Szybkie wyszukiwanie: locate, which, whereis, type, fd, ripgrep

Poziom: srednio **Dotyczy: RHEL/Rocky 9–10, różnice Debian/Ubuntu i SLES zaznaczone**

Czas: ~15 min

`locate` odpowiada natychmiast, bo pyta bazę danych, nie dysk. `which`, `type` i `whereis` szukają programów, każde inaczej. `fd` i `ripgrep` to nowsze, szybsze zamienniki `find`/`grep` — wygodne na własnej maszynie, nie zawsze dostępne na serwerze produkcyjnym.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
EPEL	Extra Packages for Enterprise Linux: dodatkowe repozytorium pakietów dla RHEL i klonów
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
SLES	SUSE Linux Enterprise Server: płatna dystrybucja Linuksa firmy SUSE, konkurent Rocky/RHEL
SUSE	nazwa niemieckiej firmy i jej dystrybucji Linuksa, od niemieckiego skrótu Software und System Entwicklung

Fiszka

TYLKO ODCZYT

Szukanie w gotowej bazie: locate

```
locate nginx.conf
locate -i readme                    # ignoruj wielkość liter
```

```
locate -c "*.log"                  # policz trafienia, nie wypisuj
locate --regex '\.conf$'           # wzorzec jako regex zamiast globu
```

Szukamy nazwy pliku w bazie zbudowanej wcześniej przez `updatedb`, nie na żywym dysku — dlatego dostajemy odpowiedź natychmiast, nawet dla całego systemu plików.

Przykład:

```
/etc/nginx/nginx.conf
/usr/share/doc/nginx/examples/nginx.conf
```

locate = szybkie wyszukiwanie plików po nazwie na podstawie wcześniej zbudowanego indeksu

TYLKO ODCZYT

Przebudowanie bazy: updatedb

```
sudo updatedb
```

Przechodzimy cały system plików (z pominięciem katalogów wpisanych w `/etc/updatedb.conf`) i na nowo zapisujemy spis ścieżek, z którego korzysta `locate`. Uruchamiamy ją ręcznie, gdy potrzebujemy świeżych wyników zaraz po masowej operacji na plikach.

TYLKO ODCZYT

Odnalezienie pliku wykonywalnego w PATH: which

```
which ls
```

Szukamy pliku wykonywalnego w katalogach z `PATH`. Widzimy tylko zewnętrzne komendy — `which` nic nie wie o aliasach ani funkcjach powłoki, bo to twory samej powłoki, niewidoczne dla procesu zewnętrznego.

Przykład:

```
/usr/bin/ls
```

TYLKO ODCZYT

Sprawdzenie rodzaju komendy i wszystkich dopasowań: type -a

```
type cd
type ls
type -a python
```

`type` to wbudowana komenda powłoki, więc widzi wszystko, co widzi powłoka: aliasy, funkcje, słowa kluczowe, builtiny i dopiero na końcu pliki z `PATH`. `-a` (all) pokazuje wszystkie dopasowania w kolejności, w jakiej `PATH` je przeszukuje — pierwsze jest tym, które faktycznie się uruchomi.

Przykład:

```
cd is a shell builtin
ls is aliased to `ls --color=auto'
python is /usr/bin/python
python is /usr/local/bin/python
```

type = pokaż, czym jest dane słowo dla powłoki: poleceniem, aliasem czy funkcją

TYLKO ODCZYT

Sprawdzenie komendy w skryptach: command -v

```
command -v ls
```

Robimy to samo co `type`, ale w formacie do parsowania w skryptach — dostajemy samą ścieżkę bez opisowego zdania. Jest częścią standardu POSIX, więc działa identycznie we wszystkich zgodnych powłokach.

Przykład:

```
/usr/bin/ls
```

POSIX = Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach · **command** = uruchom polecenie z pominięciem aliasów i funkcji powłoki o tej samej nazwie

TYLKO ODCZYT

Odnalezienie binarki, strony podręcznika i źródeł: whereis

```
whereis ls
```

Szukamy binarki, strony `man` i plików źródłowych w ustalonym zestawie standardowych katalogów, nie w całym `PATH`. Przydaje się, gdy chcemy sprawdzić, czy dla programu istnieje strona podręcznika, bez wywoływania `man`.

Przykład:

```
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz
```

□□ TYLKO ODCZYT

Wyszukanie plików nowoczesnym odpowiednikiem find: fd

```
fd nginx
```

Szukamy „nginx" jako podciągu nazwy, rekurencyjnie od bieżącego katalogu w dół — bez potrzeby pisania `-name "*nginx*"` jak w `find`.

Przykład:

```
etc/nginx/nginx.conf
etc/nginx/conf.d/ssl.conf
```

□□ **fd** = szybszy i prostszy w składni zamiennik find

□□ TYLKO ODCZYT

Uwzględnienie plików i katalogów ukrytych: fd -H

```
fd -H nginx
```

`fd` domyślnie pomija wszystko, co zaczyna się od kropki. `-H` włącza też pliki i katalogi ukryte do przeszukania.

□□ **fd** = szybszy i prostszy w składni zamiennik find

□□ TYLKO ODCZYT

Filtrowanie po rozszerzeniu: fd -e

```
fd -e conf
```

Ograniczamy wynik do plików z podanym rozszerzeniem — krócej niż odpowiadający wzorzec glob w `find`.

fd = szybszy i prostszy w składni zamiennik find

OSTROŻNIE · NIEODWRACALNE

Wykonanie komendy na wynikach: fd -x

```
fd -e log -x rm
fd -e log -X ls -la
```

`-x` uruchamia podaną komendę osobno dla każdego znalezionego pliku (jak `find -exec {} \;`), `-X` (wielka litera) uruchamia ją raz, ze wszystkimi wynikami naraz (jak `find -exec {} +`).

fd = szybszy i prostszy w składni zamiennik find

TYLKO ODCZYT

Wyszukanie treści nowoczesnym odpowiednikiem grep -r: rg

```
rg "connection refused"
```

Szukamy rekurencyjnie od bieżącego katalogu, domyślnie z numerami linii i pomijaniem `.gitignore`/plików ukrytych. Wynik grupujemy po pliku: nazwa pliku jako nagłówek wyróżniony pogrubieniem i kolorem, pod nią numer linii na zielono, a sam dopasowany fragment tekstu podświetlony na czerwono w obrębie linii.

Przykład:

```
app.log
12:2026-09-17 08:12:03 error: connection refused
145:2026-09-17 09:41:07 error: connection refused
```

TYLKO ODCZYT

Pominięcie .gitignore i plików ukrytych: rg -uu

```
rg -uu "TODO"
```

Podwójne `-u` wyłącza jednocześnie respektowanie `.gitignore` i pomijanie plików ukrytych — przeszukujemy dosłownie wszystko pod bieżącym katalogiem, tak jak zrobiłby to zwykły `grep -r`.

◫◫ TYLKO ODCZYT

Filtrowanie po typie pliku: rg --type

```
rg --type py "def main"
```

Rozpoznajemy typ pliku po rozszerzeniu i nazwie bez ręcznego `--include` — `--type py` (albo skrót `-tpy`) ogranicza wyszukiwanie do plików Pythona.

◫◫ TYLKO ODCZYT

Wszystko razem

Narzędzie	Co robi	Kiedy używasz
`locate`	szuka nazwy pliku w gotowej bazie danych	masz pewność, że plik istniał wczoraj, i szukasz po nazwie
`updatedb`	przebudowuje bazę dla `locate`	po masowej operacji na plikach, zanim zaufasz wynikom `locate`
`which`	szuka pliku wykonywalnego w `PATH`	sprawdzasz, którą binarkę uruchomi powłoka
`type` / `type -a`	mówi, czym jest komenda: alias, funkcja, builtin, plik	właściwe pierwsze pytanie w bashu — patrz *Anatomia komendy*
`command -v`	jak `type`, ale w formie do użycia w skryptach	test w `if command -v docker >/dev/null`
`whereis`	binarka + man + źródła w standardowych ścieżkach	szukasz strony podręcznika albo źródeł pakietu
`fd`	rekurencyjne wyszukiwanie plików po nazwie	szybka alternatywa dla `find` na stacji roboczej
`fd -H` / `fd -I`	włącza ukryte / ignoruje `.gitignore`	plik istnieje, ale `fd` go domyślnie pomija
`fd -e`	filtr po rozszerzeniu pliku	szukasz tylko `.conf`, `.log` itp.
`fd -x` / `fd -X`	uruchamia komendę na wyniku (osobno / razem)	dalsza obróbka znalezionych plików
`ripgrep` (`rg`)	rekurencyjne wyszukiwanie treści	szybka alternatywa dla `grep -r` w repozytorium kodu

Narzędzie	Co robi	Kiedy używasz
`rg -uu`	ignoruje `.gitignore` oraz pliki ukryte	plik istnieje, ale `rg` go domyślnie pomija
`rg --type`	filtr po rozpoznanym typie pliku	zamiast ręcznego `--include='*.py'`

Uważaj

- `locate` pokaże plik usunięty wczoraj i nie pokaże pliku utworzonego minutę temu — baza jest tak aktualna, jak ostatnie uruchomienie `updatedb`.
- `fd` i `rg` domyślnie pomijają pliki ukryte i wszystko, co pasuje do `.gitignore` — jeśli szukasz czegoś i „nie ma", sprawdź `-H` / `-I` (fd) albo `-uu` (rg), zanim uznasz, że plik zniknął.
- Nazwa pakietu i nazwa komendy bywają różne między dystrybucjami: pakiet `mlocate` daje komendę `locate`, pakiet `fd-find` na Debianie/Ubuntu daje komendę `fdfind`, nie `fd`. Szczegóły niżej.

Z życia wzięte

Z ŻYCIA WZIĘTE

Poniedziałek, 8:15. Tomek dostał od Oli plik konfiguracyjny na `ci01` pięć minut temu i odpala `locate compose.yaml`, żeby szybko sprawdzić jego pełną ścieżkę. Wynik jest pusty, choć plik na pewno tam jest, więc chwilę się denerwuje, zanim przypomina sobie, że `updatedb` odpala się raz na dobę przez systemd timer. `sudo updatedb` ręcznie i `locate` w końcu go widzi.

Po co to istnieje

`find / -name "nginx.conf"` na serwerze z milionami plików i dyskiem sieciowym potrafi trwać minutę i mocniej obciążyć I/O, niż byś chciał — bo `find` fizycznie odwiedza każdy katalog na ścieżce przeszukiwania. Jeśli szukasz czegoś po nazwie i odpowiedź nie musi być aktualna co do sekundy, `locate` odpowie w milisekundach, bo nie dotyka dysku z plikami — czyta tylko wcześniej przygotowany indeks.

Osobny problem: „skąd wiem, którą komendę faktycznie uruchomi moja powłoka, skoro mam alias, funkcję i trzy wersje `python` w różnych katalogach `PATH`?". Do tego służą `which`, `type`, `command -v` i `whereis` — każde odpowiada na inne pytanie, co jest źródłem częstych nieporozumień.

Trzeci wątek to `fd` i `ripgrep` — narzędzia napisane od nowa (w Rust), gdzie domyślne zachowanie ma być tym, czego programista najczęściej chce: pomijaj `.git`, pomijaj pliki ukryte, licz się z rdzeniami procesora. Kosztem jest to, że nie zawsze są zainstalowane, więc w skrypcie jeżdżącym po nieznanych serwerach `find`/`grep` wciąż są bezpieczniejszym wyborem.

Analogia

Pomyśl o różnicy między **katalogiem bibliotecznym a chodzeniem między regałami**.

Katalog (dawniej szuflada z fiszkami, dziś terminal) odpowiada natychmiast: wpisujesz tytuł, dostajesz numer półki. Nie musisz przejść całej biblioteki. Ale katalog odzwierciedla stan **ostatniej inwentaryzacji** — jeśli ktoś zwrócił książkę pięć minut temu i bibliotekarz jeszcze nie zdążył jej zeskanować z powrotem, katalog powie „wypożyczona", choć fizycznie książka stoi na półce.

To jest dokładnie `locate` — szybki, ale patrzący w przeszłość o tyle, ile czasu minęło od ostatniego przebudowania bazy przez `updatedb`.

Chodzenie między regałami to `find`. Wolniejsze, bo fizycznie sprawdzasz każdą półkę, ale zawsze zgodne z rzeczywistością w chwili sprawdzania — bo *jesteś* tą rzeczywistością, nie czyimś wcześniejszym zapisem o niej.

Jak to działa

Rozwiń: jak to działa pod spodem

> **Pojęcie: indeks (baza) locate.** Wcześniej zbudowany, skompresowany plik ze spisem ścieżek całego systemu plików, który `locate` przeszukuje zamiast dysku. Budowany jest przez `updatedb` — to jego jedyne źródło wiedzy. Plik leży zwykle w `/var/lib/plocate/plocate.db` albo `/var/lib/mlocate/mlocate.db`, w zależności od zainstalowanej implementacji. `locate` nie wie niczego o pliku, który powstał po ostatnim uruchomieniu `updatedb` — z tego bierze się główna pułapka tego narzędzia.



Droga od systemu plików przez updatedb i bazę do locate, z pułapką świeżo utworzonych plików (kliknięcie otwiera pełny rozmiar)

`locate` i baza danych. Baza jest budowana przez `updatedb`, które przechodzi cały system plików (z wyjątkami z `/etc/updatedb.conf` — tam wpisuje się katalogi do pominięcia, np. `/tmp`, `/proc`, montowania sieciowe) i zapisuje listę ścieżek w skompresowanym pliku. Współczesna implementacja to `plocate` — następca starszego `mlocate`, szybszy dzięki innemu formatowi indeksu. Obie respektują uprawnienia: `locate` nie pokaże ścieżki do pliku w katalogu, do którego nie masz dostępu odczytu, nawet jeśli baza tę ścieżkę zawiera — sprawdzenie odbywa się w momencie zapytania.

Odświeżanie bazy odbywa się cyklicznie przez **systemd timer** (nie `cron`, jak bywało dawniej) — zwykle raz na dobę. Na systemach z `plocate` to `plocate-updatedb.timer`, na starszych z `mlocate` — `mlocate.timer` albo wpis w `/etc/cron.daily/`. Gdy potrzebujesz świeżych danych od razu, po dużej operacji na plikach uruchamiasz `sudo updatedb` ręcznie.

Nazwy pakietów między dystrybucjami różnią się i to jest miejsce, gdzie warto sprawdzić aktualny stan, bo zmienił się w ostatnich latach:

- Fedora przeszła na `plocate` jako domyślną implementację `locate` od wersji 36. RHEL/Rocky/AlmaLinux 9 wciąż domyślnie instalują starszy `mlocate` z repozytoriów bazowych — `plocate` jest tam dostępne przez EPEL. RHEL/Rocky 10, oparte na nowszej linii Fedory, idą w stronę `plocate` jako implementacji domyślnej — sprawdź na konkretnym hoście przez `rpm -q mlocate plocate`.
- Debian i Ubuntu: od Ubuntu 22.04 domyślną implementacją jest `plocate` (`mlocate` traktowany jako przestarzały), Debian oferuje oba, z `plocate` jako zalecanym.
- SLES 15: `mlocate` dostępne przez `zypper install mlocate`, `plocate` w oficjalnych repozytoriach SUSE historycznie nie było standardem — sprawdź `zypper search plocate` przed poleganiem na nim.

Komenda `locate` działa identycznie niezależnie od tego, która implementacja stoi pod spodem — różnice dotyczą tylko wydajności i nazwy pliku bazy.

`which` vs `type` vs `command -v` vs `whereis`. To są cztery różne narzędzia odpowiadające na pokrewne, ale nietożsame pytania:

- `which` to zewnętrzny program, który przeszukuje katalogi z `PATH` w poszukiwaniu pliku wykonywalnego. Nie wie nic o aliasach ani funkcjach powłoki, bo to twory samej powłoki, niewidoczne dla procesu zewnętrznego. Na części instalacji minimalnych Debiana/Ubuntu bywa nawet pominięte na rzecz wbudowanego `command -v` — nie zakładaj jego obecności w kontenerze.
- `type` to **wbudowana komenda powłoki** (bash builtin), więc widzi wszystko, co powłoka widzi: aliasy, funkcje, słowa kluczowe, builtiny i dopiero na końcu pliki z `PATH`. To dlatego `type` jest właściwą odpowiedzią w bashu na pytanie „co się wykona, gdy

wpiszę tę nazwę" — temat rozwinięty na stronie *Anatomia komendy: polecenie, opcje, argumenty*.
- `command -v` robi to samo co `type`, ale w formacie do parsowania w skryptach — zwraca samą ścieżkę bez opisowego zdania, i jest częścią standardu POSIX, więc działa identycznie we wszystkich zgodnych powłokach.
- `whereis` ma inny cel: szuka binarki, strony podręcznika i plików źródłowych w ustalonym zestawie standardowych katalogów (nie w całym `PATH`). Przydaje się, gdy chcesz sprawdzić, czy dla programu istnieje strona `man`, bez wywoływania `man`.

`fd` i `ripgrep`. Oba napisane w Rust, oba stawiają na domyślne zachowanie dopasowane do pracy z kodem (pomijanie `.git`, `.gitignore`, plików ukrytych) i na przeszukiwanie równoległe z użyciem wielu rdzeni, co przy dużych drzewach daje odczuwalną przewagę nad jednowątkowymi `find`/`grep`. `fd` szuka po nazwie (dopasowanie podciągu, nie pełny glob jak `find -name`), pomija ukryte i zignorowane ścieżki. `ripgrep` robi to samo dla treści, z dodatkowym atutem: rozpoznaje typ pliku po rozszerzeniu (`--type py`) bez ręcznego `--include`.

Krok po kroku

1. Znajdź plik po nazwie, jeśli wiesz, że gdzieś na dysku istnieje.

```
locate nginx.conf
```

```
/etc/nginx/nginx.conf
/usr/share/doc/nginx/examples/nginx.conf
```

2. Sprawdź, czy plik utworzony przed chwilą już jest w bazie.

```
cd /tmp
touch swiezy_plik_testowy.txt
locate swiezy_plik_testowy.txt
```

```
(brak wyniku)
```

Baza nie zna pliku, bo `updatedb` jeszcze nie przeszedł od czasu jego utworzenia.

3. Odśwież bazę ręcznie i powtórz.

```
sudo updatedb
locate swiezy_plik_testowy.txt
```

```
/tmp/swiezy_plik_testowy.txt
```

4. Sprawdź, co się właściwie wykona pod nazwą `python`.

```
type -a python
```

```
python is /usr/bin/python
python is /usr/local/bin/python
```

`-a` (all) pokazuje wszystkie dopasowania w kolejności, w jakiej `PATH` je przeszukuje — pierwsze jest tym, które faktycznie się uruchomi.

5. Znajdź wszystkie pliki `.log` w drzewie katalogów, pomijając ukryte.

```
fd -e log /var/log
```

6. Poszukaj wzorca w kodzie, tylko w plikach Pythona, z kontekstem.

```
rg -tpy -C 2 "def main"
```

Częste błędy

`locate` nie znajduje pliku utworzonego przed chwilą.

```
touch /tmp/raport-2026-09-17.csv
locate raport-2026-09-17
```

```
(brak wyniku)
```

To nie błąd narzędzia — baza jeszcze nie wie o tym pliku. Rozwiązanie na już: `sudo updatedb`. Jeśli regularnie potrzebujesz świeżych wyników zaraz po utworzeniu plików, `locate` nie jest właściwym narzędziem — użyj `find` bezpośrednio.

Przekonanie, że `which` widzi alias.

```
alias ls='ls --color=auto'
which ls
```

```
/usr/bin/ls
```

`which` pokazuje binarkę z `PATH`, ignorując istnienie aliasu — bo alias jest tworem powłoki, a `which` to osobny proces bez wglądu w jej wewnętrzny stan. Żeby zobaczyć, że `ls` to w rzeczywistości `ls --color=auto`:

```
type ls
```

```
ls is aliased to `ls --color=auto'
```

Instalacja niewłaściwego pakietu przez nieuwagę na nazwę dystrybucyjną.

Skrypt napisany i przetestowany na Fedorze/RHEL (gdzie pakiet z EPEL instaluje komendę `fd` wprost) wywołujący po prostu `fd`, na świeżym Ubuntu skończy się `fd: command not found`. Powód: pakiet `fd-find` na Debianie/Ubuntu instaluje komendę pod nazwą `fdfind`, nie `fd` — to konflikt nazwy z innym, starszym pakietem w archiwum. Obejście: alias albo dowiązanie symboliczne:

```
mkdir -p ~/.local/bin
ln -s $(which fdfind) ~/.local/bin/fd
```

Zdziwienie, że `fd`/`rg` „nie widzi" pliku, który `find`/`grep` widzą bez problemu.

```
fd config .
```

```
(brak wyniku, choć plik .config/app.conf istnieje)
```

`fd` domyślnie pomija pliki i katalogi ukryte oraz wszystko wykluczone przez `.gitignore`. Ma to sens w repozytorium kodu, ale bywa mylące poza nim. Poprawka:

```
fd -H -I config .
```

`-H` włącza pliki ukryte, `-I` ignoruje reguły `.gitignore`.

□□ Ćwiczenia

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

Ćwiczenie 1 — opóźnienie `locate` na własne oczy

```
cd /tmp
mkdir -p test_locate
touch test_locate/plik_testowy_$(date +%s).txt
locate plik_testowy
```

Sprawdź, czy wynik jest pusty (prawdopodobnie tak, jeśli `updatedb` nie uruchamiał się od utworzenia pliku), potem:

```
sudo updatedb
locate plik_testowy
```

Ćwiczenie 2 — `which`, `type` i `command -v` na tej samej nazwie

```
type cd
which cd 2>&1
command -v cd
```

Zastanów się, dlaczego `which cd` daje inny wynik (albo błąd, albo nic) niż pozostałe dwa.

Odpowiedź

`cd` jest wbudowaną komendą powłoki (shell builtin) — nie istnieje jako osobny plik wykonywalny w `PATH`. `type cd` i `command -v cd` wiedzą o tym, bo działają w kontekście powłoki i widzą jej listę builtinów. `which cd` szuka pliku w `PATH`, nic nie znajduje i zwraca błąd. To dlatego `type` jest właściwym narzędziem w bashu.

Ćwiczenie 3 — `fd` względem plików ukrytych

```
cd /tmp
mkdir -p projekt_test && cd projekt_test
touch .env config.yaml
fd .                    # ile plików widzisz?
fd -H .                 # a teraz?
```

Pytania kontrolne

1. `locate raport.csv` zwraca pustą odpowiedź, choć na pewno widziałeś ten plik dzisiaj rano w `ls`. Co sprawdzisz jako pierwsze?

Odpowiedź

Czy `updatedb` uruchamiał się od czasu utworzenia pliku (albo od czasu, gdy plik trafił do katalogu, którego `locate` w ogóle indeksuje — sprawdź `/etc/updatedb.conf` pod kątem wykluczonych ścieżek). `locate` nie patrzy na żywy system plików, tylko na wcześniej zbudowaną bazę, więc świeży plik po prostu może w niej jeszcze nie istnieć. Rozwiązanie doraźne: `sudo updatedb`, potem ponów zapytanie.

2. Dlaczego w skrypcie powłoki, który ma sprawdzić, czy dostępna jest komenda `docker`, lepiej użyć `command -v docker` niż `which docker`?

Odpowiedź

`command -v` jest częścią standardu POSIX i wbudowana w powłokę, więc działa identycznie na każdym systemie zgodnym z POSIX, nawet jeśli zewnętrzny `which` nie jest zainstalowany (zdarza się w minimalnych obrazach kontenerowych). Typowy wzorzec: `if command -v docker >/dev/null 2>&1; then ...`.

3. Masz repozytorium Gita z katalogiem `build/` wpisanym do `.gitignore`, w którym leżą wygenerowane pliki `.log`. `rg "error" .` nie znajduje w nich niczego, mimo że błędy tam są. Dlaczego i jak to obejść?

Odpowiedź

`ripgrep` domyślnie respektuje `.gitignore` — traktuje wykluczone ścieżki tak, jakby ich nie było, żeby nie przeszukiwać artefaktów budowania. Żeby mimo to przeszukać `build/`: `rg -uu "error" .` (podwójne `-u` wyłącza reguły ignorowania i pomijanie plików ukrytych) albo węziej: `rg --no-ignore "error" build/`.

4. Piszesz skrypt instalacyjny, który ma działać zarówno na serwerach RHEL, jak i na stacjach z Ubuntu, i chcesz w nim użyć `fd` do wyszukiwania plików konfiguracyjnych. Czy to dobry pomysł?

Odpowiedź

Ryzykowny. `fd` nie jest domyślnie zainstalowane na żadnym z tych systemów, nazwa komendy różni się między dystrybucjami (`fdfind` na Debianie/Ubuntu), a na minimalnym serwerze może być niedostępne bez dodatkowego repozytorium (EPEL na RHEL). W skrypcie jeżdżącym po nieznanych serwerach bezpieczniejszy jest przenośny `find` — działa wszędzie bez doinstalowywania czegokolwiek. `fd`/`rg` mają sens na własnej stacji roboczej albo tam, gdzie masz pełną kontrolę nad środowiskiem.

Powiązane: Anatomia komendy: polecenie, opcje, argumenty · grep: wyszukiwanie w treści plików · Jak czytać dokumentację: man, --help, apropos, info, tldr · System plików, ścieżki, pwd, cd, ls

Źródła: `man 1 locate` · `man 1 updatedb` · `man 5 updatedb.conf` · `man 1 type` (bash builtin) · `man 1 which` · `man 1 whereis` · Fedora — Plocate as the default locate implementation · plocate, a much faster locate · Debian — pakiet fd-find · sharkdp/fd — dokumentacja projektu · BurntSushi/ripgrep — dokumentacja projektu

---

Revision #18
Created 2026-09-17 15:06:56 UTC by Tomasz Konieczny
Updated 2026-10-03 19:07:56 UTC by Tomasz Konieczny