bez stałego nadzoru użytkownika. Może tworzyć artefakty, przetwarzać katalogi, sprawdzać dostępność usług, generować raporty, archiwizować dane albo przygotowywać wyniki dla innych narzędzi. Z tego powodu musi kontrolować nie tylko dane wejściowe, ale również cały przebieg pracy: konfigurację, zależności, blokady, zasoby tymczasowe, operacje zewnętrzne, wynik końcowy i raportowanie.

W części zadaniowej ten model zostanie zastosowany do konkretnego scenariusza operacyjnego, czyli skryptu przygotowującego i weryfikującego artefakt wynikowy. Sam scenariusz jest przykładem. Najważniejszym celem laboratorium nie jest wykonanie jednego rodzaju operacji, lecz opanowanie wzorca projektowania skryptu, który może działać bez nadzoru i nie pozostawia środowiska w stanie przypadkowym.

Najważniejsza zasada jest następująca: skrypt nie ma działać wyłącznie w przypadku idealnym. Ma również bezpiecznie zakończyć pracę po błędzie, przerwaniu albo ponownym uruchomieniu. Dopiero taki skrypt można traktować jako narzędzie operacyjne.

Cykl życia skryptu operacyjnego

Skrypt operacyjny powinien mieć jasno określoną kolejność działania. Nie należy zaczynać od właściwej operacji, jeżeli wcześniej nie zostały sprawdzone argumenty, konfiguracja, wymagania środowiska i warunki bezpiecznego wykonania. Taki porządek jest szczególnie ważny w skryptach uruchamianych bez nadzoru, ponieważ po starcie skryptu nie ma użytkownika, który ręcznie poprawi brakujący katalog, zatrzyma drugą instancję procesu albo usunie częściowy wynik.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W laboratorium przyjmujemy następujący ogólny cykl życia skryptu operacyjnego:

1. Odczyt argumentów uruchomieniowych
2. Wczytanie konfiguracji
3. Preflight procesu operacyjnego
4. Założenie blokady równoległego uruchomienia
5. Sprawdzenie zależnej usługi lub zasobu
6. Utworzenie katalogu tymczasowego
7. Wykonanie właściwej operacji
8. Weryfikacja artefaktu wynikowego
9. Wygenerowanie manifestu
10. Uruchomienie pomocniczego skryptu Pythona
11. Utworzenie raportu końcowego
12. Sprzątanie zasobów tymczasowych

Pierwsze etapy decydują, czy skrypt może rozpocząć właściwą pracę. Skrypt powinien sprawdzić argumenty, wczytać plik konfiguracyjny, potwierdzić obecność wymaganych narzędzi i katalogów, a następnie założyć blokadę chroniącą przed równoległym uruchomieniem. Jeżeli którykolwiek z tych etapów zakończy się błędem, właściwa operacja nie powinna zostać wykonana.

Dopiero po przejściu etapów kontrolnych skrypt wykonuje operacje robocze. W zależności od scenariusza może to być archiwizacja katalogu, przetworzenie danych, pobranie raportu, przygotowanie paczki wynikowej albo inna operacja utrzymaniowa. W zadaniach laboratoryjnych model ten zostanie zastosowany do skryptu `ops_backup.sh`, który tworzy i weryfikuje archiwum katalogu projektu.

Końcowe etapy odpowiadają za weryfikację, raportowanie i sprzątanie. Wynik nie powinien być uznany za poprawny tylko dlatego, że powstał plik. Skrypt powinien sprawdzić artefakt, zapisać manifest, utworzyć raport końcowy, odnotować wynik w logu i usunąć zasoby tymczasowe. Sprzątanie powinno działać zarówno po sukcesie, jak i po błędzie lub przerwaniu działania.

Minimalny szkielet logiczny skryptu może wyglądać następująco:

```
main() {  
  parse_args "$@"  
  load_config  
  preflight  
  acquire_lock  
  setup_traps  
  check_dependencies_or_service  
  create_tmp_workspace
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
run_operation
verify_artifact
generate_manifest
run_python_summary
generate_final_report
}
```

Ten schemat nie jest gotową implementacją. Pokazuje kolejność odpowiedzialności. Każda funkcja powinna realizować jeden etap procesu i zwracać jednoznaczny kod zakończenia, aby skrypt główny mógł zdecydować, czy kontynuować pracę, czy przejść do obsługi błędu.

Dobłą praktyką jest unikanie przypadkowego wywoływania exit głęboko w funkcjach pomocniczych. Funkcje takie jak preflight, acquire_lock albo run_checked powinny przede wszystkim zwracać kod zakończenia, natomiast decyzja o zakończeniu całego programu powinna być podejmowana w main. Dzięki temu przepływ sterowania jest czytelniejszy, a funkcje łatwiej testować.

Struktura katalogów procesu operacyjnego

Skrypt operacyjny powinien pracować w uporządkowanej strukturze katalogów. Oddzielenie konfiguracji, danych wejściowych, wyników, plików tymczasowych, logów i raportów ułatwia kontrolę procesu oraz ogranicza ryzyko przypadkowego nadpisania danych.

W laboratorium przyjmujemy następującą strukturę katalogów:

```
lab9/
├── bin/
├── lib/
├── config/
├── input/project_data/
├── output/
├── tmp/
├── reports/
├── logs/
├── python/
├── tests/
└── cron/
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Katalog `bin/` zawiera główny skrypt uruchomieniowy. Katalog `lib/` przechowuje funkcje pomocnicze, na przykład logowanie, obsługę ponowień i kontrolowane uruchamianie poleceń zewnętrznych. Katalog `config/` zawiera plik konfiguracyjny procesu. Katalog `input/project_data/` przechowuje dane wejściowe, na których wykonywana jest operacja.

Katalog `output/` powinien zawierać wyłącznie poprawnie utworzone artefakty końcowe. Katalog `tmp/` jest przeznaczony na zasoby robocze i pliki tymczasowe. Katalog `reports/` zawiera manifesty, raporty cząstkowe i raport końcowy. Katalog `logs/` przechowuje log operacyjny. Katalog `python/` zawiera pomocniczy skrypt Pythona, `tests/` testy smoke, a `cron/` przykładowy wpis harmonogramu.

Podstawowa zasada jest następująca: pliki robocze powstają w `tmp/`, a do `output/` trafiają dopiero wyniki zweryfikowane. Dzięki temu katalog wynikowy nie zawiera plików częściowych, testowych ani uszkodzonych.

Konfiguracja i bezinterakcyjność

Skrypt operacyjny przeznaczony do pracy bez nadzoru nie powinien pytać użytkownika o dane w trakcie działania. Nie należy opierać takiego procesu na `read`, potwierdzeniach z klawiatury ani ręcznym wyborze ścieżek. Wszystkie parametry wymagane do pracy powinny pochodzić z argumentów uruchomieniowych albo z pliku konfiguracyjnego.

W laboratorium główny skrypt jest uruchamiany ze wskazaniem pliku konfiguracyjnego:

```
bin/ops_backup.sh --config config/ops.env
```

Przykładowy plik `config/ops.env` może zawierać:

```
PROJECT_ROOT=/home/student/lab9
```

```
SOURCE_DIR=/home/student/lab9/input/project_data
```

```
OUTPUT_DIR=/home/student/lab9/output
```

```
TMP_DIR=/home/student/lab9/tmp
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
REPORT_DIR=/home/student/lab9/reports  
LOG_DIR=/home/student/lab9/logs  
LOG_FILE=/home/student/lab9/logs/ops_backup.log  
BACKUP_PREFIX=project_backup  
SERVICE_URL=http://127.0.0.1:8000/health  
LOCK_FILE=/home/student/lab9/tmp/ops_backup.lock
```

W konfiguracji zaleca się stosowanie ścieżek bezwzględnych. Jest to szczególnie ważne przy uruchamianiu skryptu przez cron, ponieważ harmonogram zadań nie musi uruchamiać procesu z katalogu projektu. Ścieżki względne są dopuszczalne podczas pracy ręcznej, ale wtedy skrypt powinien jednoznacznie ustalać katalog roboczy albo wymagać uruchomienia z katalogu lab9/.

Skrypt powinien sprawdzić, czy plik konfiguracyjny istnieje oraz czy zawiera wszystkie wymagane zmienne. Brak konfiguracji jest błędem krytycznym, ponieważ bez niej proces nie ma jednoznacznie określonych katalogów wejściowych, wyjściowych, logów, raportów i blokady.

Uwaga: wczytywanie konfiguracji przez source config/ops.env oznacza wykonanie zawartości tego pliku przez powłokę Bash. Dlatego taki sposób jest dopuszczalny tylko wtedy, gdy plik konfiguracyjny jest zaufany i kontrolowany przez autora skryptu. W bardziej restrykcyjnym wariacie skrypt powinien parsować wyłącznie dozwolone wpisy KEY=VALUE i ignorować lub odrzucać inne konstrukcje. Niezależnie od wybranego wariantu należy sprawdzać obecność wymaganych zmiennych oraz nie przyjmować pustych wartości dla katalogów, pliku logu, pliku blokady i prefiksu backupu.

Tryb --dry-run powinien korzystać z tej samej konfiguracji, ale nie wykonywać operacji zmieniających stan katalogu output/. Jego celem jest pokazanie planu działania: źródła danych, katalogu wynikowego, liczby plików, rozmiaru danych i adresu usługi zależnej.

Kontrakt kodów wyjścia

Skrypt operacyjny powinien zwracać kody zakończenia zgodne z typem sytuacji, która wystąpiła. Kod wyjścia jest ważny nie tylko dla użytkownika uruchamiającego skrypt ręcznie, ale także dla testów smoke, harmonogramu cron i innych narzędzi automatyzacji.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W laboratorium należy przyjąć jawny kontrakt kodów wyjścia, na przykład:

```
EXIT_OK=0
EXIT_RUNTIME_ERROR=1
EXIT_USAGE=2
EXIT_CONFIG=3
EXIT_PREFLIGHT=4
EXIT_LOCKED=10
EXIT_EXTERNAL=20
EXIT_REPORT=30

EXIT_SIGINT=130
EXIT_SIGTERM=143
```

Kod 0 oznacza poprawne wykonanie. Kod 2 oznacza błędne użycie skryptu, na przykład brak wymaganego parametru --config. Kod 3 oznacza problem z konfiguracją. Kod 4 oznacza niepowodzenie etapu preflight. Kod 10 oznacza, że skrypt nie wystartował, ponieważ aktywna jest blokada poprzedniego uruchomienia.

Kod 20 oznacza błąd polecenia zewnętrznego, na przykład niepowodzenie tar, curl albo sha256sum. Kod 30 oznacza problem z utworzeniem raportu końcowego. Kod 130 oznacza kontrolowane przerwanie działania, na przykład po Ctrl+C.

W kodzie skryptu należy używać nazwanych stałych, a nie przypadkowych wartości exit 1. Dzięki temu zachowanie skryptu jest czytelne, a testy mogą jednoznacznie sprawdzać konkretne przypadki awaryjne.

Preflight procesu operacyjnego

Preflight to etap wykonywany przed operacją główną. Jego zadaniem jest sprawdzenie, czy proces może zostać bezpiecznie uruchomiony. W tym laboratorium preflight nie jest ogólną diagnostyką systemu, lecz kontrolą warunków wymaganych przez konkretny job operacyjny.

Minimalny zakres preflightu obejmuje sprawdzenie narzędzi zewnętrznych. Należy również sprawdzić katalogi i uprawnienia procesu: czy istnieje SOURCE_DIR, czy jest katalogiem, czy można utworzyć katalogi OUTPUT_DIR, TMP_DIR, REPORT_DIR i LOG_DIR, oraz czy możliwy jest zapis do pliku logu.

Wynik preflightu powinien zostać zapisany do raportu, na przykład:

reports/preflight_report.txt



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykładowy format:

```
OK: tar
OK: gzip
OK: find
OK: SOURCE_DIR=input/project_data
OK: OUTPUT_DIR=output
OK: TMP_DIR=tmp
OK: LOG_FILE=logs/ops_backup.log
```

Jeżeli brakuje narzędzia krytycznego albo katalog źródłowy jest niedostępny, skrypt powinien zakończyć działanie przed wykonaniem operacji głównej. W takiej sytuacji właściwa operacja nie powinna zostać rozpoczęta, ponieważ mogłaby zostawić częściowy wynik albo błędny raport.

Logowanie operacyjne

Skrypt uruchamiany bez nadzoru musi zapisywać przebieg działania do pliku logu. Logowanie do pliku jest możliwe dopiero po poprawnym wczytaniu konfiguracji, ponieważ ścieżka do pliku logu pochodzi ze zmiennej LOG_FILE. Błędy występujące wcześniej, na przykład brak argumentu --config, nieistniejący plik konfiguracyjny albo brak zmiennej LOG_FILE, powinny być wypisywane na STDERR. Nie należy próbować zapisywać ich do logu, którego lokalizacja nie została jeszcze poprawnie ustalona.

Log pozwala odtworzyć, kiedy proces wystartował, które etapy zostały wykonane, gdzie wystąpił błąd i z jakim statusem skrypt zakończył działanie.

W laboratorium należy przygotować funkcje logujące, na przykład w pliku:

lib/logging.sh

Minimalny zestaw funkcji:

```
log_info
log_warn
log_error
```

Wpisy w logu powinny zawierać znacznik czasu i poziom komunikatu:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

[2026-05-15 12:00:00] INFO Start joba
[2026-05-15 12:00:01] WARN Ponawiam próbę po błędzie
[2026-05-15 12:00:03] ERROR Nie udało się utworzyć artefaktu

Komunikaty błędów powinny trafiać zarówno do pliku logu, jak i na STDERR. Dzięki temu błąd jest widoczny przy ręcznym uruchomieniu, ale pozostaje również zapisany dla późniejszej analizy.

Log nie zastępuje raportu końcowego. Log opisuje przebieg procesu krok po kroku, natomiast raport końcowy podsumowuje wynik operacji: status, artefakt, manifest, sumę kontrolną, lokalizację logu i komunikat końcowy.

run_checked jako kontrola poleceń zewnętrznych

Skrypt operacyjny zwykle korzysta z wielu narzędzi zewnętrznych, takich jak tar, curl, sha256sum, mkdir, mv albo python3. Samo uruchomienie polecenia nie oznacza jeszcze, że operacja zakończyła się sukcesem. Każde istotne polecenie powinno być kontrolowane przez kod zakończenia.

W laboratorium należy zastosować funkcję run_checked, której zadaniem jest uruchomienie polecenia, zapisanie informacji o wykonywanej operacji i zgłoszenie błędu w przypadku niepowodzenia. Funkcja powinna zwracać kod zakończenia polecenia, aby skrypt główny mógł zdecydować, czy kontynuować pracę, czy zakończyć proces.

Przykładowe użycie:

```
run_checked "Utworzenie katalogu output" mkdir -p "$OUTPUT_DIR"  
run_checked "Sprawdzenie archiwum" tar -tzf "$backup_file"  
run_checked "Obliczenie sumy kontrolnej" sha256sum "$backup_file"
```

Funkcja run_checked nie powinna ukrywać błędów. Jeżeli polecenie zewnętrzne zakończy się niepowodzeniem, informacja o tym powinna trafić do logu, a w razie potrzeby również na STDERR. Dzięki temu wszystkie operacje krytyczne są obsługiwane w jednolity sposób.

Minimalny wzorzec funkcji może wyglądać następująco:

```
run_checked() {  
    local description="$1"  
    shift
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
log_info "$description"
```

```
"$@"
```

```
local status=$?
```

```
if [ "$status" -ne 0 ]; then
```

```
    log_error "$description zakończono błędem, kod: $status"
```

```
    return "$status"
```

```
fi
```

```
return 0
```

```
}
```

Takie rozwiązanie ogranicza powtarzanie tego samego schematu po każdym poleceniu zewnętrznym i ułatwia późniejsze rozszerzenie logowania albo raportowania.

Blokada równoległego uruchomienia

Skrypt uruchamiany cyklicznie może zostać wystartowany ponownie, zanim poprzednia instancja zakończy pracę. Taka sytuacja jest ryzykowna, ponieważ dwa procesy mogą jednocześnie modyfikować te same katalogi, tworzyć pliki o podobnych nazwach albo nadpisywać raporty i logi.

Do ochrony przed równoległym uruchomieniem stosuje się blokadę, na przykład mechanizm flock. Jeżeli blokada jest już zajęta, skrypt powinien zakończyć działanie kontrolowanym kodem, zamiast rozpoczynać drugą kopię procesu.

Przykładowy wzorzec:

```
acquire_lock() {
```

```
    exec 9>"$LOCK_FILE"
```

```
    if ! flock -n 9; then
```

```
        log_error "Inna instancja skryptu już działa"
```

```
        return "$EXIT_LOCKED"
```

```
    fi
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
return 0  
}
```

W funkcji main wynik można obsłużyć następująco:

```
acquire_lock || exit "$?"
```

Aktywny lock nie oznacza awarii danych ani błędu programu. Oznacza, że skrypt poprawnie wykrył trwające uruchomienie i odmówił startu w celu ochrony procesu. Jest to szczególnie istotne przy uruchamianiu z cron.

Plik blokady powinien znajdować się w lokalizacji kontrolowanej przez konfigurację, na przykład:

```
LOCK_FILE=/home/student/lab9/tmp/ops_backup.lock
```

Dzięki temu skrypt nie musi zakładać stałej ścieżki systemowej, a testy mogą łatwo sprawdzać scenariusz aktywnej blokady.

Odporność na chwilowe awarie z wykorzystaniem timeout i retry

Skrypt bez nadzoru nie powinien czekać w nieskończoność na odpowiedź usługi albo zakończenie polecenia zewnętrznego. Każda operacja zależna od zewnętrznego zasobu powinna mieć ograniczony czas wykonania. Do tego służy timeout.

Przykład:

```
timeout 5 curl -fsS "$SERVICE_URL"
```

Jeżeli problem może być chwilowy, można zastosować ponowienie operacji. Funkcja retry powinna wykonać polecenie określoną liczbę razy, odczekać między próbami i zwrócić błąd dopiero po wyczerpaniu limitu prób.

Przykład użycia:

```
retry 3 2 timeout 5 curl -fsS "$SERVICE_URL"
```

W tym przykładzie skrypt wykonuje maksymalnie trzy próby. Między próbami czeka dwie sekundy, a każda pojedyncza próba ma limit pięciu sekund.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Minimalny wzorzec funkcji:

```
retry() {  
    local attempts="$1"  
    local delay="$2"  
    shift 2  
  
    local i  
    for ((i = 1; i <= attempts; i++)); do  
        if "$@"; then  
            return 0  
        fi  
  
        log_warn "Próba $i/$attempts nie powiodła się"  
  
        if [ "$i" -lt "$attempts" ]; then  
            sleep "$delay"  
        fi  
    done  
  
    return 1  
}
```

Nie każdą operację należy ponawiać. Sensowne jest ponawianie chwilowej niedostępności usługi HTTP, krótkiego problemu sieciowego albo tymczasowego błędu zasobu. Nie należy ponawiać błędnego użycia skryptu, braku konfiguracji, braku narzędzia ani niepoprawnej ścieżki wejściowej.

Zasoby tymczasowe, mktemp i trap

Skrypt operacyjny nie powinien tworzyć plików tymczasowych pod ręcznie wymyślonymi nazwami, takimi jak tmp/file.txt albo tmp/backup.tmp. Takie nazwy mogą wejść w konflikt z poprzednim uruchomieniem albo z drugim procesem. Bezpieczniejszym rozwiązaniem jest użycie mktemp.

Przykład utworzenia katalogu tymczasowego wewnątrz katalogu wskazanego w konfiguracji:

```
tmp_workspace="$(mktemp -d "$TMP_DIR/ops_job.XXXXXX")"
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wszystkie pliki robocze powinny powstawać w tym katalogu. Dotyczy to między innymi częściowych archiwów, tymczasowych raportów, manifestów roboczych i danych pomocniczych.

Do sprzątania zasobów tymczasowych należy użyć funkcji cleanup oraz mechanizmu trap:

```
cleanup() {  
    if [ -n "${tmp_workspace:-}" ] && [ -d "$tmp_workspace" ]; then  
        rm -rf "$tmp_workspace"  
    fi  
}
```

```
trap cleanup EXIT
```

Funkcja cleanup powinna usuwać tylko zasoby tymczasowe. Nie powinna usuwać poprawnie utworzonych artefaktów końcowych z katalogu output/ ani raportów zapisanych w reports/.

Mechanizm trap cleanup EXIT powoduje, że procedura porządkująca zostanie wykonana przy zakończeniu skryptu. Dzięki temu katalog tymczasowy nie pozostaje w systemie po poprawnym wykonaniu, błędzie lub wcześniejszym zakończeniu pracy.

Obsługa SIGINT i SIGTERM

Skrypt operacyjny może zostać przerwany przed zakończeniem pracy. Najczęstszy przypadek to SIGINT, wysyłany po naciśnięciu Ctrl+C. Innym typowym sygnałem jest SIGTERM, używany przez system albo proces nadrzędny do zakończenia działania programu. Skrypt uruchamiany bez nadzoru powinien obsłużyć takie przerwanie w sposób kontrolowany.

Przerwanie nie powinno zostawiać procesu w stanie nieokreślonym. Jeżeli skrypt utworzył katalog tymczasowy, plik częściowy albo raport roboczy, powinien je usunąć albo oznaczyć jako niedokończone. Jeżeli operacja została przerwana przed weryfikacją artefaktu, taki artefakt nie powinien zostać przeniesiony do katalogu finalnego.

W praktyce procedura obsługi przerwania powinna wykonać następujące kroki:

1. zapisać informację o przerwaniu do logu,
2. ustawić status operacji na INTERRUPTED,
3. utworzyć raport końcowy, jeżeli jest to możliwe,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



4. usunąć zasoby tymczasowe,
5. zakończyć działanie kodem 130.

Przykładowy wzorzec:

```
STATUS="FAILED"
```

```
cleanup() {
```

```
  if [ -n "${tmp_workspace:-}" ] && [ -d "$tmp_workspace" ]; then
```

```
    rm -rf "$tmp_workspace"
```

```
  fi
```

```
}
```

```
on_sigint() {
```

```
  STATUS="INTERRUPTED"
```

```
  log_error "Przerwano działanie skryptu sygnałem SIGINT"
```

```
  generate_final_report "$STATUS" "Skrypt został przerwany przez użytkownika"
```

```
  cleanup
```

```
  exit "$EXIT_SIGINT"
```

```
}
```

```
on_sigterm() {
```

```
  STATUS="TERMINATED"
```

```
  log_error "Przerwano działanie skryptu sygnałem SIGTERM"
```

```
  generate_final_report "$STATUS" "Skrypt został zakończony sygnałem SIGTERM"
```

```
  cleanup
```

```
  exit "$EXIT_SIGTERM"
```

```
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



trap cleanup EXIT

trap on_sigint INT

trap on_sigterm TERM

Kod 130 odpowiada typowemu przerwaniu przez Ctrl+C, czyli sygnałowi SIGINT. Dla sygnału SIGTERM można przyjąć kod 143, zgodnie z często stosowaną konwencją 128 + numer sygnału. W prostszym wariantcie dopuszczalne jest obsłużenie obu sygnałów jednym handlerem i zakończenie kodem 130, ale rozdzielenie SIGINT i SIGTERM jest bardziej precyzyjne.

Funkcja cleanup powinna być bezpieczna przy wielokrotnym wywołaniu. Jest to istotne, ponieważ po obsłudze sygnału skrypt może dodatkowo wykonać procedury przypięte do EXIT. Dlatego cleanup powinien sprawdzać, czy katalog tymczasowy istnieje, zanim spróbuje go usunąć.

Należy również rozróżnić zasoby tymczasowe i wyniki finalne. Przerwanie powinno usuwać katalog roboczy, pliki częściowe i dane pomocnicze. Nie powinno usuwać poprawnie utworzonego, zweryfikowanego artefaktu, który został już przeniesiony do katalogu output/.

Dobłą praktyką jest utworzenie raportu ze statusem INTERRUPTED. Brak raportu utrudnia odróżnienie sytuacji, w której skrypt w ogóle nie wystartował, od sytuacji, w której rozpoczął pracę, ale został zatrzymany w trakcie działania.

Atomowe tworzenie artefaktu wynikowego

Artefakt wynikowy to finalny plik lub zestaw plików utworzonych przez skrypt, na przykład archiwum, raport, paczka danych albo wynik przetwarzania. W tym laboratorium przykładowym artefaktem będzie archiwum .tar.gz tworzone na podstawie katalogu wejściowego.

Plik wynikowy nie powinien być tworzony od razu w katalogu output/ jako finalny rezultat. Jeżeli operacja zostanie przerwana w połowie, w katalogu wynikowym mógłby pozostać częściowy lub uszkodzony plik. Poprawny wzorec polega na utworzeniu artefaktu w katalogu tymczasowym, sprawdzeniu go, a dopiero potem przeniesieniu do lokalizacji finalnej.

Przykładowy schemat:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
timestamp="$(date '+%Y-%m-%d_%H%M%S')"
```

```
tmp_artifact="$tmp_workspace/${BACKUP_PREFIX}_${timestamp}.tar.gz"
```

```
final_artifact="$OUTPUT_DIR/${BACKUP_PREFIX}_${timestamp}.tar.gz"
```

```
tar -czf "$tmp_artifact" -C "$SOURCE_DIR" .
```

```
tar -tzf "$tmp_artifact" >/dev/null
```

```
mv "$tmp_artifact" "$final_artifact"
```

W tym układzie katalog output/ otrzymuje plik dopiero wtedy, gdy archiwum zostało poprawnie utworzone i da się je odczytać. Jeżeli tar zakończy się błędem, plik pozostaje w katalogu tymczasowym i zostanie usunięty przez cleanup.

Ten wzorzec należy stosować nie tylko dla backupu, ale również dla raportów i manifestów. Finalny plik powinien pojawić się w lokalizacji docelowej dopiero po poprawnym zakończeniu etapu, który go tworzy.

Manifest i weryfikacja artefaktu

Utworzenie pliku nie wystarcza do uznania operacji za zakończoną sukcesem. Skrypt powinien potwierdzić, że artefakt istnieje, ma niezerowy rozmiar, da się go odczytać i można go jednoznacznie zidentyfikować. Do tego służy manifest.

Manifest to plik opisujący utworzony artefakt. W laboratorium przyjmujemy format KEY=VALUE, ponieważ jest prosty do wygenerowania w Bashu i łatwy do późniejszego przetworzenia.

Przykładowy manifest:

```
BACKUP_FILE=output/project_backup_2026-05-15_120000.tar.gz
```

```
BACKUP_SIZE_BYTES=12345
```

```
FILES_IN_ARCHIVE=5
```

```
SHA256=2f4c...
```

```
ARCHIVE_STATUS=OK
```

```
CREATED_AT=2026-05-15T12:00:00
```

Minimalna weryfikacja archiwum powinna obejmować odczyt listy plików oraz obliczenie sumy kontrolnej:

```
tar -tzf "$final_artifact" >/dev/null
```

```
sha256sum "$final_artifact"
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Liczbę plików w archiwum można wyznaczyć na przykład tak:

```
files_in_archive="$(tar -tzf "$final_artifact" | wc -l)"
```

Rozmiar pliku można pobrać poleceniem:

```
backup_size="$(du -b "$final_artifact" | awk '{print $1}')"
```

Lub:

```
backup_size="$(wc -c < "$final_artifact")"
```

Jeżeli archiwum nie przejdzie weryfikacji, manifest nie powinien zawierać statusu OK. Skrypt powinien zapisać błąd w logu, ustawić odpowiedni status operacji i przejść do raportowania problemu.

Pomocniczy skrypt Pythona w procesie operacyjnym

Bash dobrze sprawdza się jako warstwa orkiestrująca proces: uruchamia polecenia, pracuje na plikach i katalogach, sprawdza kody zakończenia, zakłada blokady, obsługuje sygnały i integruje narzędzia systemowe. Nie jest jednak wygodnym narzędziem do bardziej złożonego przetwarzania danych strukturalnych, takich jak raporty KEY=VALUE, JSON, CSV albo większe zestawienia tekstowe.

W praktyce często stosuje się podział odpowiedzialności: Bash steruje przebiegiem operacji, a Python wykonuje wybrany etap przetwarzania danych. Pomocniczy skrypt Pythona może na przykład sprawdzić kompletność manifestu, przekształcić dane do czytelnego raportu, policzyć statystyki, utworzyć raport JSON albo przygotować podsumowanie na podstawie kilku plików wejściowych.

Pomocniczy skrypt Pythona powinien mieć jednoznaczny interfejs uruchomieniowy. W tym celu należy użyć modułu argparse, aby jawnie zdefiniować wymagane argumenty, na przykład plik wejściowy i plik wyjściowy:

```
python3 python/summarize_manifest.py \
--input reports/backup_manifest.env \
--output reports/backup_summary.txt
```

Taki skrypt nie powinien pytać użytkownika o dane w trakcie działania. Powinien pobrać wszystkie parametry z argumentów, sprawdzić poprawność plików wejściowych, utworzyć wynik w przewidywalnej lokalizacji i zwrócić kod zakończenia informujący o sukcesie albo błędzie.

Współpraca Basha i Pythona powinna być kontrolowana tak samo jak każde inne polecenie zewnętrzne. Główny skrypt Bash powinien sprawdzić, czy python3 jest



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



dostępny, uruchomić skrypt pomocniczy przez `run_checked` i zareagować na kod zakończenia. Jeżeli skrypt Pythona zwróci błąd, proces operacyjny nie powinien udawać, że raport został poprawnie przygotowany.

Minimalna zasada integracji jest następująca:

Bash odpowiada za proces.

Python odpowiada za wybrany etap przetwarzania danych.

Kod zakończenia Pythona wraca do Basha i wpływa na status całego joba.

W tym laboratorium Python może zostać użyty do przetworzenia manifestu artefaktu i przygotowania czytelnego podsumowania. Jest to przykład wzorca, który można zastosować także do innych danych: logów, plików CSV, odpowiedzi JSON albo raportów wygenerowanych przez narzędzia zewnętrzne.

Raport końcowy operacji

Raport końcowy podsumowuje wynik całego procesu. Nie jest tym samym co log. Log pokazuje przebieg działania krok po kroku, natomiast raport końcowy zawiera ostateczny status, lokalizację artefaktów i najważniejsze informacje diagnostyczne.

Raport powinien powstać zarówno po sukcesie, jak i po błędzie, o ile skrypt jest w stanie go zapisać. Dzięki temu po uruchomieniu bez nadzoru można szybko sprawdzić, czy proces zakończył się poprawnie, został przerwany albo nie wykonał się z powodu problemu.

Przykładowy format raportu:

```
DATE=2026-05-15T12:00:00
STATUS=OK
SOURCE_DIR=input/project_data
BACKUP_FILE=output/project_backup_2026-05-15_120000.tar.gz
BACKUP_SIZE_BYTES=12345
FILES_IN_ARCHIVE=5
SHA256=2f4c...
LOG_FILE=logs/ops_backup.log
MANIFEST_FILE=reports/backup_manifest.env
MESSAGE=Operacja zakończona poprawnie
```

Dopuszczalne wartości statusu powinny być ograniczone, na przykład:

```
OK
FAILED
INTERRUPTED
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Raport końcowy również powinien być zapisywany atomowo. Najpierw należy utworzyć plik tymczasowy, a dopiero po poprawnym zapisie przenieść go do lokalizacji docelowej:

```
tmp_report="$tmp_workspace/ops_backup_report.txt"  
final_report="$REPORT_DIR/ops_backup_report.txt"
```

```
generate_report_content > "$tmp_report"  
mv "$tmp_report" "$final_report"
```

Jeżeli raportu nie da się zapisać, skrypt powinien zakończyć działanie kodem przewidzianym dla błędu raportowania, na przykład EXIT_REPORT.

Testy smoke

Testy smoke to prosty zestaw sprawdzeń, które potwierdzają, że najważniejsze tryby działania skryptu nadal funkcjonują. Nie zastępują pełnych testów jednostkowych ani integracyjnych. Mają szybko wykrywać podstawowe uszkodzenia po zmianach w kodzie.

W laboratorium testy smoke należy umieścić w pliku:

tests/smoke_tests.sh

Minimalny zakres testów powinien obejmować:

- help zwraca kod 0,
- brak --config zwraca kod 2,
- nieistniejący config zwraca kod EXIT_CONFIG,
- dry-run nie tworzy pliku w output/,
- poprawne uruchomienie tworzy artefakt,
- poprawne uruchomienie tworzy manifest,
- poprawne uruchomienie tworzy raport,
- aktywna blokada powoduje kod EXIT_LOCKED.

Wynik testów powinien być czytelny:

PASS: help
PASS: missing config
PASS: dry-run
PASS: artifact created
PASS: manifest created
FAIL: lock test



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Test smoke powinien kończyć się kodem 0, jeżeli wszystkie sprawdzenia zakończyły się sukcesem. Jeżeli co najmniej jeden test nie przejdzie, skrypt testowy powinien zwrócić kod niezerowy.

W praktyce testy smoke są szczególnie przydatne po zmianach w funkcjach pomocniczych, takich jak `run_checked`, `retry`, `cleanup` albo generowanie raportu. Pozwalają szybko sprawdzić, czy nie uszkodzono podstawowego przepływu procesu.

Przygotowanie do uruchamiania z cron

Skrypt operacyjny powinien być przygotowany do uruchamiania cyklicznego. W tym laboratorium służy do tego przykładowy wpis cron, zapisany w katalogu:

`cron/ops_backup.cron`

Przykładowy wpis:

```
*/15 * * * * /home/student/lab9/bin/ops_backup.sh --config /home/student/lab9/config/ops.env >>  
/home/student/lab9/logs/cron.log 2>&1
```

Wpis cron powinien używać ścieżek bezwzględnych. Harmonogram zadań nie musi uruchamiać skryptu z tego samego katalogu, z którego student uruchamia go ręcznie. Użycie ścieżek względnych może więc prowadzić do błędów trudnych do odtworzenia.

Należy również jawnie przekierować wyjście:

```
>> /home/student/lab9/logs/cron.log 2>&1
```

Dzięki temu standardowe wyjście i standardowe wyjście błędów zostają zapisane do pliku logu crona. Nie zastępuje to właściwego logu operacyjnego skryptu, ale pozwala wykryć problemy występujące jeszcze przed startem logiki programu, na przykład błędną ścieżkę do skryptu albo brak uprawnień wykonania.

Przy uruchamianiu cyklicznym szczególnie ważny jest `lock`. Jeżeli poprzednia instancja nadal działa, kolejna powinna zakończyć się kodem `EXIT_LOCKED`, zamiast wykonywać drugi proces równolegle.

Ćwiczenia samodzielne

Wszystkie zadania w tym laboratorium prowadzą do przygotowania jednego skryptu operacyjnego:

`bin/ops_backup.sh`



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Skrypt ten ma wykonywać przykładowe zadanie utrzymaniowe: przygotowanie artefaktu wynikowego w postaci backupu katalogu projektu. Backup jest tu scenariuszem roboczym. Głównym celem nie jest nauka wykonywania kopii zapasowych, lecz zbudowanie skryptu, który działa bez nadzoru, obsługuje błędy, zapisuje log, tworzy raport, zabezpiecza się przed równoległym uruchomieniem i sprząta zasoby tymczasowe.

Zadania należy wykonywać kolejno. Każde kolejne zadanie rozbudowuje wcześniejszy kod.

Do laboratorium dołączone jest archiwum lab9-health-mock.zip, które zawiera prosty lokalny serwis HTTP uruchamiany przez Docker Compose. Serwis ten udostępnia endpointy wykorzystywane do testowania mechanizmów timeout i retry.

Archiwum należy rozpakować obok katalogu lab9/, a następnie uruchomić serwis:

```
unzip lab9-health-mock.zip
```

```
cd lab9-health-mock
```

```
docker compose up --build
```

Po uruchomieniu serwis powinien być dostępny pod adresem:

```
http://127.0.0.1:8000
```

Podstawowy endpoint kontrolny:

```
http://127.0.0.1:8000/health
```

Do sprawdzenia działania można użyć polecenia:

```
curl -fsS http://127.0.0.1:8000/health
```

Oczekiwany wynik to odpowiedź OK oraz kod zakończenia 0.

Zadanie 0. Przygotowanie środowiska operacyjnego

Celem zadania jest przygotowanie katalogów, konfiguracji i danych testowych, na których będzie pracował skrypt operacyjny. Struktura katalogów ma od początku oddzielać kod skryptu, konfigurację, dane wejściowe, wyniki, logi, raporty, testy i pliki tymczasowe.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Należy utworzyć strukturę:

```
lab9/  
├── bin/  
├── lib/  
├── config/  
├── input/project_data/  
├── output/  
├── tmp/  
├── reports/  
├── logs/  
├── python/  
├── tests/  
└── cron/
```

W katalogu `input/project_data/` należy przygotować przykładowe dane do przetwarzania:

```
input/project_data/  
├── app.log  
├── config.json  
├── notes.txt  
└── data/  
    ├── users.csv  
    └── orders.csv
```

Pliki mogą mieć prostą zawartość testową. Przykładowo `app.log` może zawierać kilka linii INFO, WARN i ERROR, a pliki CSV po kilka rekordów.

W pliku:

`config/ops.env`

należy umieścić konfigurację procesu:

`PROJECT_ROOT=/home/student/lab9`

`SOURCE_DIR=/home/student/lab9/input/project_data`

`OUTPUT_DIR=/home/student/lab9/output`

`TMP_DIR=/home/student/lab9/tmp`

`REPORT_DIR=/home/student/lab9/reports`



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
LOG_DIR=/home/student/lab9/logs  
LOG_FILE=/home/student/lab9/logs/ops_backup.log  
BACKUP_PREFIX=project_backup  
SERVICE_URL=http://127.0.0.1:8000/health  
LOCK_FILE=/home/student/lab9/tmp/ops_backup.lock
```

Efektem zadania ma być gotowe środowisko robocze, na którym będą działały kolejne etapy skryptu.

Zadanie 1. Interfejs finalnego skryptu operacyjnego

Celem zadania jest przygotowanie podstawowego interfejsu uruchomieniowego skryptu. Skrypt operacyjny powinien być uruchamiany za pomocą argumentów, a nie przez ręczne wpisywanie danych w trakcie działania.

Należy utworzyć plik:

```
bin/ops_backup.sh
```

Skrypt ma obsługiwać co najmniej następujące wywołania:

```
bin/ops_backup.sh --help  
bin/ops_backup.sh --config config/ops.env  
bin/ops_backup.sh --config config/ops.env --dry-run
```

Opcja `--help` ma wypisać krótką instrukcję użycia. Opcja `--config FILE` wskazuje plik konfiguracyjny procesu. Opcja `--dry-run` będzie używana później do pokazania planu działania bez tworzenia backupu.

W przypadku błędnego użycia, na przykład braku argumentu po `--config` albo nieznanej opcji, skrypt ma wypisać komunikat na `STDERR` i zakończyć działanie kodem 2.

W komunikacie pomocy należy opisać:

```
--help  
--config FILE  
--dry-run
```

Efektem zadania jest szkielet skryptu, który ma stabilny interfejs i może być dalej rozbudowywany.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 2. Kontrakt kodów wyjścia

Celem zadania jest wprowadzenie jawnej mapy kodów zakończenia. Skrypt operacyjny będzie używany przez człowieka, testy smoke i potencjalnie przez cron, dlatego różne klasy problemów powinny mieć różne kody wyjścia.

W pliku `bin/ops_backup.sh` należy zdefiniować stałe:

```
EXIT_OK=0
EXIT_RUNTIME_ERROR=1
EXIT_USAGE=2
EXIT_CONFIG=3
EXIT_PREFLIGHT=4
EXIT_LOCKED=10
EXIT_EXTERNAL=20
EXIT_REPORT=30
EXIT_INTERRUPTED=130
```

Skrypt ma używać tych nazw zamiast przypadkowych wartości typu `exit 1`.

Znaczenie kodów:

- 0 sukces
- 1 ogólny błąd wykonania
- 2 błędne użycie skryptu
- 3 błąd konfiguracji
- 4 błąd preflightu
- 10 aktywna blokada innego uruchomienia
- 20 błąd polecenia zewnętrznego
- 30 błąd raportowania
- 130 przerwanie przez użytkownika lub sygnał

Efekt zadania jest czytelny kontrakt zachowania skryptu w sytuacjach poprawnych i awaryjnych.

Zadanie 3. Wczytanie konfiguracji i walidacja wymaganych ustawień

Celem zadania jest wczytanie parametrów pracy z pliku konfiguracyjnego. Skrypt nie powinien mieć na sztywno wpisanych ścieżek do katalogów, logów, raportów ani pliku blokady.

Skrypt ma wczytywać konfigurację z pliku wskazanego przez:

```
--config config/ops.env
```

Należy sprawdzić, czy plik konfiguracyjny istnieje i czy zawiera wymagane zmienne:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



SOURCE_DIR
OUTPUT_DIR
TMP_DIR
REPORT_DIR
LOG_DIR
LOG_FILE
BACKUP_PREFIX
SERVICE_URL
LOCK_FILE

Jeżeli plik nie istnieje albo brakuje którejkolwiek wymaganej zmiennej, skrypt ma wypisać komunikat diagnostyczny i zakończyć działanie kodem: *EXIT_CONFIG*

Na tym etapie skrypt nie musi jeszcze zapisywać błędów do pliku logu. Jeżeli konfiguracja nie została poprawnie wczytana, ścieżka LOG_FILE może być nieznana albo niepoprawna. W takim przypadku komunikat błędów powinien trafić na STDERR.

Efektom zadania jest skrypt, który potrafi uruchomić się z konfiguracją i odmówić pracy, jeżeli konfiguracja jest niekompletna.

Zadanie 4. Tryb --dry-run

Celem zadania jest dodanie trybu planowania operacji. Tryb --dry-run pozwala sprawdzić, co skrypt zamierza zrobić, bez tworzenia backupu i bez modyfikowania katalogu output/.

Należy dodać obsługę wywołania:

```
bin/ops_backup.sh --config config/ops.env --dry-run
```

W tym trybie skrypt ma:

1. wczytać konfigurację,
2. sprawdzić podstawowe informacje o katalogu źródłowym,
3. policzyć liczbę plików przeznaczonych do przetworzenia,
4. policzyć łączny rozmiar danych wejściowych,
5. wypisać plan operacji.

Przykładowy wynik:

```
MODE=DRY_RUN
```

```
SOURCE_DIR=/home/student/lab9/input/project_data
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
OUTPUT_DIR=/home/student/lab9/output  
TMP_DIR=/home/student/lab9/tmp  
REPORT_DIR=/home/student/lab9/reports  
BACKUP_PREFIX=project_backup  
FILES_TO_BACKUP=5  
TOTAL_SIZE_BYTES=12345  
SERVICE_URL=http://127.0.0.1:8000/health
```

Po uruchomieniu w trybie `--dry-run` w katalogu `output/` nie powinien pojawić się żaden nowy backup.

Efektem zadania jest możliwość bezpiecznego sprawdzenia planu działania przed wykonaniem operacji z efektami ubocznymi.

Zadanie 5. Preflight procesu operacyjnego

Celem zadania jest sprawdzenie, czy skrypt może bezpiecznie rozpocząć właściwą operację. Preflight ma wykryć problemy przed utworzeniem backupu, raportu lub plików tymczasowych.

Należy dodać funkcję:

`preflight`

Funkcja ma sprawdzić:

czy istnieje `SOURCE_DIR`,
czy `SOURCE_DIR` jest katalogiem,
czy można utworzyć `OUTPUT_DIR`,
czy można utworzyć `TMP_DIR`,
czy można utworzyć `REPORT_DIR`,
czy można utworzyć `LOG_DIR`,
czy możliwy jest zapis do `LOG_FILE`,
czy dostępne są wymagane narzędzia:
`tar`, `gzip`, `find`, `du`, `date`, `sha256sum`, `mktemp`, `flock`, `timeout`, `curl`, `python3`.

Wynik preflightu należy zapisać do:

`reports/preflight_report.txt`



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykładowy raport:

```
OK: tar
OK: gzip
OK: find
OK: SOURCE_DIR=/home/student/lab9/input/project_data
OK: OUTPUT_DIR=/home/student/lab9/output
OK: TMP_DIR=/home/student/lab9/tmp
OK: LOG_FILE=/home/student/lab9/logs/ops_backup.log
```

Jeżeli brakuje elementu krytycznego, skrypt ma zakończyć działanie kodem:

```
EXIT_PREFLIGHT
```

Efektem zadania jest zabezpieczenie przed rozpoczęciem procesu w środowisku, które nie spełnia wymagań.

Zadanie 6. Logowanie operacyjne ze znacznikiem czasu

Celem zadania jest dodanie trwałego logu działania skryptu. Skrypt uruchamiany bez nadzoru musi zostawiać ślad, który pozwala później ustalić, co zostało wykonane i gdzie wystąpił problem.

Należy przygotować plik:

```
lib/logging.sh
```

W pliku należy zaimplementować funkcje:

```
log_info
log_warn
log_error
```

Funkcje mają zapisywać wpisy do pliku wskazanego przez zmienną:

```
LOG_FILE
```

Format wpisu:

```
[2026-05-15 12:00:00] INFO Start joba
[2026-05-15 12:00:01] WARN Ponawiam próbę po błędzie
[2026-05-15 12:00:03] ERROR Nie udało się utworzyć backupu
```

Funkcja `log_error` ma zapisywać komunikat do logu oraz wypisywać go na `STDERR`.

Efektem zadania jest mechanizm logowania, który będzie wykorzystywany przez kolejne etapy skryptu.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 7. run_checked dla poleceń zewnętrznych

Celem zadania jest ujednolichenie sposobu uruchamiania poleceń zewnętrznych. Skrypt będzie korzystał z takich narzędzi jak mkdir, tar, sha256sum, mv, curl i python3. Każde z nich może zakończyć się błędem, dlatego kod zakończenia musi być kontrolowany.

Należy przygotować plik:

lib/run_checked.sh

W pliku należy zaimplementować funkcję:

run_checked

Funkcja powinna przyjmować opis operacji oraz polecenie do wykonania. Przykład użycia:

```
run_checked "Utworzenie katalogu output" mkdir -p "$OUTPUT_DIR"  
run_checked "Sprawdzenie archiwum" tar -tzf "$backup_file"
```

Funkcja ma:

- zapisać w logu informację o rozpoczęciu operacji,
- uruchomić polecenie,
- odczytać kod zakończenia,
- w przypadku błędu zapisać komunikat przez log_error,
- zwrócić kod zakończenia polecenia.

Skrypt główny ma na podstawie wyniku run_checked zdecydować, czy kontynuować pracę, czy zakończyć proces kodem EXIT_EXTERNAL.

Efektem zadania jest centralny mechanizm kontroli poleceń zewnętrznych.

Zadanie 8. Blokada równoległego uruchomienia przez flock

Celem zadania jest zabezpieczenie skryptu przed jednoczesnym uruchomieniem dwóch instancji. Jest to ważne szczególnie wtedy, gdy skrypt ma być uruchamiany cyklicznie przez cron.

Należy dodać mechanizm blokady z użyciem pliku wskazanego przez:

LOCK_FILE

Jeżeli blokada jest zajęta, skrypt ma zakończyć działanie kodem:

EXIT_LOCKED



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Należy przetestować sytuację, w której pierwsze uruchomienie skryptu nadal trwa, a drugie próbuje wystartować.

Aby przetestować blokadę ręcznie należy w pierwszym terminalu wejść do katalogu projektu i ręcznie założyć blokadę na ten sam plik, którego używa skrypt:

```
mkdir -p /home/student/lab9/tmp  
exec 9>/home/student/lab9/tmp/ops_backup.lock  
flock -n 9
```

Następnie, bez zamykania pierwszego terminala, w drugim terminalu należy uruchomić skrypt:

```
/home/student/lab9/bin/ops_backup.sh --config /home/student/lab9/config/ops.env  
echo $?
```

Oczekiwanym wynikiem jest kod 10, czyli EXIT_LOCKED.

Po zakończeniu testu blokadę można zwolnić w pierwszym terminalu:

```
flock -u 9  
exec 9>&-
```

Efektom zadania jest ochrona przed równoległym zapisem do tych samych katalogów, raportów i logów.

Zadanie 9. timeout i retry dla zależnej usługi

Celem zadania jest dodanie odporności na chwilową niedostępność usługi. Skrypt operacyjny nie powinien czekać bez końca na odpowiedź HTTP ani kończyć pracy po pierwszym chwilowym błędzie.

Należy przygotować plik:

```
lib/retry.sh
```

W pliku należy zaimplementować funkcję:

```
retry
```

Funkcja ma przyjmować:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



liczbę prób,
czas oczekiwania między próbami,
polecenie do wykonania.

Przykładowe użycie:

```
retry 3 2 timeout 5 curl -fsS "$SERVICE_URL"
```

Znaczenie parametrów:

- 3 - maksymalnie trzy próby,
- 2 - dwie sekundy przerwy między próbami,
- 5 - maksymalnie pięć sekund na pojedyncze wywołanie curl.

Jeżeli usługa odpowie poprawnie, skrypt może kontynuować pracę. Jeżeli wszystkie próby zakończą się błędem, należy zapisać problem w logu i zakończyć działanie kontrolowanym kodem, na przykład EXIT_EXTERNAL.

Do testów należy użyć lokalnego serwisu z archiwum lab9-health-mock.zip. Serwis należy uruchomić w osobnym terminalu poleceniem:

```
cd lab9-health-mock
```

```
docker compose up --build
```

Dostępne są następujące endpointy testowe:

```
http://127.0.0.1:8000/health
```

zwraca poprawną odpowiedź i służy do testu przypadku sukcesu;

```
http://127.0.0.1:8000/health/fail
```

zwraca błąd HTTP 503 i służy do testu przypadku trwałego niepowodzenia;

```
http://127.0.0.1:8000/health/slow
```

odpowiada z opóźnieniem i służy do testu działania timeout;

```
http://127.0.0.1:8000/health/flaky
```

symuluje chwilową awarię, czyli początkowo zwraca błąd, a po kolejnych próbach zwraca odpowiedź poprawną;

```
http://127.0.0.1:8000/reset
```

resetuje licznik używany przez endpoint /health/flaky.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykładowe testy ręczne:

- `curl -fsS http://127.0.0.1:8000/health`
- `timeout 5 curl -fsS http://127.0.0.1:8000/health/slow`
- `curl -fsS http://127.0.0.1:8000/reset`
- `curl -i http://127.0.0.1:8000/health/flaky`

Efektem zadania jest mechanizm, który rozróżnia chwilowy problem od trwałej awarii.

Zadanie 10. Katalog tymczasowy i trap cleanup EXIT

Celem zadania jest przeniesienie operacji roboczych do bezpiecznego katalogu tymczasowego. Skrypt nie powinien tworzyć częściowych plików bezpośrednio w katalogu `output/`.

Należy utworzyć katalog tymczasowy za pomocą:

```
tmp_workspace="$(mktemp -d "$TMP_DIR/ops_backup.XXXXXX")"
```

Następnie należy przygotować funkcję:

```
cleanup
```

Funkcja ma usuwać katalog tymczasowy, jeżeli został utworzony.

Przykładowy wzorzec:

```
cleanup() {  
    if [ -n "${tmp_workspace:-}" ] && [ -d "$tmp_workspace" ]; then  
        rm -rf "$tmp_workspace"  
    fi  
    tmp_workspace=""  
}
```

trap cleanup EXIT

- Funkcja `cleanup` powinna być idempotentna, czyli bezpieczna przy wielokrotnym wywołaniu. Jest to ważne, ponieważ sprzątanie może zostać uruchomione zarówno przez `trap cleanup EXIT`, jak i przez funkcję obsługującą przerwanie działania.
- Funkcja `cleanup` nie może usuwać poprawnie utworzonych artefaktów końcowych z katalogu `output/`.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

- Po poprawnym zakończeniu skryptu w katalogu tmp/ nie powinny pozostawać katalogi robocze po danym uruchomieniu.
- Efektem zadania jest bezpieczne zarządzanie zasobami roboczymi.

Zadanie 11. Obsługa SIGINT i SIGTERM

Celem zadania jest kontrolowane zakończenie skryptu po przerwaniu działania. Skrypt może zostać zatrzymany przez użytkownika przez Ctrl+C albo przez system przez SIGTERM.

Należy dodać obsługę sygnałów:

```
trap on_interrupt INT TERM
```

Funkcja on_interrupt ma wykonać następujące czynności:

zapisać informację o przerwaniu do logu,
wypisać komunikat na STDERR,
ustawić status operacji na INTERRUPTED,
utworzyć raport końcowy, jeżeli jest to możliwe,
wykonać cleanup,
zakończyć działanie kodem 130.

Do testu należy uruchomić skrypt, doprowadzić do momentu, w którym istnieje katalog tymczasowy, a następnie przerwać działanie przez Ctrl+C.

Po przerwaniu należy sprawdzić:

czy katalog tymczasowy został usunięty,
czy w logu znajduje się informacja o przerwaniu,
czy raport końcowy ma status INTERRUPTED,
czy kod zakończenia wynosi 130.

Efektem zadania jest skrypt, który nie zostawia środowiska w stanie przypadkowym po wymuszonym zatrzymaniu.

Zadanie 12. Atomowe tworzenie backupu

Celem zadania jest utworzenie artefaktu wynikowego w sposób atomowy. Finalny backup nie może powstawać bezpośrednio w katalogu output/, ponieważ przerwanie operacji mogłoby zostawić tam uszkodzony plik.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Należy utworzyć archiwum katalogu wskazanego przez:

SOURCE_DIR

Archiwum ma najpierw powstać w katalogu tymczasowym, na przykład:

/home/student/lab9/tmp/ops_backup.XYZ/project_backup_2026-05-15_120000.tar.gz

Dopiero po poprawnym utworzeniu i podstawowej weryfikacji archiwum plik ma zostać przeniesiony do katalogu:

OUTPUT_DIR

Przykładowa finalna nazwa:

/home/student/lab9/output/project_backup_2026-05-15_120000.tar.gz

Jeżeli utworzenie archiwum się nie powiedzie, w katalogu output/ nie powinien pojawić się nowy plik.

W tym laboratorium zakładamy, że katalog roboczy i katalog output/ znajdują się w obrębie tego samego projektu. Dzięki temu przeniesienie zweryfikowanego artefaktu do katalogu output/ dobrze pokazuje ideę atomowego publikowania wyniku. W bardziej rygorystycznym wariacie można tworzyć plik roboczy bezpośrednio w output/ pod nazwą zaczynającą się od kropki i kończącą się sufiksem .tmp, a po weryfikacji wykonać samo przemianowanie na nazwę finalną.

Efektem zadania jest bezpieczne tworzenie artefaktu wynikowego bez pozostawiania częściowych backupów.

Zadanie 13. Manifest i weryfikacja artefaktu

Celem zadania jest potwierdzenie, że utworzony backup jest możliwy do sprawdzenia. Samo istnienie pliku .tar.gz nie wystarcza do uznania operacji za poprawną.

Po utworzeniu backupu należy wygenerować manifest:

reports/backup_manifest.env

Minimalna zawartość manifestu:

BACKUP_FILE=/home/student/lab9/output/project_backup_2026-05-15_120000.tar.gz
BACKUP_SIZE_BYTES=...
FILES_IN_ARCHIVE=...
SHA256=...



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



ARCHIVE_STATUS=OK
CREATED_AT=...

Weryfikacja ma obejmować:

```
tar -tzf "$backup_file"  
sha256sum "$backup_file"
```

Należy również policzyć liczbę plików w archiwum oraz rozmiar pliku backupu.

Jeżeli archiwum nie da się odczytać, skrypt ma:

zapisać błąd do logu,
nie oznaczać manifestu statusem OK,
przejsć do raportowania błędu.

Efektem zadania jest manifest pozwalający zweryfikować wynik operacji.

Zadanie 14. Pomocniczy skrypt Pythona do podsumowania manifestu

Celem zadania jest pokazanie praktycznej współpracy Basha i Pythona. Bash steruje całym procesem, a Python wykonuje wybrany etap przetwarzania danych.

Należy przygotować skrypt:

python/summarize_manifest.py

Skrypt ma używać argparse i przyjmować argumenty:

```
python3 python/summarize_manifest.py \  
--input reports/backup_manifest.env \  
--output reports/backup_summary.txt
```

Skrypt Pythona ma:

wczytać manifest w formacie KEY=VALUE,
sprawdzić obecność wymaganych pól,
utworzyć czytelne podsumowanie tekstowe,
zwrócić kod niezerowy, jeżeli manifest jest niekompletny albo nieczytelny.

Przykładowy wynik:

Backup summary

File: /home/student/lab9/output/project_backup_2026-05-15_120000.tar.gz

Size: 12345 bytes

Files: 5

SHA256: ...

Status: OK



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Główny skrypt Bash powinien uruchamiać skrypt Pythona przez `run_checked`. Jeżeli Python zwróci błąd, cały proces powinien zostać oznaczony jako niepoprawny.

Efektem zadania jest kontrolowana integracja pomocniczego narzędzia Pythona z procesem Bash.

Zadanie 15. Raport końcowy zapisywany atomowo

Celem zadania jest utworzenie końcowego raportu operacji. Raport ma podsumować wynik całego procesu w formie czytelnej dla użytkownika, testów lub systemu automatyzacji.

Skrypt ma utworzyć plik:

`reports/ops_backup_report.txt`

Raport powinien zawierać co najmniej:

```
DATE=...  
STATUS=OK|FAILED|INTERRUPTED  
SOURCE_DIR=...  
BACKUP_FILE=...  
BACKUP_SIZE_BYTES=...  
FILES_IN_ARCHIVE=...  
SHA256=...  
LOG_FILE=...  
MANIFEST_FILE=...  
MESSAGE=...
```

Raport ma powstać po sukcesie, po błędzie oraz po przerwaniu działania, o ile jest to technicznie możliwe.

Raport również powinien być zapisywany atomowo:

najpierw plik tymczasowy,
potem `mv` do `reports/ops_backup_report.txt`.

Jeżeli raportu nie uda się zapisać, skrypt powinien zakończyć działanie kodem:

```
EXIT_REPORT
```

Efektem zadania jest jednoznaczny raport końcowy opisujący status całej operacji.

Zadanie 16. Testy smoke

Celem zadania jest przygotowanie prostego zestawu testów, które szybko sprawdzają, czy najważniejsze tryby działania skryptu nadal funkcjonują. Testy



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



smoke nie są pełnymi testami jednostkowymi. Mają wykrywać podstawowe awarie po zmianach w kodzie.

Należy przygotować skrypt:

tests/smoke_tests.sh

Skrypt testowy ma automatycznie uruchamiać bin/ops_backup.sh w kilku scenariuszach i sprawdzać kody zakończenia oraz obecność wymaganych plików.

Minimalny zakres testów:

--help zwraca kod 0,
brak --config zwraca kod 2,
nieistniejący config zwraca kod EXIT_CONFIG,
--dry-run nie tworzy backupu w output/
poprawne uruchomienie tworzy backup,
poprawne uruchomienie tworzy manifest,
poprawne uruchomienie tworzy raport,
druga instancja przy aktywnym locku kończy się kodem EXIT_LOCKED.

Test aktywnej blokady można najpierw wykonać ręcznie, bez modyfikowania kodu skryptu, tak jak zostało to przedstawione w zadaniu numer 8.

Ten sam scenariusz powinien zostać następnie zautomatyzowany w tests/smoke_tests.sh.

Skrypt testowy powinien wypisywać czytelne podsumowanie:

PASS: help
PASS: missing config
PASS: dry-run
PASS: backup created
PASS: manifest created
FAIL: lock test

Jeżeli wszystkie testy przejdą poprawnie, tests/smoke_tests.sh ma zakończyć się kodem 0. Jeżeli co najmniej jeden test zakończy się niepowodzeniem, skrypt testowy ma zwrócić kod niezerowy.

Efektem zadania jest podstawowa kontrola regresji po zmianach w skrypcie operacyjnym.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 17. Przygotowanie wpisu cron

Celem zadania jest przygotowanie skryptu do uruchamiania bez nadzoru. Wpis cron nie musi być instalowany w systemie, ale powinien być poprawnie zapisany i możliwy do użycia.

Należy przygotować plik:

cron/ops_backup.cron

Przykładowa zawartość:

```
*/15 * * * * /home/student/lab9/bin/ops_backup.sh --config /home/student/lab9/config/ops.env >>  
/home/student/lab9/logs/cron.log 2>&1
```

W krótkim komentarzu lub raporcie należy wyjaśnić:

dlaczego użyto ścieżek bezwzględnych,
gdzie trafia STDOUT i STDERR,
dlaczego skrypt nie może wymagać interakcji,
dlaczego potrzebny jest flock,
co stanie się, gdy poprzednie uruchomienie jeszcze trwa.

Efektem zadania jest przygotowanie skryptu do cyklicznego uruchamiania w sposób zgodny z wymaganiami pracy bez nadzoru.

Zadania dodatkowe

Zadanie dodatkowe A. Retencja backupów

Celem zadania jest ograniczenie liczby przechowywanych backupów. Skrypt operacyjny uruchamiany cyklicznie będzie tworzył kolejne pliki, dlatego katalog output/ nie powinien rosnąć bez kontroli.

Należy przygotować tryb lub osobny skrypt:

bin/backup_retention.sh

Skrypt ma zostawiać w output/ tylko ostatnie N backupów, a starsze przenosić do:

tmp/old_backups/



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



albo usuwać, jeżeli tak zostanie ustalone w konfiguracji.

Wartość N powinna pochodzić z konfiguracji:

KEEP_LAST=5

Zadanie dodatkowe B. Test odtworzenia backupu

Celem zadania jest sprawdzenie, czy backup można realnie odtworzyć. Backup, którego nie da się rozpakować i sprawdzić, nie powinien być uznawany za wiarygodny.

Należy przygotować skrypt:

bin/restore_check.sh

Skrypt ma rozpakować wskazany backup do katalogu tymczasowego i sprawdzić, czy znajdują się w nim oczekiwane pliki, na przykład:

app.log
config.json
notes.txt
data/users.csv
data/orders.csv

Po zakończeniu testu katalog tymczasowy powinien zostać usunięty.

Zadanie dodatkowe C. Tryb --json-report

Celem zadania jest przygotowanie raportu w formacie łatwym do przetwarzania przez inne narzędzia.

Należy dodać tryb:

bin/ops_backup.sh --config config/ops.env --json-report

Po jego uruchomieniu powinien powstać raport:

reports/ops_backup_report.json

Raport JSON powinien zawierać co najmniej:

status,
date,
source_dir,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



backup_file,
backup_size_bytes,
files_in_archive,
sha256,
log_file,
message.

Raport może zostać utworzony bezpośrednio w Bashu albo przez pomocniczy skrypt Pythona.

Zadanie dodatkowe D. Guard procesu i limit rozmiaru logu

Celem zadania jest przygotowanie prostego mechanizmu kontrolnego, który sprawdza stan procesu lub rozmiar pliku logu i wykonuje bezpieczną akcję po przekroczeniu ustalonego limitu. Zadanie pokazuje, że skrypt operacyjny może nie tylko wykonywać pracę, ale także pilnować warunków brzegowych: czasu działania procesu, rozmiaru logów albo obecności pliku PID.

Należy przygotować skrypt:

bin/ops_guard.sh

Skrypt powinien korzystać z konfiguracji, na przykład:

```
LOG_FILE=logs/ops_backup.log  
MAX_LOG_SIZE_BYTES=1048576  
PID_FILE=tmp/mock_worker.pid  
MAX_RUNTIME_SECONDS=60
```

Minimalny zakres działania:

1. sprawdzić, czy istnieje LOG_FILE,
2. odczytać rozmiar logu,
3. jeżeli log przekracza MAX_LOG_SIZE_BYTES, przenieść go do logs/archive/ albo utworzyć kopię z datą,
4. utworzyć nowy pusty log,
5. sprawdzić, czy istnieje PID_FILE,
6. jeżeli proces z PID_FILE istnieje zbyt długo, zakończyć go kontrolowanie,
7. zapisać działania do logu operacyjnego.

Do sprawdzenia rozmiaru pliku można użyć:

```
log_size="$(stat -c%s "$LOG_FILE")"
```

Przykładowa rotacja logu:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
timestamp="$(date '+%Y-%m-%d_%H%M%S')"  
mkdir -p logs/archive
```

```
mv "$LOG_FILE" "logs/archive/ops_backup_$timestamp.log"  
: > "$LOG_FILE"
```

Do pracy z procesem należy używać **pliku PID**, a nie ogólnego `kill` nazwa_procesu, ponieważ `kill` po nazwie może przypadkowo zakończyć nie ten proces.

Przykład kontrolowanego zakończenia procesu testowego:

```
if [ -f "$PID_FILE" ]; then  
    pid="$(cat "$PID_FILE")"  
  
    if kill -0 "$pid" 2>/dev/null; then  
        kill "$pid"  
        sleep 2  
  
        if kill -0 "$pid" 2>/dev/null; then  
            kill -9 "$pid"  
        fi  
    fi  
fi
```

Do testu należy przygotować sztuczny proces, na przykład:

```
sleep 300 &  
echo "$!" > tmp/mock_worker.pid
```

Efektem zadania jest prosty mechanizm strażnika, który reaguje na przekroczenie limitu rozmiaru logu albo na zbyt długo działający proces testowy.

Pytania pomocnicze

1. Czym różni się prosty skrypt uruchamiany ręcznie od skryptu operacyjnego uruchamianego bez nadzoru?
2. Dlaczego skrypt przeznaczony do uruchamiania przez cron nie powinien wymagać interakcji z użytkownikiem?
3. Jaką rolę pełni plik konfiguracyjny `.env` albo plik w formacie `KEY=VALUE` w skrypcie operacyjnym?
4. Po co wykonuje się preflight przed rozpoczęciem właściwej operacji?
5. Dlaczego skrypt operacyjny powinien sprawdzać dostępność narzędzi zewnętrznych przed ich użyciem?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

6. Czym różni się preflight ogólnego środowiska od preflightu konkretnego procesu operacyjnego?
7. Do czego służy flock i jaki problem rozwiązuje w skryptach uruchamianych cyklicznie?
8. Co może się stać, jeżeli dwa uruchomienia tego samego skryptu jednocześnie modyfikują te same pliki?
9. W jakiej sytuacji skrypt powinien odmówić startu z powodu aktywnego locka?
10. Do czego służy timeout?
11. Dlaczego polecenia zewnętrzne albo zapytania do usług nie powinny czekać bez limitu czasu?
12. Na czym polega mechanizm retry?
13. W jakich sytuacjach warto ponawiać operację, a w jakich nie powinno się tego robić?
14. Czym powinna zajmować się funkcja run_checked?
15. Dlaczego samo uruchomienie polecenia zewnętrznego nie oznacza jeszcze, że operacja zakończyła się sukcesem?
16. Do czego służy mktemp?
17. Dlaczego mktemp jest bezpieczniejszy niż ręczne tworzenie nazw plików tymczasowych?
18. Do czego służy trap w skryptach Bash?
19. Czym różni się obsługa zdarzeń EXIT, INT i TERM?
20. Dlaczego po przerwaniu działania przez Ctrl+C często stosuje się kod zakończenia 130?
21. Jaką rolę pełni funkcja cleanup?
22. Dlaczego funkcja cleanup nie powinna usuwać poprawnie utworzonych wyników końcowych?
23. Na czym polega atomowy zapis pliku wynikowego?
24. Dlaczego nie należy tworzyć finalnego pliku backupu bezpośrednio w katalogu docelowym?
25. Czym różni się plik tymczasowy, plik częściowy i finalny artefakt wynikowy?
26. Co powinien zawierać manifest backupu lub innego artefaktu wynikowego?
27. Po co oblicza się sumę kontrolną pliku wynikowego?
28. Jak można sprawdzić, czy archiwum .tar.gz da się poprawnie odczytać?
29. Jaką rolę pełni log operacyjny ze znacznikami czasu?
30. Czym różni się komunikat informacyjny, ostrzeżenie i błąd w logu operacyjnym?
31. Jakie informacje powinny znaleźć się w raporcie końcowym operacji?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



32. Dlaczego warto stosować jawny kontrakt kodów wyjścia?
33. Jakie przykładowe kody wyjścia można przypisać do: błędnego użycia, braku konfiguracji, aktywnego locka, błędu operacji zewnętrznej i przerwania sygnałem?
34. W jakiej roli Bash sprawdza się dobrze, a w jakiej roli pomocniczy Python może być wygodniejszy?
35. Do czego służy argparse w pomocniczym skrypcie Pythona?
36. Czym są testy smoke?
37. Co powinien sprawdzać minimalny zestaw testów smoke dla skryptu operacyjnego?
38. Dlaczego testy smoke są szczególnie przydatne po zmianach w skrypcie?
39. Co powinien zawierać przykładowy wpis cron uruchamiający skrypt cyklicznie?
40. Dlaczego w zadaniach cron warto używać ścieżek bezwzględnych i jawnego przekierowania logów?

Podsumowanie

Laboratorium dziewiąte domyka cykl zajęć z programowania skryptowego, pokazując przejście od prostych skryptów Bash do skryptu operacyjnego przeznaczonego do pracy bez nadzoru. Temat „Obsługa błędów i raportowanie problemów” zostaje tu potraktowany praktycznie: nie jako zestaw pojedynczych komunikatów błędu, lecz jako projektowanie całego procesu, który ma działać przewidywalnie także wtedy, gdy wystąpią problemy z konfiguracją, narzędziami, usługą zewnętrzną, plikami tymczasowymi albo przerwaniem działania.

W ramach laboratorium student przygotowuje skrypt operacyjny realizujący zadanie utrzymaniowe, na przykład bezpieczny backup katalogu projektu. Skrypt korzysta z konfiguracji, wykonuje preflight, zabezpiecza się przed równoległym uruchomieniem, stosuje timeout i retry, kontroluje polecenia zewnętrzne, tworzy zasoby tymczasowe przez mktemp, sprząta je za pomocą trap, generuje artefakt wynikowy w sposób atomowy, zapisuje log operacyjny oraz tworzy raport końcowy.

Istotnym elementem zajęć jest odróżnienie poprawnego wykonania operacji od samego uruchomienia polecenia. Backup, raport lub manifest nie powinny być uznane za poprawne tylko dlatego, że plik został utworzony. Skrypt powinien zweryfikować wynik, obliczyć sumę kontrolną, sprawdzić możliwość odczytu



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



archiwum i zapisać jednoznaczny status operacji. Dzięki temu powstaje narzędzie, które zostawia po sobie ślad diagnostyczny i pozwala odtworzyć przebieg działania.

Laboratorium utrwała również zasadę, że skrypt operacyjny powinien posiadać jawny kontrakt kodów wyjścia. Inaczej należy traktować błędne użycie, brak konfiguracji, błąd preflightu, aktywny lock, niepowodzenie polecenia zewnętrznego, błąd raportowania i przerwanie sygnałem. Takie rozróżnienie jest konieczne, gdy skrypt ma być wykorzystywany przez inne narzędzia, harmonogramy zadań lub procesy automatyzacji.

Ważną częścią laboratorium jest przygotowanie skryptu do pracy cyklicznej. W tym celu wykorzystywane są mechanizmy blokady przez flock, logowania do pliku, bezinterakcyjnej konfiguracji, przykładowego wpisu cron oraz testów smoke. Testy smoke pozwalają szybko sprawdzić, czy najważniejsze tryby działania skryptu nadal funkcjonują po wprowadzeniu zmian.

Efektem końcowym laboratorium jest skrypt klasy roboczej: konfigurowalny, bezinterakcyjny, odporny na podstawowe awarie, bezpiecznie zarządzający plikami tymczasowymi, zapisujący logi i raporty, gotowy do uruchamiania ręcznego lub cyklicznego. W ten sposób laboratorium stanowi profesjonalną klamrę całego cyklu, łącząc wcześniejsze zagadnienia: argumenty, funkcje, pliki, konfigurację, potoki, narzędzia zewnętrzne i obsługę błędów w jeden spójny proces automatyzacji operacyjnej.

Literatura

1. The Linux Command Handbook – Learn Linux Commands for Beginners, Link: <https://www.freecodecamp.org/news/the-linux-commands-handbook/>
2. Mendel Cooper, Advanced Bash-Scripting Guide, An in-depth exploration of the art of shell scripting, Link: <https://tldp.org/LDP/abs/html/>, format pdf: <https://tldp.org/LDP/abs/abs-guide.pdf>
3. Linux Handbook, Bash Scripting, Link: <https://linuxhandbook.com/courses/bash/>



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

