Architektura Nginx: procesy, konteksty, pliki konfiguracyjne

Poziom: podstawy **Dotyczy: Nginx 1.18+ (dystrybucje RHEL/Rocky, Debian/Ubuntu)**

Czas: ~15 min

Zanim zaczniemy zmieniać cokolwiek w konfiguracji, sprawdzamy z czego Nginx w ogóle się składa — procesów, kontekstów konfiguracji i plików na dysku.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
CPU	Central Processing Unit: procesor
HTTP	Hypertext Transfer Protocol: protokół stron WWW
PID	Process ID: numer procesu w systemie
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
URI	ogólny adres zasobu w internecie, część adresu URL bez samej domeny

Fiszka

TYLKO ODCZYT

Sprawdzenie wersji i zainstalowanych modułów: nginx -V

```
nginx -V
```

Wypisujemy wersję oraz komplet flag `--with-...`, z jakimi binarka została zbudowana — przydaje się, gdy szukamy modułu (np. `http_ssl_module` albo `stream`) i chcemy wiedzieć, czy w ogóle jest dostępny bez przebudowy pakietu.

Przykład:

```
nginx version: nginx/1.24.0
built by gcc 12.2.0
built with OpenSSL 3.0.9
TLS SNI support enabled
configure arguments: --with-http_ssl_module --with-http_v2_module --with-stream --with-
http_realip_module
```

nginx = engine x: serwer WWW i reverse proxy

TYLKO ODCZYT

Sprawdzenie składni konfiguracji: nginx -t

```
sudo nginx -t
```

Prosimy proces o wczytanie i sprawdzenie wszystkich plików konfiguracyjnych bez ich faktycznego zastosowania — sam brak błędu składniowego nie oznacza, że logika reguł jest poprawna, tylko że plik da się w ogóle wczytać. To jest komenda, którą wykonujemy zawsze przed każdym przeładowaniem.

Przykład:

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

nginx = engine x: serwer WWW i reverse proxy

TYLKO ODCZYT

Przeładowanie konfiguracji bez zrywania połączeń: nginx -s reload

```
sudo nginx -s reload
```

Wysyłamy do procesu głównego sygnał, który każe mu wczytać nową konfigurację i uruchomić nowe procesy robocze z tą konfiguracją, jednocześnie dając starym procesom dokończyć już obsługiwane połączenia i dopiero potem się zakończyć. Żadne aktywne żądanie nie zostaje przerwane — to jest różnica względem restartu usługi, który zrywa wszystko natychmiast.

Przykład:

```
$ sudo nginx -s reload
$ sudo systemctl status nginx.service | grep Active
   Active: active (running) since Tue 2026-09-22 08:14:03 UTC; 3 days ago
```

Równoważna komenda serwisowa, robiąca dokładnie to samo poprzez systemd:

```
sudo systemctl reload nginx.service
```

nginx = engine x: serwer WWW i reverse proxy · **systemctl** = system control: sterowanie usługami systemd

TYLKO ODCZYT

Wyświetlenie drzewa procesów: ps

```
ps -eo pid,ppid,cmd --forest | grep nginx
```

Pokazujemy proces główny (master) i jego dzieci (worker) w hierarchii — przydatne do policzenia, ile faktycznie procesów roboczych działa, i do potwierdzenia, że stare procesy po `reload` zniknęły.

Przykład:

```
  1042       1 nginx: master process /usr/sbin/nginx
  1043    1042  \_ nginx: worker process
  1044    1042  \_ nginx: worker process
  1045    1042  \_ nginx: worker process
  1046    1042  \_ nginx: worker process
```

ps = process status: pokaż procesy

Z życia wzięte

Z ŻYCIA WZIĘTE

W środę w południe Tomek zmienia limit `client_max_body_size` na `docker01` i odruchowo wykonuje `systemctl restart nginx.service` zamiast `reload`. Ola akurat wgrywa duży plik

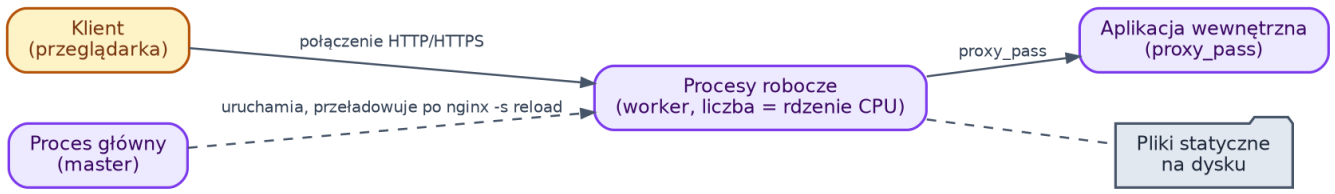
przez aplikację za tym samym nginksem i dostaje zerwane połączenie w połowie transferu. Po tym incydencie Tomek trzyma nad biurkiem karteczkę: zwykła zmiana configu to zawsze `reload`, restart tylko dla zmian wymagających nowego procesu głównego.

Czym jest Nginx

Pojęcie: Nginx. Serwer HTTP i reverse proxy działający w modelu zdarzeniowym — jeden proces potrafi obsłużyć tysiące jednoczesnych połączeń bez tworzenia osobnego wątku systemowego na każde z nich. W naszej infrastrukturze pełni głównie rolę reverse proxy przed aplikacjami wewnętrznymi, opisaną na stronie Reverse proxy i równoważenie obciążenia, rzadziej rolę zwykłego serwera plików statycznych.

Po co to istnieje

Zanim czytamy jakikolwiek plik konfiguracyjny, warto wiedzieć, jakie procesy w ogóle stoją za Nginksem, bo od tego zależy, co znaczy „zmiana działa od razu", a co wymaga przeładowania albo restartu.



Klient łączy się z procesem roboczym, proces główny nim zarządza, a proces roboczy przekazuje żądanie dalej przez proxy_pass albo oddaje plik statyczny (kliknięcie otwiera pełny rozmiar)

Pojęcie: proces główny (master). Jeden proces uruchamiany przy starcie usługi, działający zwykle jako root. Nie obsługuje ruchu HTTP — jego zadaniem jest czytanie konfiguracji, otwieranie gniazd nasłuchu na portach z dyrektywy `listen`, i zarządzanie procesami roboczymi: ich uruchamianiem, zamykaniem oraz przeładowaniem po `nginx -s reload`.

Pojęcie: proces roboczy (worker). Proces potomny procesu głównego, to on faktycznie odbiera połączenia klientów, czyta żądania, woła backend przez `proxy_pass` i odsyła odpowiedź. Liczbę procesów roboczych ustawiamy dyrektywą `worker_processes` w kontekście `main`:

```
worker_processes auto;
```

Pojęcie: worker_processes. Wartość `auto` każe Nginksowi odpalić tyle procesów roboczych, ile system zgłasza rdzeni CPU — to jest wartość, którą zostawiamy w niemal każdej instalacji, bo więcej procesów niż rdzeni nie przyspiesza niczego, tylko zwiększa liczbę przełączeń kontekstu.

Pojęcie: worker_connections. Limit liczby jednoczesnych połączeń, jaki pojedynczy proces roboczy może obsłużyć naraz, ustawiany w kontekście `events`:

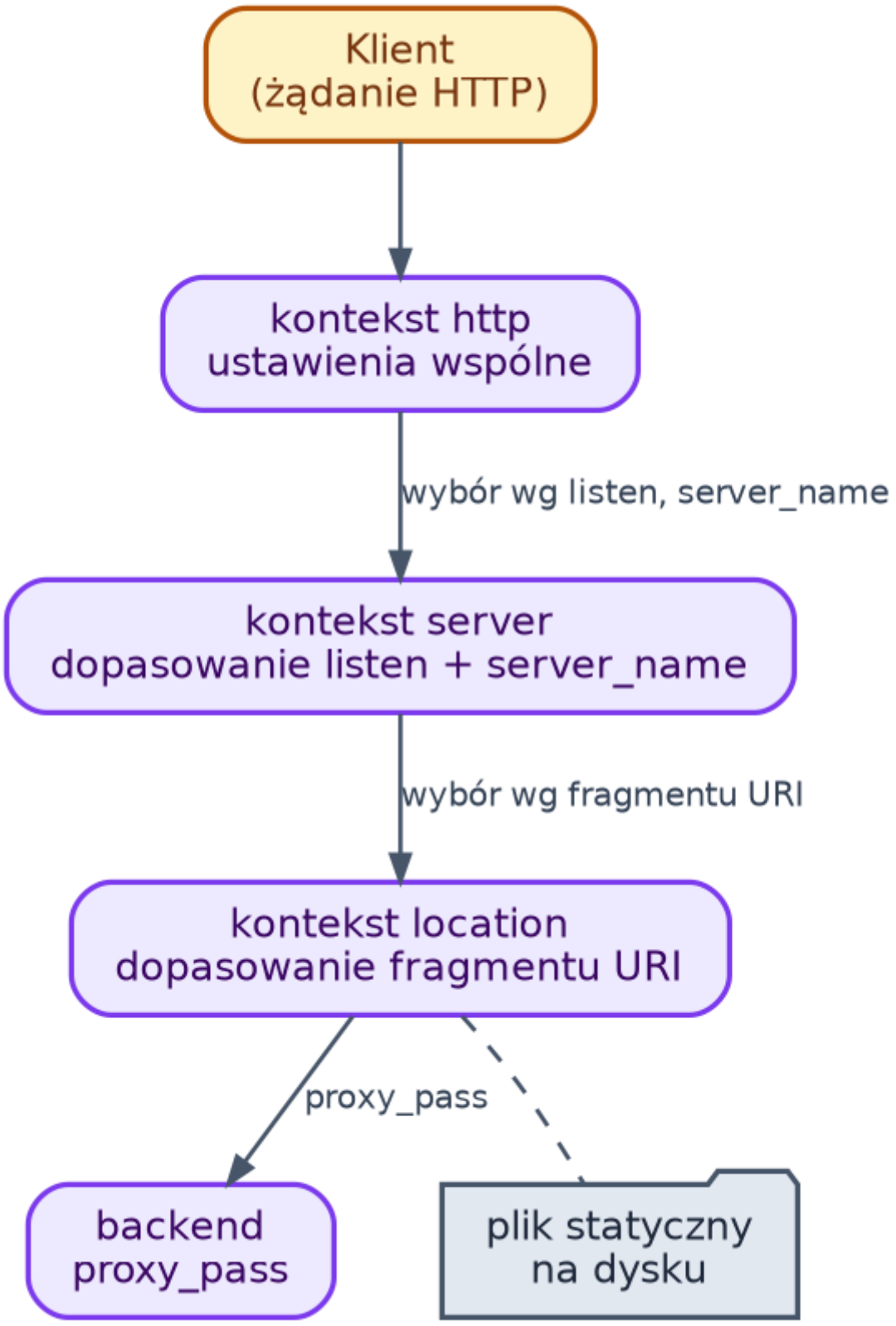
```
events {
    worker_connections 1024;
}
```

Maksymalna teoretyczna liczba jednoczesnych połączeń serwera to iloczyn `worker_processes` razy `worker_connections` — w praktyce mniej, bo część tych połączeń idzie na komunikację z backendem przez `proxy_pass`, więc jedno żądanie klienta potrafi zająć dwa gniazda naraz (jedno do klienta, jedno do backendu).

Jak to działa

Rozwiń: jak to działa pod spodem

Pojęcie: kontekst konfiguracji. Blok w pliku `nginx.conf` ograniczony klamrami `{ }`, w którym obowiązują określone dyrektywy — dyrektywa ustawiona w kontekście nadrzędnym jest dziedziczona przez kontekst zagnieżdżony, chyba że ten ją nadpisze. Struktura kontekstów jest hierarchiczna i zawsze idzie w tej kolejności:


Klient
(żądanie HTTP)
kontekst http
ustawienia wspólne
wybór wg listen, server_name
kontekst server
dopasowanie listen + server_name
wybór wg fragmentu URI
kontekst location
dopasowanie fragmentu URI
proxy_pass
backend
proxy_pass
plik statyczny
na dysku

Żądanie klienta przechodzi kolejno przez kontekst http, potem server dopasowany po listen i server_name, potem location dopasowany po URI, zanim trafi do backendu albo do pliku na dysku (kliknięcie otwiera pełny rozmiar)

```
# kontekst main — poza jakimikolwiek klamrami, ustawienia procesu głównego
user nginx;
worker_processes auto;

events {
    # kontekst events — parametry obsługi połączeń na poziomie procesu roboczego
    worker_connections 1024;
}

http {
    # kontekst http — wszystko wspólne dla ruchu HTTP: typy MIME, logi, gzip, timeouty
domyślne
    include /etc/nginx/mime.types;

    server {
        # kontekst server — jeden wirtualny host, jedna domena albo grupa domen
        listen 80;
        server_name app01.zielona.example;

        location /api/ {
            # kontekst location — reguła dopasowania fragmentu URI wewnątrz danego server
            proxy_pass http://backend_app;
        }
    }
}
```

Każdy z tych kontekstów opisujemy osobno, bo ich odpowiedzialności się nie pokrywają:

- **main** — ustawienia globalne całego procesu: pod jakim userem działają workery, ile ich jest, gdzie leży plik PID.
- **events** — jedyny kontekst dotyczący warstwy połączeń sieciowych, a nie HTTP; tu ląduje `worker_connections` i metoda multipleksowania (`epoll` na Linuksie, dobierana automatycznie).
- **http** — kontener na całą logikę HTTP: ustawienia wspólne dla wszystkich wirtualnych hostów, plik `mime.types`, domyślny `log_format`.

- **server** — jeden wirtualny host, opisany szerzej na stronie Bloki server i location. Może być ich w pliku dowolnie wiele, każdy z innym `server_name` i innym `listen`.
- **location** — reguła wewnątrz `server`, dopasowująca fragment ścieżki URI i decydująca, co z takim żądaniem zrobić — oddać plik statyczny, przekazać dalej przez `proxy_pass`, zwrócić przekierowanie.

Uważaj: dyrektywa wpisana w złym kontekście (np. `proxy_pass` bezpośrednio w `server`, poza jakimkolwiek `location`) albo nie zadziała, albo `nginx -t` zwróci błąd `directive is not allowed here` — to jest najczęstszy powód błędu składni u kogoś, kto kopiuje fragment konfiguracji z internetu bez sprawdzenia, w jakim kontekście oryginalnie się znajdował.

Pliki konfiguracyjne na dysku

Sam mechanizm kontekstów jest identyczny niezależnie od dystrybucji, różni się tylko układ plików na dysku i to, które z nich są wczytywane domyślnie.

Plik / katalog	Rola
`/etc/nginx/nginx.conf`	plik główny — konteksty `main` i `events`, początek `http`, dyrektywy `include` wciągające resztę
`/etc/nginx/conf.d/*.conf`	RHEL/Rocky/CentOS: każdy plik to zwykle jeden lub kilka bloków `server`, wciągane przez `include /etc/nginx/conf.d/*.conf;` w `nginx.conf`
`/etc/nginx/sites-available/*`	Debian/Ubuntu: pełne definicje wirtualnych hostów, jeszcze nieaktywne
`/etc/nginx/sites-enabled/*`	Debian/Ubuntu: dowiązania symboliczne do plików z `sites-available`, tylko to jest faktycznie wczytywane przez `include /etc/nginx/sites-enabled/*;`

Różnica między dystrybucjami sprowadza się do tego, że Debian/Ubuntu rozdziela „zdefiniowane" od „aktywne" osobnym katalogiem i linkiem symbolicznym — wyłączenie strony to usunięcie linku z `sites-enabled`, bez kasowania samej definicji z `sites-available`. RHEL/Rocky nie ma tego rozróżnienia: plik leżący w `conf.d` jest od razu aktywny, wyłączenie strony to przeniesienie pliku poza katalog albo zmiana rozszerzenia tak, żeby wzorzec `*.conf` przestał go łapać.

Włączenie strony na Debianie/Ubuntu:

```
sudo ln -s /etc/nginx/sites-available/NazwaDomeny.conf /etc/nginx/sites-
enabled/NazwaDomeny.conf
```

Po dodaniu, usunięciu albo edycji dowolnego z tych plików kolejność jest zawsze ta sama — sprawdzenie składni, a dopiero potem przeładowanie:

```
sudo nginx -t && sudo systemctl reload nginx.service
```

Częste błędy

Edycja pliku i oczekiwanie efektu bez przeładowania. Nginx nie obserwuje plików konfiguracyjnych na żywo — zmiana leży na dysku i nic nie robi, dopóki nie wykonamy `nginx -s reload` albo `systemctl reload`.

`nginx -t` zwraca `unknown directive`. Zwykle literówka w nazwie dyrektywy albo brak średnika na końcu poprzedniej linii — parser Nginksa zgłasza błąd przy dyrektywie następującej po tej, w której faktycznie brakuje średnika, co bywa mylące przy pierwszym kontakcie.

Restart zamiast reload przy zwykłej zmianie configu. `systemctl restart nginx.service` zamyka wszystkie procesy robocze natychmiast, zrywając aktywne połączenia — dla zwykłej zmiany w `server`/`location` wystarczy `reload`, restart zostawiamy dla sytuacji, w których zmieniamy parametry wymagające ponownego uruchomienia procesu głównego (np. liczbę procesów roboczych po zmianie liczby rdzeni maszyny).

□□ Ćwiczenia

> Wszystko poniżej wykonujemy na własnej maszynie testowej z zainstalowanym Nginksem, najlepiej w kontenerze albo maszynie wirtualnej, którą można bezkarnie popsuć.

Ćwiczenie 1 — znalezienie własnej konfiguracji

```
nginx -V 2>&1 | grep -o -- '--conf-path=\S*'
```

Sprawdzamy, jaka jest domyślna ścieżka do `nginx.conf` w naszej instalacji i porównujemy z tym, co faktycznie widzimy w `/etc/nginx/`.

Odpowiedź

Wynik pokazuje ścieżkę wkompilowaną w binarkę przy budowaniu pakietu — na większości dystrybucji to `/etc/nginx/nginx.conf`, ale niestandardowo zbudowana binarka (np. z kontenera) może mieć zupełnie inną domyślną ścieżkę, np. `/usr/local/nginx/conf/nginx.conf`.

Pytania kontrolne

1. Dlaczego `nginx -s reload` nie zrywa istniejących połączeń, a `systemctl restart` tak?

Odpowiedź

`reload` uruchamia nowe procesy robocze z nową konfiguracją, jednocześnie zostawiając stare procesy robocze przy życiu do momentu, aż dokończą obsługiwane już żądania — dopiero wtedy proces główny je zamyka. `restart` zabija cały proces główny razem ze wszystkimi workerami natychmiast i uruchamia usługę od zera.

2. W jakim kontekście ustawiamy `worker_connections` i dlaczego nie w `http`?

Odpowiedź

W kontekście `events`, bo dotyczy warstwy połączeń sieciowych obsługiwanych przez proces roboczy, a nie logiki HTTP — `events` jest jedynym kontekstem tego typu w całym pliku.

Powiązane: Bloki server i location · Reverse proxy i równoważenie obciążenia · Logi i diagnostyka

Źródła: `man 8 nginx` · dokumentacja nginx.org, rozdział „Beginner's Guide"

Revision #38
Created 2026-09-25 07:40:10 UTC by Tomasz Konieczny
Updated 2026-10-26 14:16:10 UTC by Tomasz Konieczny