jq: JSON w powłoce

◫◫ Poziom: średni **◫◫ Dotyczy: wszystkie dystrybucje** **⏱ Czas: ~18 min**

`jq` to `sed`/`awk` dla JSON-a — parsuje strukturę zamiast dopasowywać tekst, więc odpowiedź API, `docker inspect` czy `ip -j addr` da się bezpiecznie odpytać jednym poleceniem, zamiast liczyć na to, że `grep` trafi w klucz w odpowiednim miejscu.

◫◫ Skróty użyte na tej stronie (9)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CI	Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie
CSV	w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami
EPEL	Extra Packages for Enterprise Linux: dodatkowe repozytorium pakietów dla RHEL i klonów
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
SQL	Structured Query Language: język zapytań do relacyjnych baz danych
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

> **Pojęcie: JSON jako drzewo, filtr jako ścieżka.** Dokument JSON to drzewo zagnieżdżonych wartości: obiekty i tablice to węzły, które rozgałęziają się w głąb, a stringi, liczby, `true`/`false`/`null` to liście na końcach gałęzi. Filtr jq to zapis ścieżki w tym drzewie, dokładnie jak ścieżka do pliku w katalogach — `.uzytkownicy[0].imie` znaczy „wejdź do węzła `uzytkownicy`, zejdź do pierwszego elementu, weź z niego pole `imie`". Każdy filtr poniżej to inny sposób poruszania się po tym samym drzewie: w dół do konkretnego węzła, na boki po wszystkich elementach tablicy, albo z filtrowaniem po warunku po drodze.

Wszystkie poniższe fiszki odpytują ten sam plik:

```
cd /tmp
cat > serwer.json <<'EOF'
{
  "host": "srv01",
  "wersja": "2.4.1",
  "uzytkownicy": [
    {"id": 1, "imie": "Alicja", "aktywny": true, "email": "alicja@ad.zielona.example", "role":
["admin", "user"]},
    {"id": 2, "imie": "Bartek", "aktywny": false, "role": ["user"]}
  ],
  "limity": {
    "pamiec": {"mb": 8192, "swap_mb": 2048}
  }
}
EOF
```

TYLKO ODCZYT

Zwrócenie wejścia bez zmian: `.`

```
jq '.' serwer.json
```

Zwracamy całe wejście bez zmian, tylko sformatowane z wcięciami i podświetlone kolorami. Najprostszy filtr — punkt wyjścia do wszystkich pozostałych.

Przykład:

```
{
  "host": "srv01",
  "wersja": "2.4.1",
  "uzytkownicy": [...],
  "limity": {...}
}
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Wybranie wartości jednego pola: `.klucz`

```
jq '.host' serwer.json
```

Projekcja na jedno pole obiektu najwyższego poziomu.

Przykład:

```
"srv01"
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Rozwinięcie tablicy: `.[]`

```
jq '.uzytkownicy[]' serwer.json
```

Zamieniamy jedną wartość tablicową w strumień jej elementów — każdy obiekt trafia na wyjście osobno, gotowy do dalszego przetwarzania kolejnym filtrem.

Przykład:

```
{
  "id": 1,
  "imie": "Alicja",
  "aktywny": true,
  "email": "alicja@ad.zielona.example",
  "role": ["admin", "user"]
}
{
  "id": 2,
  "imie": "Bartek",
  "aktywny": false,
  "role": ["user"]
}
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Wybór konkretnego elementu tablicy: `.[0]`

```
jq '.uzytkownicy[0]' serwer.json
```

Indeksy liczone od zera; ujemne (`.[-1]`) liczą od końca tablicy. W przeciwieństwie do `.[]`, zwracamy tylko jeden, wskazany element.

Przykład:

```
{
  "id": 1,
  "imie": "Alicja",
  "aktywny": true,
  "email": "alicja@ad.zielona.example",
  "role": ["admin", "user"]
}
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Odczytanie zagnieżdżonej wartości: `.a.b.c`

```
jq '.limity.pamiec.mb' serwer.json
```

Trzy kropki, trzy poziomy zagnieżdżenia — czyta się dokładnie jak ścieżkę do pliku: `limity` → `pamiec` → `mb`.

Przykład:

```
8192
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Uniknięcie błędu przy niezgodności typu: `?`

```
jq '.uzytkownicy[1].email?' serwer.json
```

Bartek nie ma pola `email` w danych, więc odpytanie zwraca `null` — samo w sobie bez błędu. `?` chroni przed sytuacją, w której dalsze zejście w głąb takiej wartości (np. `.email?.domena`) rzuciłoby błędem kończącym pracę zamiast po cichu pominąć rekord.

Przykład:

```
null
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Filtrowanie po warunku: `select()`

```
jq '.uzytkownicy[] | select(.aktywny)' serwer.json
```

Przepuszczamy dalej tylko te wartości, dla których warunek jest prawdziwy — odpowiednik `WHERE` w SQL. Tu zostaje tylko Alicja, bo `.aktywny` Bartka to `false`.

Przykład:

```
{
  "id": 1,
  "imie": "Alicja",
  "aktywny": true,
  "email": "alicja@ad.zielona.example",
  "role": ["admin", "user"]
}
```

SQL = Structured Query Language: język zapytań do relacyjnych baz danych · **jq** = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Przekształcenie każdego elementu tablicy: `map()`

```
jq '.uzytkownicy | map(.imie)' serwer.json
```

Skrót na `[.[] | .imie]`: rozwijamy tablicę, stosujemy filtr do każdego elementu, zbieramy wynik z powrotem w nową tablicę.

Przykład:

```
[
  "Alicja",
```

```
  "Bartek"
]
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Sprawdzenie, czy klucz istnieje: `has()`

```
jq '.uzytkownicy[1] | has("email")' serwer.json
```

Zwracamy `true` / `false` w zależności od tego, czy podany klucz występuje w obiekcie — przydatne zamiast polegania na tym, że brakujący klucz „po cichu" da `null`.

Przykład:

```
false
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Policzenie elementów: `length`

```
jq '.uzytkownicy | length' serwer.json
```

Dla tablicy zwraca liczbę elementów, dla obiektu liczbę kluczy, dla stringa liczbę znaków.

Przykład:

```
2
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Wypisanie kluczy obiektu: `keys`

```
jq '.limity.pamiec | keys' serwer.json
```

Zwracamy tablicę nazw kluczy obiektu, posortowaną alfabetycznie.

Przykład:

```
[
  "mb",
  "swap_mb"
]
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Zamiana obiektu na tablicę par: `to_entries`

```
jq '.limity.pamiec | to_entries' serwer.json
```

Zamieniamy obiekt `{klucz: wartość, ...}` na tablicę obiektów `{"key": klucz, "value": wartość}` — przydatne, gdy trzeba iterować po parach klucz-wartość, a nie tylko po samych wartościach.

Przykład:

```
[
  {"key": "mb", "value": 8192},
  {"key": "swap_mb", "value": 2048}
]
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Wypisanie surowego tekstu bez cudzysłowów: `-r`

```
jq -r '.host' serwer.json
```

Bez `-r` string wychodzi z otaczającymi go cudzysłowami JSON-a (`"srv01"`). Z `-r` — sam tekst, gotowy do podstawienia w powłoce.

Przykład:

```
srv01
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Spłaszczenie wyniku do jednej linii: `-c`

```
jq -c '.uzytkownicy[0]' serwer.json
```

Spłaszczamy wynik do jednej linii, bez wcięć i łamania wierszy — wygodne do logów albo dalszego przetwarzania liniowego.

Przykład:

```
{"id":1,"imie":"Alicja","aktywny":true,"email":"alicja@ad.zielona.example","role":["admi
n","user"]}
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Sformatowanie jako linia CSV: `@csv`

```
jq -r '.uzytkownicy[] | [.id, .imie] | @csv' serwer.json
```

Formatujemy tablicę wartości jako jedną linię CSV, z automatycznym cudzysłowowaniem stringów zawierających przecinek.

Przykład:

```
1,"Alicja"
2,"Bartek"
```

CSV = w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami ·**jq** = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Sformatowanie jako linia TSV: `@tsv`

```
jq -r '.uzytkownicy[] | [.id, .imie] | @tsv' serwer.json
```

To samo co `@csv`, ale pola rozdzielone tabulatorem zamiast przecinka i bez cudzysłowów — wygodniejsze wejście dla `cut` czy `column -t`.

Przykład:

```
1 Alicja
2 Bartek
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Wstrzyknięcie stringa z powłoki: `--arg`

```
szukany="Bartek"
jq --arg imie "$szukany" '.uzytkownicy[] | select(.imie == $imie)' serwer.json
```

Ustawiamy zmienną `$imie` wewnątrz filtra jq, przed jego uruchomieniem — bez interpolacji tekstowej, więc znaki specjalne w `$szukany` nie rozjadą filtra.

Przykład:

```
{
  "id": 2,
  "imie": "Bartek",
  "aktywny": false,
  "role": ["user"]
}
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

ZMIENIA SYSTEM

Wstrzyknięcie liczby (albo innej wartości JSON): `--argjson`

```
jq --argjson min 4096 '.limity.pamiec | select(.mb > $min)' serwer.json
```

`--arg` zawsze zrobiłby z `$min` string, a porównanie liczby ze stringiem (`8192 > "4096"`) skończyłoby się błędem typu. Flagą `--argjson` parsujemy wartość jako prawdziwą liczbę JSON.

Przykład:

```
{
  "mb": 8192,
  "swap_mb": 2048
}
```

JSON = JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami ·**jq** = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Skonstruowanie nowego obiektu

```
jq '{id: .uzytkownicy[0].id, nazwa: .uzytkownicy[0].imie}' serwer.json
```

Składnią `{klucz: wartość}` tworzymy zupełnie nowy obiekt JSON, niezależny od kształtu wejścia — przydatne przy przekształcaniu odpowiedzi API do formatu, którego oczekuje kolejne narzędzie w potoku.

Przykład:

```
{
  "id": 1,
  "nazwa": "Alicja"
}
```

JSON = JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami ·**API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą ·**jq** = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Zwrócenie wartości domyślnej: `//`

```
jq '.uzytkownicy[1].email // "brak_adresu"' serwer.json
```

Gdy lewa strona to `null` (albo `false`), zwracamy prawą stronę zamiast niej. Bartek nie ma pola `email`, więc zamiast `null` wychodzi wartość domyślna.

Przykład:

```
"brak_adresu"
```

jq = JSON query: wycinanie i przekształcanie danych w formacie JSON z linii poleceń

TYLKO ODCZYT

Wszystkie flagi

Filtr / flaga	Co robi	Kiedy używasz
`.a.b.c`	ścieżka do zagnieżdżonej wartości	dokładnie jak ścieżka do pliku w katalogach
`.tab[]`	rozwija tablicę na osobne wartości	dalsze przetwarzanie każdego elementu z osobna
`select(warunek)`	filtruje elementy strumienia	odpowiednik `WHERE` w SQL, `grep` dla struktur
`map(filtr)`	przekształca każdy element tablicy	budowanie nowej tablicy z jednym polem na element
`-r`	wypisuje surowy string, bez cudzysłowów	zawsze, gdy wynik trafia do zmiennej powłoki
`-c`	spłaszcza wynik do jednej linii	logi, dalsze przetwarzanie liniowe
`--arg` / `--argjson`	bezpiecznie wstrzykuje wartość z zewnątrz	zmienna powłoki wewnątrz filtra jq
`//`	wartość domyślna, gdy lewa strona to `null`	dane z API o niepewnej kompletności

Uważaj

- Bez `-r` `jq '.klucz'` na stringu zwraca go **w cudzysłowach** — `if [ "$(jq '.status' r.json)" = "ok" ]` nigdy nie będzie prawdą, bo porównujesz `"ok"` (z cudzysłowami) z `ok`.
- `.pole` na kluczu, którego nie ma, zwraca `null` po cichu — ale `.pole.dalej` na wartości, która jest liczbą albo `null`, rzuca **błędem kończącym pracę** (`Cannot index number with string`), chyba że dodasz `?`.
- `grep`/`sed` na JSON-ie łamią się przy każdej zmianie formatowania (spacje, kolejność kluczy, jedna linia vs wiele) — JSON nie ma gwarantowanego układu tekstu, tylko gwarantowaną strukturę, którą trzeba parsować.

SQL = Structured Query Language: język zapytań do relacyjnych baz danych · **API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **JSON** = JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami

Z życia wzięte

> **Z ŻYCIA WZIĘTE**
>
> O 11:15 Ola pyta, czy kontener z Gitea na docker01 w ogóle ma przypisany adres IP, bo aplikacja CI nie może się z nim połączyć. Zamiast czytać cały zrzut docker inspect, jednym poleceniem jq wyciągamy tylko pole NetworkSettings.IPAddress. Okazuje się, że kontener wisi w sieci none, więc Tomek poprawia compose.yaml i restartuje usługę.

Po co to istnieje

`curl` do API zwraca kilkaset linii zagnieżdżonego JSON-a, a Ciebie interesuje jedna wartość — status ostatniego zadania, adres IP kontenera, lista nazw z tablicy obiektów. Odruch kogoś, kto zna `grep` i `sed`, to spróbować wyłuskać to regexem: `grep -o '"status":"[^"]*"'`. Działa, dopóki ktoś nie doda spacji po dwukropku, nie zmieni kolejności pól, nie zagnieździ tego samego klucza `"status"` głębiej w strukturze albo nie skróci JSON-a do jednej linii. JSON nie ma ustalonego formatowania tekstowego — ma tylko ustaloną **strukturę** (obiekty, tablice, stringi, liczby, `true`/`false`/`null`), a `grep` i `sed` nie wiedzą nic o tej strukturze, tylko o wzorcach znaków.

`jq` parsuje JSON tak, jak parser go rozumie — buduje drzewo, a potem pozwala je odpytywać ścieżkami i filtrami, niezależnie od tego, czy wejściowy plik ma jeden klucz w linii czy wszystko w jednej. To jest różnica jakościowa, nie kosmetyczna: `jq '.status'` zadziała identycznie na JSON-ie sformatowanym ładnie i na JSON-ie z jedną linią i zerem spacji.

Analogia

Pomyśl o **nawigacji po drzewie katalogów, tylko wewnątrz dokumentu**.

Ścieżka `.uzytkownicy[0].adres.miasto` czyta się dokładnie tak, jak ścieżka do pliku: wejdź do `uzytkownicy`, weź pierwszy element (`[0]`), wejdź do jego pola `adres`, z niego weź `miasto`. `cd uzytkownicy && cd 0 && cd adres && cat miasto`, gdyby dało się to zrobić powłoką. Kropka między segmentami pełni tę samą rolę, co ukośnik w ścieżce plikowej — łączy poziomy zagnieżdżenia w jedną, czytelną trasę.

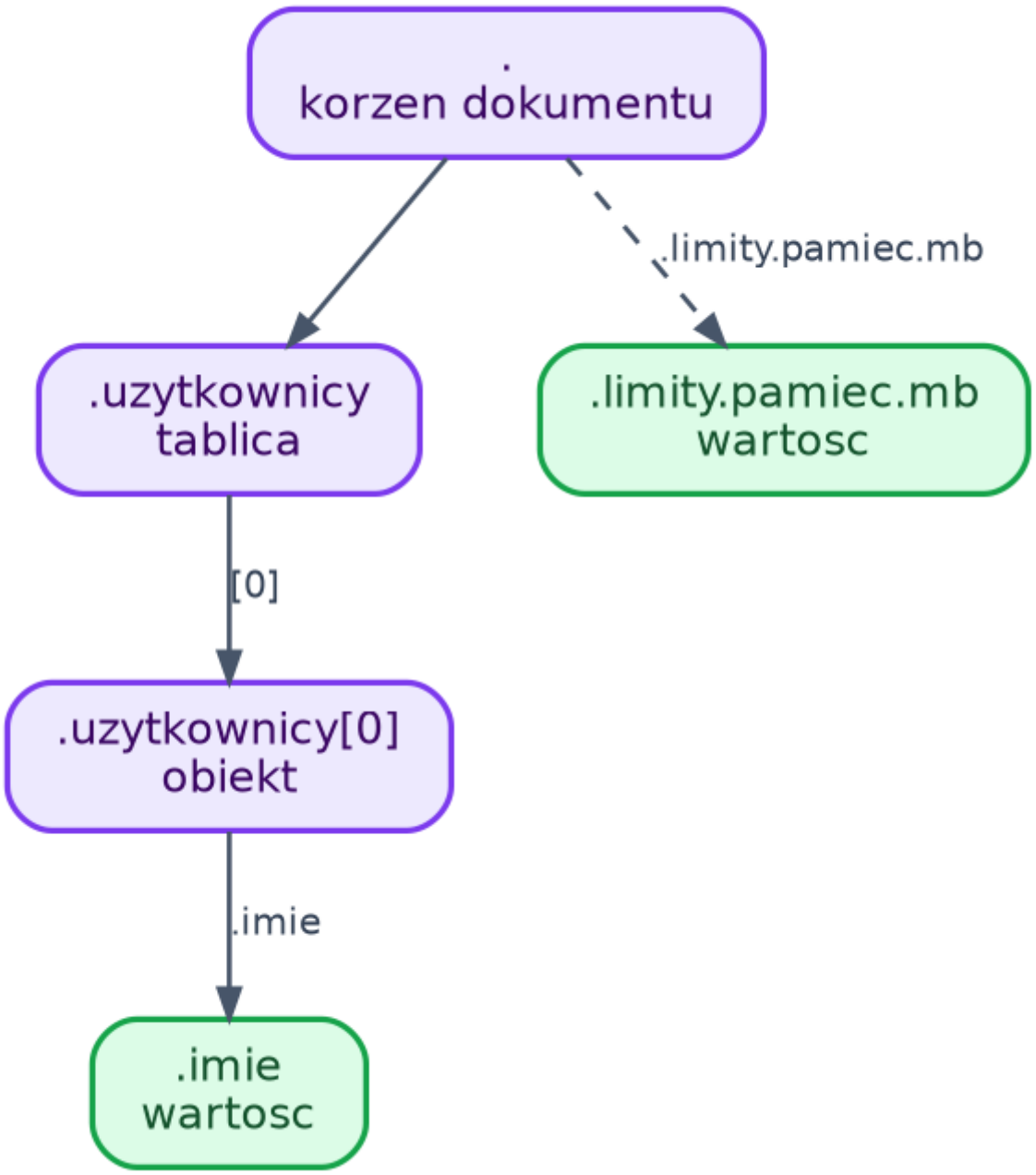
Różnica, która czyni `jq` bardziej użytecznym niż zwykłe `cd`, to `[]` bez indeksu — odpowiednik „wejdź do każdego podkatalogu na raz i zrób to samo dla każdego z osobna". W systemie plików nie ma takiej operacji jednym poleceniem `cd`; w `jq` `.uzytkownicy[]` rozwija całą tablicę w strumień osobnych wartości, z których każdą można dalej odpytywać tym samym filtrem, jednocześnie. To

jest mechanizm, na którym opiera się właściwie każdy nietrywialny filtr w tym narzędziu.

Jak to działa

Rozwiń: jak to działa pod spodem

`jq` czyta wejście jako **strumień wartości JSON** (zwykle jedną, ale może być więcej, jedna po drugiej) i przepuszcza każdą przez podany **filtr**. Filtr to nie polecenie w sensie powłokowym, tylko wyrażenie w małym, funkcyjnym języku — `.` to filtr tożsamościowy (zwróć wejście bez zmian), `.klucz` to projekcja na jedno pole, `|` to potok **wewnątrz** samego jq, przekazujący wynik jednego filtra jako wejście następnego, dokładnie jak `|` powłoki między procesami.



Filtr jq schodzi po drzewie JSON: z korzenia do tablicy, z tablicy do jej elementu, z elementu do pojedynczej wartości (kliknięcie otwiera pełny rozmiar)

Indeksowanie działa dwojako: `.klucz` dla obiektów, `[N]` dla tablic (liczone od zera, ujemne indeksy liczą od końca). `.[]` (albo `.klucz[]`) to operator **rozwinięcia** — zamienia jedną wartość tablicową albo obiektową w strumień jej elementów. To jest różnica, o którą najczęściej się potykają początkujący: `.tablica` zwraca całą tablicę jako jeden JSON, `.tablica[]` zwraca każdy element jako osobną wartość wyjściową.

`select(warunek)` przepuszcza wartość dalej tylko wtedy, gdy warunek jest prawdziwy, i odrzuca ją w przeciwnym razie — działa najczęściej po `.[]`, tworząc idiom `.tablica[] | select(...)`, odpowiednik `WHERE` w zapytaniu SQL. `map(filtr)` to skrót na `[.[] | filtr]` — rozwiń tablicę, zastosuj filtr do każdego elementu, zbierz z powrotem w tablicę.

Błędy kontra `null`. `jq` rozróżnia dwie różne sytuacje, które w innych narzędziach często się zlewają: odpytanie nieistniejącego klucza obiektu (`{"a":1} | .b`) zwraca po cichu `null` — to nie błąd, bo obiekt JSON z definicji może nie mieć danego klucza. Ale próba zejścia głębiej w wartość, która nie jest obiektem ani `null` (np. `.a.b`, gdzie `.a` to liczba albo string) **rzuca błędem i kończy pracę** z niezerowym kodem wyjścia — bo to sygnał, że założenie o strukturze danych było błędne, a nie że brakuje jednego pola. Operator `?` po dowolnym filtrze zamienia taki błąd z powrotem w ciche pominięcie (`empty`), przydatne przy danych o niepewnym, zmiennym kształcie.

Flagi wyjścia. Domyślnie `jq` formatuje wynik z wcięciami i cudzysłowami wokół stringów — czytelne dla człowieka, ale niewygodne do dalszego przetwarzania w powłoce. `-r` (`--raw-output`) wypisuje stringi bez cudzysłowów, `-c` (`--compact-output`) spłaszcza cały JSON do jednej linii. Obie flagi są niezależne i często idą razem: `jq -rc`.

`--arg` kontra `--argjson`. Oba wstrzykują wartość ze zmiennej powłoki do środka filtra jq, dostępną tam jako `$nazwa` — ale `--arg` zawsze traktuje wartość jako **string** (nawet jeśli to `"100"`), a `--argjson` parsuje ją jako **prawdziwy JSON** (liczbę, `true`/`false`, obiekt). Do porównań tekstowych (`select(.nazwa == $p)`) używasz `--arg`, do porównań liczbowych (`select(.rozmiar > $l)`) — `--argjson`, bo `"100" > 50` w JSON-ie jest błędem typu (string kontra liczba), a `100 > 50` już nie.

Krok po kroku

Przykładowy plik na potrzeby tej sekcji:

```
cd /tmp
cat > dane.json <<'EOF'
{
  "uzytkownicy": [
    {"id": 1, "imie": "Alicja", "aktywny": true, "role": ["admin", "user"]},
    {"id": 2, "imie": "Bartek", "aktywny": false, "role": ["user"]}
  ],
  "wersja": "1.0"
}
EOF
```

1. Wyciągnij jedną wartość ze ścieżki.

```
jq '.uzytkownicy[0].imie' dane.json
```

```
"Alicja"
```

Zwrócono JSON-owy string, z cudzysłowami. Do dalszego użycia w powłoce potrzebny jest `-r`:

```
jq -r '.uzytkownicy[0].imie' dane.json
```

```
Alicja
```

2. Rozwiń tablicę i przefiltruj po warunku.

```
jq '.uzytkownicy[] | select(.aktywny) | .imie' dane.json
```

```
"Alicja"
```

`.uzytkownicy[]` rozwija tablicę na dwa osobne obiekty, `select(.aktywny)` przepuszcza tylko ten, gdzie pole `aktywny` jest prawdziwe (`true`, nie `null`, nie `false`), `.imie` bierze z niego jedno pole.

3. Zbuduj nowy obiekt z wybranych pól.

```
jq '{id: .uzytkownicy[0].id, nazwa: .uzytkownicy[0].imie}' dane.json
```

```
{
  "id": 1,
  "nazwa": "Alicja"
}
```

Składnia `{klucz: wartość}` tworzy zupełnie nowy obiekt JSON, niezależny od kształtu wejścia — przydatne przy przekształcaniu odpowiedzi API do formatu, którego oczekuje kolejne narzędzie w potoku.

4. Wypisz jako CSV, gotowe do dalszego przetwarzania.

```
jq -r '.uzytkownicy[] | [.id, .imie] | @csv' dane.json
```

```
1,"Alicja"
2,"Bartek"
```

`@csv` formatuje tablicę wartości jako jedną linię CSV, z automatycznym cudzysłowowaniem stringów zawierających przecinki — bezpieczniejsze niż ręczne sklejanie `"\(.id),\(.imie)"`, bo `@csv` sam obsłuży ewentualny przecinek czy cudzysłów wewnątrz wartości.

5. Wstrzyknij wartość z powłoki bezpiecznie.

```
szukany="Bartek"
jq --arg imie "$szukany" '.uzytkownicy[] | select(.imie == $imie)' dane.json
```

`--arg imie "$szukany"` ustawia zmienną `$imie` wewnątrz filtra jq, PRZED jego uruchomieniem — bez interpolacji tekstowej, więc nawet jeśli `$szukany` zawierałaby cudzysłów czy znak specjalny dla jq, nic się nie rozjedzie. To ten sam mechanizm bezpieczeństwa co `-v` w `awk`, opisany na stronie o `awk`.

6. Porównanie liczbowe wymaga `--argjson`, nie `--arg`.

```
jq --argjson min 400 '.rekordy[] | select(.rozmiar > $min)' dane.json
```

Gdyby użyto `--arg min "400"`, `$min` byłby stringiem `"400"`, a porównanie `.rozmiar > "400"` (liczba kontra string) zakończyłoby się błędem typu w jq. `--argjson` parsuje `"400"` jako liczbę JSON, więc porównanie liczbowe działa poprawnie.

7. Odpytaj żywe API i wyciągnij jedno pole.

```
curl -s https://api.ad.zielona.example/v1/status | jq -r '.dane.status'
```

Typowy wzorzec administracyjny: `curl -s` (cicho, bez paska postępu) zwraca surowy JSON na stdout, `jq -r` wyciąga i odformatowuje jedną wartość, gotową np. do warunku w skrypcie powłoki.

8. Przetwórz `docker inspect` i `ip -j`.

```
docker inspect nazwa_kontenera | jq -r '.[0].NetworkSettings.IPAddress'
ip -j addr show | jq -r '.[] | select(.ifname=="eth0") | .addr_info[0].local'
```

`docker inspect` zwraca **tablicę** z jednym elementem (nawet dla jednego kontenera) — stąd `.[0]` na początku ścieżki. `ip -j` (flaga `-j` wielu nowoczesnych narzędzi administracyjnych — `ip`, `lsblk`, `findmnt` — oznacza „wypisz jako JSON") zwraca tablicę interfejsów; filtr wybiera ten o nazwie `eth0` i schodzi do jego pierwszego adresu.

Częste błędy

Brak `-r` przy podstawianiu wyniku do zmiennej powłoki.

```
status=$(jq '.dane.status' odpowiedz.json)
if [ "$status" = "ok" ]; then
    echo "wszystko w porządku"
fi
```

Warunek nigdy nie będzie prawdziwy, mimo że pole `status` w JSON-ie faktycznie ma wartość `"ok"`. Bez `-r` `jq` zwraca string **razem z otaczającymi go cudzysłowami JSON-a** — `$status` zawiera dosłownie `"ok"` (z cudzysłowami), a nie `ok`. Porównanie `"ok"` z cudzysłowami do `ok` bez nich nigdy się nie zgadza. Poprawka:

```
status=$(jq -r '.dane.status' odpowiedz.json)
```

Zejście w głąb `null`-a albo złego typu bez `?` — skrypt się wywraca.

```
jq '.uzytkownicy[] | .adres.miasto' dane.json
```

```
jq: error (at dane.json:8): Cannot index string with "miasto"
```

Jeśli któryś użytkownik w danych nie ma pola `adres` w ogóle (więc `.adres` to `null`) albo ma je jako sam string zamiast obiektu, próba zejścia głębiej (`.miasto`) kończy się błędem, który **przerywa cały potok**, nie tylko dla tego jednego rekordu — w skrypcie to oznacza brak jakiegokolwiek wyniku i niezerowy kod wyjścia. Rozwiązanie to `?` po niepewnym kroku, ewentualnie w połączeniu z `//` dla wartości domyślnej:

```
jq '.uzytkownicy[] | .adres?.miasto // "brak_adresu"' dane.json
```

Próba użycia `grep`/`sed` na JSON-ie, który zmienia formatowanie.

```
curl -s https://api.ad.zielona.example/status | grep -o '"status":"[a-z]*"'
```

Działa dopóki API zwraca dokładnie taki układ spacji i kolejność pól, na jakim testowano regex. Jedna zmiana po stronie serwera — spacja po dwukropku (`"status": "ok"`), inna kolejność kluczy, albo zagnieżdżenie tego samego klucza `status` głębiej w strukturze (`{"meta":{"status":"..."}}` obok głównego `"status"`) — i regex albo nic nie złapie, albo złapie niewłaściwe wystąpienie. JSON gwarantuje strukturę, nie formatowanie tekstu; `grep`/`sed` czytają wyłącznie tekst. Właściwe narzędzie to `jq` z jawną ścieżką, która wskazuje dokładnie to pole, niezależnie od tego, jak jest sformatowane wokół.

Mieszanie `--arg` z porównaniem liczbowym.

```
limit=100
jq --arg limit "$limit" '.rekordy[] | select(.rozmiar > $limit)' dane.json
```

```
jq: error: object ($limit) and number cannot be compared
```

Wygląda podobnie do poprawnego wzorca z `--arg` z kroku 5, ale tu porównanie jest liczbowe (`>`), a `--arg` zawsze robi z `$limit` string. Trzeba `--argjson`, które parsuje wartość jako liczbę JSON, żeby porównanie `>` miało sens po obu stronach.

□□ Ćwiczenia

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

Ćwiczenie 1 — podstawowe filtry

Przygotuj dane w `/tmp` i przećwicz podstawowe filtry.

```
cd /tmp
cat > serwery.json <<'EOF'
{
  "serwery": [
    {"nazwa": "srv01", "rdzenie": 8, "aktywny": true, "tagi": ["prod", "web"]},
    {"nazwa": "srv02", "rdzenie": 4, "aktywny": false, "tagi": ["dev"]},
    {"nazwa": "storage01.ad.zielona.example", "rdzenie": 16, "aktywny": true, "tagi": ["prod",
"storage"]}
  ]
}
EOF
```

Zadania:

1. Wypisz same nazwy aktywnych serwerów, bez cudzysłowów.
2. Policz, ile jest serwerów z co najmniej 8 rdzeniami.
3. Zbuduj nową, uproszczoną listę obiektów `{host, rdzenie}` tylko dla serwerów z tagiem `"prod"`.
4. Wypisz wynik zadania 3 jako CSV.

Odpowiedź

```
# 1.
jq -r '.serwery[] | select(.aktywny) | .nazwa' serwery.json

# 2.
jq '[.serwery[] | select(.rdzenie >= 8)] | length' serwery.json

# 3.
jq '[.serwery[] | select(.tagi | index("prod")) | {host: .nazwa, rdzenie}]' serwery.json

# 4.
jq -r '.serwery[] | select(.tagi | index("prod")) | [.nazwa, .rdzenie] | @csv'
serwery.json
```

W zadaniu 3 `.tagi | index("prod")` zwraca pozycję elementu `"prod"` w tablicy (liczbę) albo `null`, jeśli go nie ma — a `select()` traktuje `null` jak fałsz, więc filtruje dokładnie tak, jak potrzeba, bez pisania jawnej pętli po tablicy tagów.

Pytania kontrolne

1. Dlaczego `if [ "$(jq '.status' plik.json)" = "ok" ]` prawie nigdy nie jest prawdziwe, nawet gdy pole `status` faktycznie ma wartość `"ok"`?

Odpowiedź

Bo bez flagi `-r` `jq` zwraca string JSON razem z otaczającymi go cudzysłowami — wynik podstawienia to dosłownie `"ok"` z cudzysłowami w środku, a nie sam tekst `ok`. Porównanie do `ok` bez cudzysłowów nigdy się nie zgadza. Poprawka: `jq -r '.status' plik.json`.

2. Czym różni się `.tablica` od `.tablica[]` w filtrze jq?

Odpowiedź

`.tablica` zwraca całą tablicę jako jedną wartość wyjściową (jeden JSON, w nawiasach kwadratowych). `.tablica[]` rozwija tę tablicę na strumień osobnych wartości — każdy element trafia na wyjście oddzielnie i może być dalej przetwarzany przez kolejny filtr w potoku (np. `select()`), z osobna dla każdego elementu.

3. Kiedy używasz `--arg`, a kiedy `--argjson` do przekazania wartości z powłoki?

Odpowiedź

`--arg` zawsze tworzy zmienną typu string wewnątrz filtra jq, niezależnie od tego, jak wygląda wartość — dobre do porównań tekstowych (`select(.nazwa == $x)`). `--argjson` parsuje wartość jako prawdziwy JSON (liczbę, `true`/`false`, obiekt) — potrzebne wszędzie tam, gdzie wartość ma być porównana liczbowo (`select(.rozmiar > $x)`) albo wstrzyknięta jako struktura, nie tekst.

4. Masz dane, w których część obiektów ma pole `adres`, a część go nie ma (wtedy jest `null`). Dlaczego `.uzytkownicy[] | .adres.miasto` może przerwać cały skrypt, a jak temu zapobiec?

Odpowiedź

Odpytanie nieistniejącego klucza (`.adres` gdy go nie ma) samo w sobie zwraca `null` bez błędu — ale próba zejścia głębiej w `null` (`.miasto` na wartości `null`) w jq jest traktowana jako błąd typu i przerywa całe wykonanie z niezerowym kodem wyjścia, nie tylko dla tego jednego rekordu. Zapobiega temu operator `?` po niepewnym kroku (`.adres?.miasto`), ewentualnie razem z `//` dla sensownej wartości domyślnej: `.adres?.miasto // "brak"`.

Powiązane: awk: język do danych w kolumnach · Cudzysłowy, escapowanie i ekspansja · Zmienne środowiskowe i PATH · Pliki, które nie są tekstem: file, strings, hexdump, xxd

Źródła: `man 1 jq` · jqlang.org — jq 1.8.1 (wydanie 2025-07-01) to aktualna stabilna wersja · pakiety: na RHEL 9/10 i Rocky/Alma 9/10 `dnf install jq` wystarcza bez EPEL (Red Hat utrzymuje dla niego własne biuletyny bezpieczeństwa RHSA w standardowych repozytoriach), na starszych RHEL/CentOS 7/8 bywał dostępny tylko przez `dnf install epel-release`; `apt install jq` (Debian/Ubuntu, pakiet w `main`/`universe`); `zypper install jq` (openSUSE) · dla YAML odpowiednikiem `jq` jest `yq`

Revision #19
Created 2026-09-17 15:07:08 UTC by Tomasz Konieczny
Updated 2026-10-03 19:08:07 UTC by Tomasz Konieczny