Konfiguracja

Konteksty, bloki server i location, reverse proxy oraz TLS.

- Architektura Nginx: procesy, konteksty, pliki konfiguracyjne
- Bloki server i location
- Reverse proxy i równoważenie obciążenia
- TLS w Nginx

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

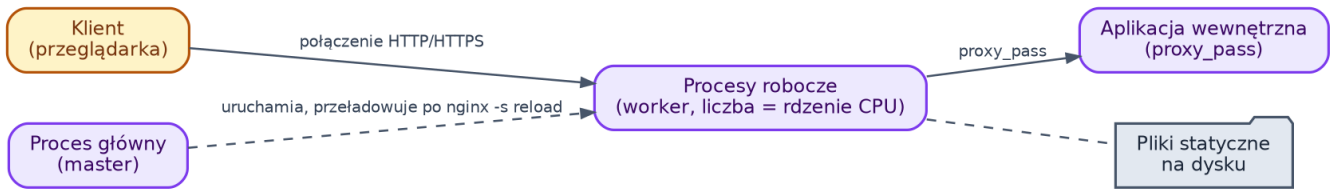
przez aplikację za tym samym nginksem i dostaje zerwane połączenie w połowie transferu. Po tym incydencie Tomek trzyma nad biurkiem karteczkę: zwykła zmiana configu to zawsze `reload`, restart tylko dla zmian wymagających nowego procesu głównego.

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`:

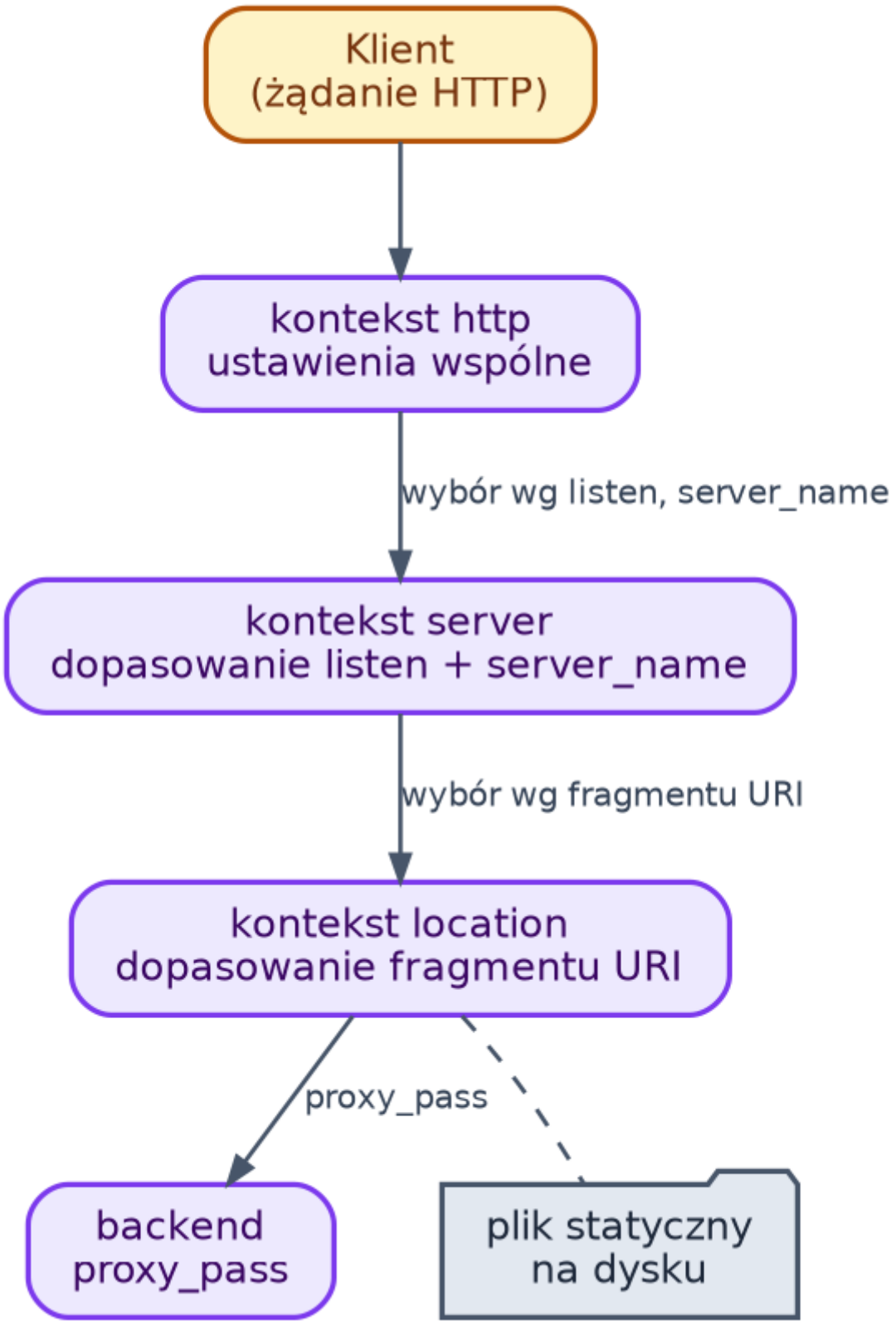
```
events {
    worker_connections 1024;
}
```

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"

Bloki server i location

Poziom: podstawy **Dotyczy: Nginx 1.18+** **Czas: ~20 min**

Kolejność dopasowania `location` jest najczęstszym źródłem nieporozumień przy konfiguracji Nginksa — ta strona rozkłada ją krok po kroku, zamiast zakładać, że jest oczywista.

Skróty użyte na tej stronie (7)

Skrót	Co znaczy
HTTP	Hypertext Transfer Protocol: protokół stron WWW
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
IPv4	Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77
IPv6	Internet Protocol wersja 6: nowszy, dłuższy zapis adresów, w liczbach szesnastkowych oddzielonych dwukropkami
PHP	PHP: Hypertext Preprocessor, język programowania, w którym napisany jest frontend Zabbixa i wiele innych aplikacji webowych
SPA	Single-Page Application: aplikacja WWW działająca w całości w przeglądarce na jednej stronie; w Entra ID osobny typ platformy rejestracji bez sekretu klienta
URI	ogólny adres zasobu w internecie, część adresu URL bez samej domeny

Fiszka

TYLKO ODCZYT

Nasłuch na porcie: listen

```
listen 80;
```

Ustawiamy adres i port, na którym dany blok `server` przyjmuje połączenia. Bez podanego adresu IP Nginx nasłuchuje na wszystkich interfejsach IPv4 maszyny.

□□ **IP** = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **IPv4** = Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77

□□ TYLKO ODCZYT

Nasłuch na konkretnym interfejsie i IPv6

```
listen 10.20.10.42:80;
listen [::]:80;
```

Ograniczamy nasłuch do jednego adresu IPv4, a drugą linią dodajemy osobny nasłuch IPv6 — Nginx nie robi tego automatycznie, IPv4 i IPv6 to dwa osobne gniazda wymagające dwóch osobnych dyrektyw `listen`.

□□ **IPv6** = Internet Protocol wersja 6: nowszy, dłuższy zapis adresów, w liczbach szesnastkowych oddzielonych dwukropkami · **IPv4** = Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77

□□ TYLKO ODCZYT

Wyznaczenie serwera domyślnego: default_server

```
listen 80 default_server;
```

Wskazujemy, który blok `server` ma obsłużyć żądanie trafiające na dany port, gdy nagłówek `Host` z żądania klienta nie pasuje do żadnego skonfigurowanego `server_name` — bez tej flagi Nginx wybiera pierwszy zdefiniowany blok `server` dla danego adresu i portu, co bywa zaskakujące, jeśli kolejność plików w `conf.d` jest przypadkowa.

□□ TYLKO ODCZYT

Dopasowanie nazwy domeny: server_name

```
server_name zielona.example www.zielona.example;
```

Podajemy jedną albo więcej nazw domen, dla których ten blok `server` ma się aktywować — Nginx porównuje je z nagłówkiem `Host` przychodzącego żądania HTTP.

□□ **HTTP** = Hypertext Transfer Protocol: protokół stron WWW

TYLKO ODCZYT

Dopasowanie wieloznacznikiem: server_name z gwiazdką

```
server_name *.zielona.example;
```

Łapiemy dowolną subdomenę pierwszego poziomu — gwiazdka na początku wzorca zastępuje dokładnie jeden segment nazwy, `test.zielona.example` pasuje, `a.test.zielona.example` już nie.

Przykład fragmentu z gwiazdką na końcu, łapiącym warianty domeny:

```
server_name zielona.*;
```

Z życia wzięte

Z ŻYCIA WZIĘTE

We wtorek po świętach Ola dodaje nowy `location ~* \.php$` na `docker01`, żeby obsłużyć nowy panel administracyjny, i wpisuje go na końcu pliku, myśląc że kolejność zapisu decyduje. Regex przejmuje część ruchu z istniejącego `location /app/`, który wcześniej działał bez zarzutu, mimo że nowy blok stoi fizycznie niżej w pliku. Dopiero dopisanie `^~` do starego prefiksu przywraca poprzednie zachowanie, bo wygrywa on z regexem, zamiast czekać na swoją kolej.

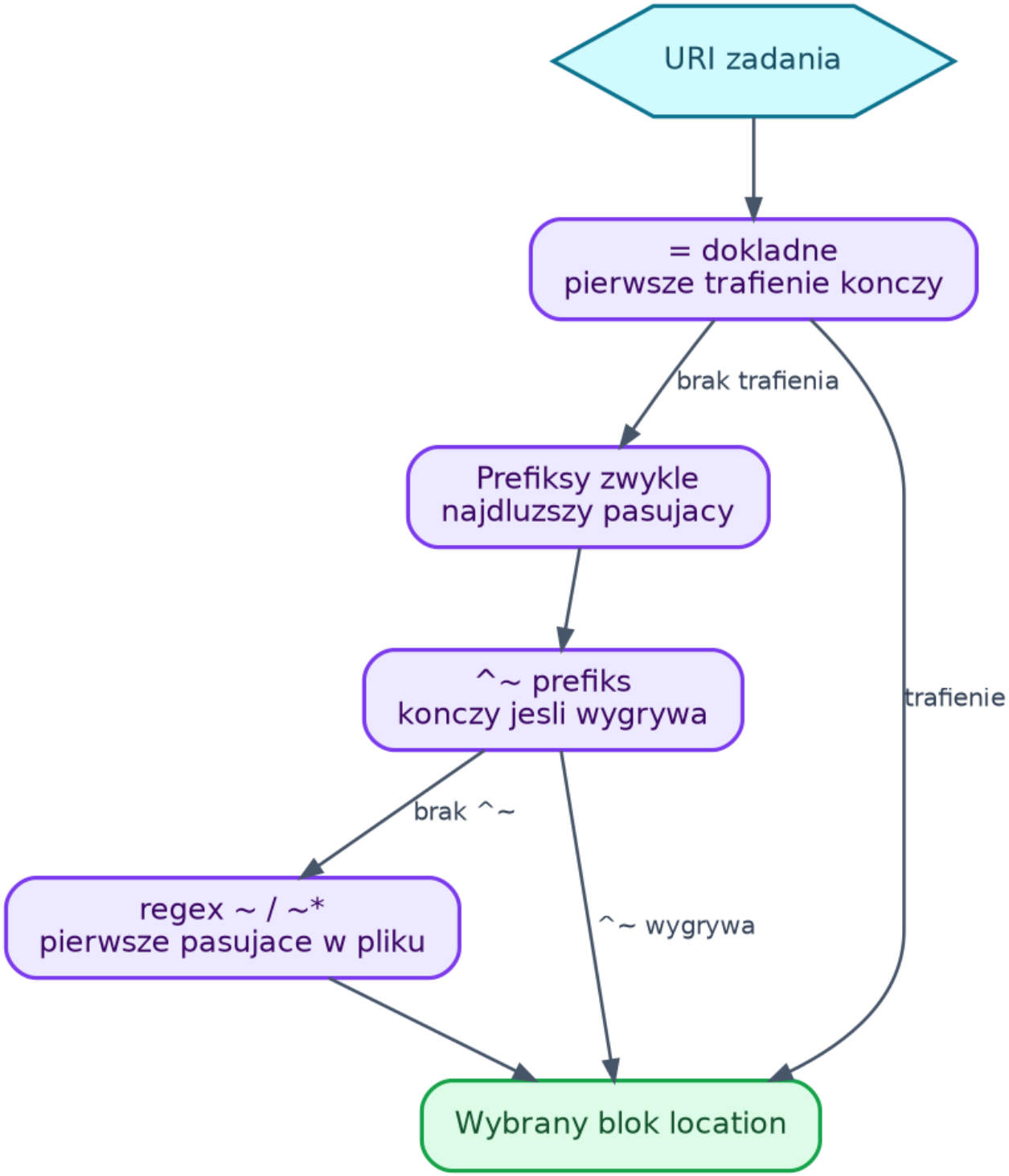
Czym jest blok server

Pojęcie: blok server. Kontekst konfiguracji opisany na stronie Architektura Nginx: procesy, konteksty, pliki konfiguracyjne, reprezentujący jeden wirtualny host — jedną kombinację adresu nasłuchu i nazwy domeny, dla której Nginx wie, co zrobić z przychodzącym żądaniem. Jeden plik konfiguracyjny może zawierać dowolnie wiele bloków `server`, każdy obsługujący inną domenę na tym samym albo innym porcie.

Kolejność dopasowania location krok po kroku

> **Pojęcie: blok location.** Reguła wewnątrz `server`, dopasowująca fragment ścieżki URI żądania i mówiąca, co z takim żądaniem zrobić — zwrócić plik z dysku, przekazać dalej do backendu przez `proxy_pass`, albo wykonać przekierowanie. Jeden `server` zwykle zawiera wiele bloków `location`, po jednym na grupę adresów URI o wspólnej obsłudze.

To, który `location` faktycznie obsłuży dane żądanie, nie zależy od kolejności zapisania bloków w pliku, tylko od typu dopasowania i długości wzorca. Nginx sprawdza reguły w ściśle określonej kolejności:



Nginx sprawdza kolejno dopasowanie dokładne, potem najdłuższy prefiks, potem prefiks z ^~, a dopiero na końcu wyrażenia regularne w kolejności zapisu w pliku (kliknięcie otwiera pełny rozmiar)

1. **Dopasowanie dokładne, prefiks `=`.** Sprawdzane jako pierwsze, zanim cokolwiek innego. Jeśli URI żądania jest identyczne z wzorcem znak w znak, Nginx od razu kończy wyszukiwanie i używa tego bloku, nie sprawdzając już żadnego innego.

```
location = /health {
    return 200 "ok";
}
```

1. **Dopasowania prefiksowe, bez żadnego znaku specjalnego.** Nginx przegląda wszystkie bloki tego typu i zapamiętuje ten o najdłuższym pasującym prefiksie — nie pierwszy napotkany, tylko najdłuższy.

```
location /static/ {
    root /var/www/example;
}
```

1. **Prefiks `^~`.** Też dopasowanie prefiksowe, ale jeśli to ono okaże się najdłuższym pasującym prefiksem spośród wszystkich prefiksowych, Nginx pomija sprawdzanie wyrażeń regularnych i od razu używa tego bloku.

```
location ^~ /static/images/ {
    root /var/www/example;
}
```

1. **Wyrażenia regularne, `~` (uwzględniające wielkość liter) i `~*` (ignorujące wielkość liter).** Sprawdzane dopiero po ustaleniu najlepszego dopasowania prefiksowego z kroków 2–3 — i tylko wtedy, gdy krok 3 nie zakończył wyszukiwania przez `^~`. Wyrażenia regularne sprawdzane są w kolejności zapisania w pliku, używany jest pierwszy pasujący, nie najdłuższy.

```
location ~* \.(jpg|jpeg|png|gif|css|js)$ {
    expires 30d;
}
```

Uważaj: to jest odwrotność intuicji, jaką ma większość osób pierwszy raz konfigurujących Nginksa — prefiksy nie są sprawdzane „od góry pliku do dołu", tylko wybierany jest najdłuższy pasujący, a wyrażenia regularne mają pierwszeństwo przed zwykłymi prefiksami, chyba że jakiś prefiks jest oznaczony `^~`. Skutek: dodanie nowego `location` z wyrażeniem regularnym potrafi przejąć ruch, który wcześniej trafiał do istniejącego prefiksu, nawet jeśli nowy blok zapisaliśmy na końcu pliku.

Pełna kolejność rozstrzygania w jednym zestawieniu:

Krok	Typ	Zasada wyboru w obrębie typu
1	`=` dokładne	pierwsze trafienie kończy wyszukiwanie natychmiast
2	prefiksowe zwykłe	najdłuższy pasujący prefiks

Krok	Typ	Zasada wyboru w obrębie typu
3	`^~` prefiksowe	najdłuższy pasujący prefiks, kończy wyszukiwanie jeśli wygrywa etap 2–3
4	`~` / `~*` regex	pierwsze pasujące wyrażenie w kolejności zapisu w pliku
—	brak location pasującego niczemu innemu	używany jest najdłuższy prefiks zwykły znaleziony w kroku 2, jeśli istnieje

Fiszka — root, alias, index, try_files

Wskazanie katalogu plików: root

```
location /static/ {
    root /var/www/example;
}
```

> **Pojęcie: root.** Do ścieżki URI żądania Nginx dokleja pełną ścieżkę z `root`, zachowując fragment dopasowany przez `location` — żądanie `/static/logo.png` przy powyższej konfiguracji szuka pliku pod `/var/www/example/static/logo.png`.

Wskazanie katalogu plików z podmianą prefiksu: alias

```
location /static/ {
    alias /var/www/example/assets/;
}
```

> **Pojęcie: alias.** W odróżnieniu od `root`, `alias` zastępuje dopasowany prefiks ścieżką z dyrektywy, zamiast go doklejać — to samo żądanie `/static/logo.png` szuka teraz pliku pod `/var/www/example/assets/logo.png`, bez segmentu `static` w ścieżce na dysku.

Uważaj: `alias` w bloku `location` z prefiksem musi kończyć się ukośnikiem, jeśli sam wzorzec `location` kończy się ukośnikiem — pominięcie go jest jednym z najczęstszych powodów błędu 404 przy plikach statycznych, bo Nginx składa ścieżkę bez separatora między katalogami.

Ustalenie pliku domyślnego katalogu: index

```
location / {
    root /var/www/example;
    index index.html index.htm;
}
```

Ustalamy, jakiego pliku Nginx ma szukać, gdy żądanie wskazuje katalog, a nie konkretny plik — sprawdza kolejno wymienione nazwy i zwraca pierwszą istniejącą.

Próba kolejnych ścieżek: try_files

```
location / {
    try_files $uri $uri/ /index.html;
}
```

> **Pojęcie: try_files.** Nginx sprawdza kolejne argumenty jako ścieżki na dysku i zwraca pierwszą istniejącą — powyższy wzorzec to typowa konfiguracja aplikacji jednostronicowej (SPA): najpierw próbujemy oddać plik dokładnie pasujący do URI (`$uri`), potem katalog o tej nazwie (`$uri/`), a jeśli nic nie istnieje, oddajemy zawsze `/index.html` i zostawiamy routing aplikacji JavaScript po stronie przeglądarki.

Częste błędy

Pomylenie root z alias. `root` wewnątrz `location /static/` zachowuje segment `static` w ścieżce na dysku, `alias` go usuwa — użycie `root` tam, gdzie potrzebny jest `alias`, kończy się błędem 404, bo Nginx szuka pliku w katalogu, który nie zawiera podkatalogu o nazwie z URI.

Nowy location z regexem psuje istniejące prefiksy. Dodanie `location ~* \.php$` bez `^~` przy istniejącym `location /app/` z zamiarem obsłużenia tylko plików PHP potrafi przejąć też żądania, które nie miały trafić w regex, jeśli dopasowanie prefiksowe do `/app/` nie zostało oznaczone `^~` —

regex zawsze wygrywa ze zwykłym prefiksem, niezależnie od kolejności w pliku.

Brak ukośnika końcowego przy alias. `alias /var/www/assets` bez końcowego `/` przy `location /static/` (z ukośnikiem) daje błąd albo nieoczekiwaną ścieżkę — reguła jest prosta: końcowe ukośniki w `location` i `alias` muszą się zgadzać.

□□ Ćwiczenia

> Wszystko poniżej sprawdzamy komendą `nginx -t` po każdej zmianie, zanim przeładujemy usługę.

Ćwiczenie 1 — przewidzenie wygranej reguły

```
location /images/ { ... }
location ^~ /images/thumbnails/ { ... }
location ~* \.png$ { ... }
```

Dla żądania `/images/thumbnails/logo.png` — który blok faktycznie obsłuży żądanie?

Odpowiedź

`location ^~ /images/thumbnails/` — to najdłuższy pasujący prefiks spośród prefiksowych, a flaga `^~` każe Nginksowi w ogóle nie sprawdzać wyrażeń regularnych, mimo że `\.png$` też by pasowało.

Pytania kontrolne

1. Dlaczego kolejność zapisania bloków `location` w pliku ma znaczenie tylko dla wyrażeń regularnych, a nie dla prefiksów?

Odpowiedź

Bo wśród dopasowań prefiksowych Nginx zawsze wybiera najdłuższy pasujący, niezależnie od kolejności zapisu — dopiero wśród wyrażeń regularnych, gdzie nie da się jednoznacznie

porównać „długości" dopasowania, obowiązuje kolejność zapisu w pliku i wygrywa pierwsze pasujące.

Powiązane: Architektura Nginx: procesy, konteksty, pliki konfiguracyjne · Reverse proxy i równoważenie obciążenia · Logi i diagnostyka

Źródła: dokumentacja nginx.org, rozdział „ngx_http_core_module" (dyrektywy `location`, `root`, `alias`, `try_files`)

Reverse proxy i równoważenie obciążenia

Poziom: średni **Dotyczy: Nginx 1.18+** **Czas: ~25 min**

Reverse proxy jest naszym najczęstszym zastosowaniem Nginksa — ta strona pokazuje, jak przekazujemy ruch do aplikacji za nim, co ustawiamy w nagłówkach i jak rozkładamy ruch na kilka instancji backendu.

Skróty użyte na tej stronie (8)

Skrót	Co znaczy
HTTP	Hypertext Transfer Protocol: protokół stron WWW
HTTPS	HTTP zabezpieczony szyfrowaniem TLS
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
TCP	Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS
URI	ogólny adres zasobu w internecie, część adresu URL bez samej domeny
URL	Uniform Resource Locator: adres strony lub zasobu
WebSocket	stałe, dwukierunkowe połączenie przeglądarki z serwerem, przez które Home Assistant na żywo odświeża panele; serwer pośredniczący musi przekazywać nagłówki Upgrade i Connection

Fiszka

TYLKO ODCZYT

Przekazanie ruchu do backendu: proxy_pass

```
location /app/ {
    proxy_pass http://10.20.10.41:8080;
}
```

Wskazujemy adres, na który Nginx ma przekazać żądanie pasujące do danego `location`. Warto zwrócić uwagę na obecność albo brak ukośnika na końcu adresu — opisujemy to niżej, bo to jest jedna z najczęstszych pułapek konfiguracyjnych.

TYLKO ODCZYT

Pułapka ukośnika na końcu proxy_pass

```
location /app/ {
    proxy_pass http://10.20.10.41:8080/;
}
```

Uważaj: ukośnik na końcu adresu w `proxy_pass` zmienia sposób składania ścieżki przekazywanej dalej. Bez ukośnika (`proxy_pass http://10.20.10.41:8080;`) Nginx dokleja całą ścieżkę URI żądania bez zmian — żądanie `/app/status` trafia do backendu jako `/app/status`. Z ukośnikiem na końcu (`proxy_pass http://10.20.10.41:8080/;`) Nginx zastępuje dopasowany prefiks `location` (tutaj `/app/`) tym, co jest po ukośniku w `proxy_pass` — to samo żądanie `/app/status` trafia do backendu jako `/status`, bez segmentu `app`. Jeżeli aplikacja backendu nie spodziewa się takiej zamiany, dostaje ścieżki, których nie rozpoznaje, i odpowiada błędem 404, mimo że sam Nginx nie zgłasza żadnego problemu.

Zapis `proxy_pass`	Żądanie klienta	Ścieżka wysłana do backendu
`proxy_pass http://backend;` (bez `/`)	`/app/status`	`/app/status`
`proxy_pass http://backend/;` (z `/`)	`/app/status`	`/status`

URI = ogólny adres zasobu w internecie, część adresu URL bez samej domeny

TYLKO ODCZYT

Przekazanie prawdziwego hosta: proxy_set_header Host

```
proxy_set_header Host $host;
```

Bez tej linii backend widzi w nagłówku `Host` adres, pod którym Nginx łączy się z nim samym (np. `10.20.10.41:8080`), a nie domenę, pod jaką klient faktycznie wszedł na stronę —

aplikacje generujące linki bezwzględne albo sprawdzające `Host` do celów bezpieczeństwa zaczynają wtedy generować złe adresy albo odrzucać żądania.

TYLKO ODCZYT

Przekazanie adresu klienta: proxy_set_header X-Real-IP

```
proxy_set_header X-Real-IP $remote_addr;
```

Bez tego backend widzi jako adres źródłowy zawsze adres samego Nginksa, bo to on fizycznie nawiązuje połączenie TCP — każdy log aplikacji, każdy licznik żądań na adres IP i każda reguła blokująca po IP przestaje działać poprawnie, jeśli aplikacja nie dostaje prawdziwego adresu klienta tym nagłówkiem.

TCP = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia ·**IP** = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)

TYLKO ODCZYT

Przekazanie łańcucha adresów pośredniczących: proxy_set_header X-Forwarded-For

```
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

> **Pojęcie: X-Forwarded-For.** Nagłówek zawierający listę adresów IP wszystkich pośredników, przez które przeszło żądanie, oddzielonych przecinkami — zmienna `$proxy_add_x_forwarded_for` dokleja adres bieżącego klienta do istniejącej wartości tego nagłówka (jeśli klient już go wysłał), zamiast go nadpisywać, co ma znaczenie przy kilku warstwach proxy ustawionych jedna za drugą.

IP = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)

TYLKO ODCZYT

Przekazanie protokołu oryginalnego żądania: proxy_set_header X-Forwarded-Proto

```
proxy_set_header X-Forwarded-Proto $scheme;
```

Informujemy backend, czy oryginalne żądanie klienta przyszło po HTTP czy po HTTPS — istotne, gdy Nginx kończy TLS i dalej łączy się z aplikacją zwykłym HTTP: bez tego nagłówka aplikacja myśli, że cały ruch jest nieszyfrowany, i potrafi generować linki `http://` zamiast `https://`, albo odrzucać ciasteczka oznaczone jako `Secure`.

Uważaj: brak kompletu czterech powyższych nagłówków (`Host`, `X-Real-IP`, `X-Forwarded-For`, `X-Forwarded-Proto`) jest najczęstszą przyczyną, dla której aplikacja za reverse proxy generuje złe adresy URL, myli protokół albo nieprawidłowo loguje adresy klientów — traktujemy ten komplet jako domyślny dla każdego nowego `location` z `proxy_pass`.

HTTP = Hypertext Transfer Protocol: protokół stron WWW · **HTTPS** = HTTP zabezpieczony szyfrowaniem TLS · **TLS** = Transport Layer Security: szyfrowanie połączeń, np. w HTTPS · **URL** = Uniform Resource Locator: adres strony lub zasobu

TYLKO ODCZYT

Limit rozmiaru przesyłanego ciała żądania: client_max_body_size

```
client_max_body_size 20m;
```

Bez podniesienia tego limitu każde żądanie z ciałem większym niż domyślny 1MB (np. wgranie pliku) kończy się błędem 413, opisanym na stronie Logi i diagnostyka.

TYLKO ODCZYT

Wydłużenie czasu oczekiwania na odpowiedź backendu: proxy_read_timeout

```
proxy_read_timeout 60s;
```

Wydłużamy czas, jaki Nginx czeka na dane od backendu między kolejnymi pakietami odpowiedzi — dotyczy zwłaszcza wolnych operacji, np. generowania raportu albo długiego zapytania do bazy danych po stronie aplikacji.

TYLKO ODCZYT

Bufor odpowiedzi backendu: proxy_buffering i proxy_buffers

```
proxy_buffering on;
proxy_buffers 8 16k;
```

Ustawiamy, ile pamięci Nginx rezerwuje na buforowanie odpowiedzi backendu, zanim zacznie ją odsyłać do klienta — zbyt mały bufor przy dużych nagłówkach odpowiedzi kończy się błędem `upstream sent too big header`, opisanym na stronie Logi i diagnostyka.

Z życia wzięte

Z ŻYCIA WZIĘTE

Aplikacja Oli za reverse proxy na `docker01` generuje w piątek linki `http://` zamiast `https://` w mailach z powiadomieniami, mimo że klienci łączą się wyłącznie po HTTPS. Tomek sprawdza konfigurację `location` i widzi brak `proxy_set_header X-Forwarded-Proto $scheme` w bloku obsługującym tę aplikację. Po dodaniu brakującego nagłówka i przeładowaniu nginksa kolejne maile mają już poprawny protokół w linkach.

Czym jest reverse proxy

Pojęcie: reverse proxy. Serwer stojący przed jedną albo wieloma aplikacjami, który dla klienta wygląda jak cel żądania, a w rzeczywistości przekazuje je dalej do właściwego serwera aplikacyjnego i zwraca klientowi jego odpowiedź. Klient zna tylko adres Nginksa — nie wie i nie musi wiedzieć, na jakim porcie, w którym kontenerze albo na którym z kilku serwerów faktycznie działa aplikacja. Odwrotnością jest zwykłe proxy (forward proxy), które w imieniu klienta łączy się ze światem zewnętrznym — reverse proxy działa w drugą stronę, w imieniu serwerów wobec świata.

U nas typowy scenariusz wygląda tak: Nginx nasłuchuje na porcie 443 z certyfikatem TLS opisanym na stronie TLS w Nginx, a ruch po odszyfrowaniu przekazuje dalej po zwykłym HTTP do aplikacji nasłuchującej na porcie wewnętrznym, często na innym serwerze w tej samej sieci.

Obsługa WebSocket

> **Pojęcie: WebSocket.** Protokół pozwalający utrzymać jedno długotrwałe, dwukierunkowe połączenie TCP między klientem a serwerem, zamiast zwykłego modelu żądanie-odpowiedź HTTP — używany np. przez czaty, powiadomienia w czasie rzeczywistym i konsole administracyjne aplikacji. Połączenie WebSocket zaczyna się jako zwykłe żądanie HTTP z nagłówkiem `Upgrade`, prosząc serwer o „podniesienie" połączenia do innego protokołu.

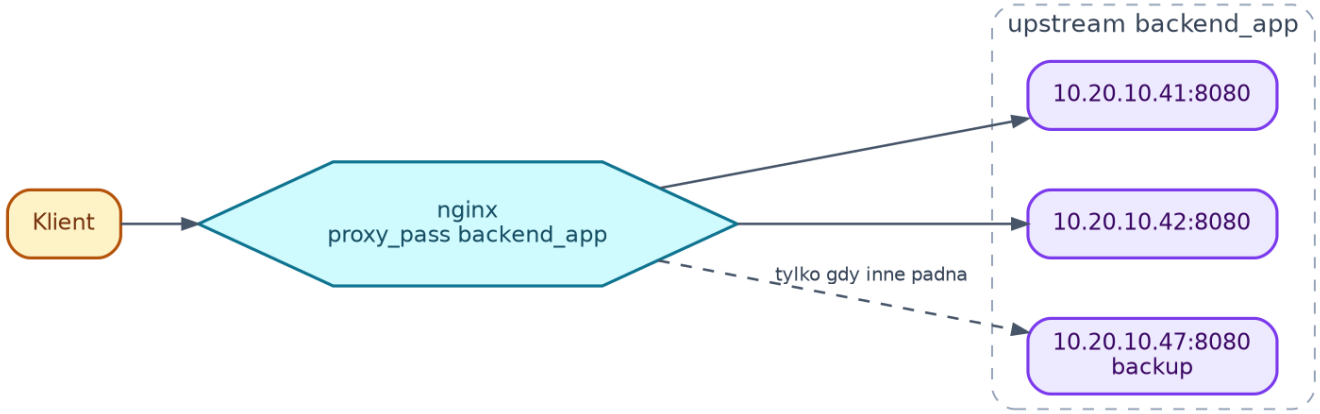
Domyślnie Nginx nie przepuszcza tego nagłówka dalej do backendu przy `proxy_pass`, więc bez dodatkowej konfiguracji próba nawiązania WebSocketu przez reverse proxy kończy się błędem po stronie aplikacji. Włączamy to jedną parą nagłówków w bloku `location` obsługującym dany endpoint:

```
location /ws/ {
    proxy_pass http://10.20.10.41:8080;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
```

`proxy_http_version 1.1` jest wymagane, bo mechanizm `Upgrade` nie istnieje w HTTP/1.0, którego Nginx używa domyślnie przy łączeniu się z backendem. Dwie kolejne linie przekazują nagłówki `Upgrade` i `Connection` z żądania klienta dalej do backendu, tak jak w oryginalnym żądaniu.

Czym jest blok upstream

> **Pojęcie: upstream.** Nazwana grupa serwerów backendu, do której `proxy_pass` odwołuje się jedną nazwą zamiast pojedynczego adresu — to jest mechanizm, dzięki któremu Nginx w ogóle potrafi rozkładać ruch na więcej niż jeden serwer aplikacyjny.



Nginx rozkłada ruch na dwa serwery backendu w grupie upstream, trzeci oznaczony backup dostaje ruch dopiero, gdy pozostałe są niedostępne (kliknięcie otwiera pełny rozmiar)

```
upstream backend_app {
    server 10.20.10.41:8080;
    server 10.20.10.42:8080;
    server 10.20.10.47:8080 backup;
}

server {
    location /app/ {
        proxy_pass http://backend_app;
    }
}
```

Serwer oznaczony `backup` dostaje ruch tylko wtedy, gdy wszystkie pozostałe serwery w grupie są niedostępne — normalnie stoi bezczynnie.

Metody równoważenia obciążenia

> **Pojęcie: równoważenie obciążenia (load balancing).** Sposób, w jaki Nginx wybiera, który z serwerów wymienionych w bloku `upstream` obsłuży kolejne żądanie. Domyślną metodą, działającą bez żadnej dodatkowej dyrektywy, jest round-robin — kolejne żądania trafiają do serwerów po kolei, w kółko.

Metoda	Dyrektywa	Jak wybiera serwer
round-robin (domyślna)	brak, działa sama	kolejno, po kolei, w kółko

Metoda	Dyrektywa	Jak wybiera serwer
least_conn	`least_conn;`	serwer z najmniejszą liczbą aktualnie otwartych połączeń
ip_hash	`ip_hash;`	hash z adresu IP klienta — ten sam klient trafia zawsze na ten sam serwer

```
upstream backend_app {
    least_conn;
    server 10.20.10.41:8080;
    server 10.20.10.42:8080;
}
```

> **Pojęcie: ip_hash.** Metoda przydatna, gdy aplikacja trzyma stan sesji lokalnie na danym serwerze (sesja w pamięci procesu, nie w zewnętrznym Redisie) — bez niej round-robin albo least_conn potrafi w kolejnych żądaniach tego samego klienta trafiać na inny serwer, który nie zna jego sesji, i wylogowywać go losowo.

Utrzymanie połączeń do backendu: keepalive

```
upstream backend_app {
    server 10.20.10.41:8080;
    server 10.20.10.42:8080;
    keepalive 32;
}
```

Ustawiamy liczbę połączeń do backendu, jakie Nginx utrzymuje otwarte i gotowe do ponownego użycia, zamiast nawiązywać nowe połączenie TCP przy każdym żądaniu — realnie obniża obciążenie backendu przy dużym ruchu.

Wykrywanie awarii backendu: max_fails i fail_timeout

```
upstream backend_app {
    server 10.20.10.41:8080 max_fails=3 fail_timeout=30s;
```

```
    server 10.20.10.42:8080 max_fails=3 fail_timeout=30s;
}
```

Nginx wyłącza dany serwer z rotacji na czas `fail_timeout`, jeśli w tym samym oknie czasowym zanotował `max_fails` nieudanych prób połączenia — po upływie tego czasu ponownie próbuje wysłać do niego ruch, sprawdzając, czy serwer wrócił do życia.

Częste błędy

Ukośnik na końcu proxy_pass zmienia ścieżkę bez ostrzeżenia. `nginx -t` nie zgłasza żadnego błędu składniowego przy błędnie dobranym ukośniku, więc jedynym objawem jest błąd 404 albo 400 zwracany przez aplikację, która dostała inną ścieżkę niż się spodziewała.

Brak proxy_http_version 1.1 przy WebSocket. Próba nawiązania WebSocketu kończy się błędem po stronie klienta (np. `Error during WebSocket handshake: Unexpected response code: 400`), mimo poprawnie ustawionych nagłówków `Upgrade` i `Connection` — HTTP/1.0 domyślnie używane przez Nginksa do backendu w ogóle nie obsługuje mechanizmu `Upgrade`.

Round-robin z aplikacją trzymającą sesję lokalnie. Objaw to losowe wylogowania i utrata koszyka/sesji użytkownika — naprawą jest `ip_hash` albo przeniesienie sesji do zewnętrznego magazynu wspólnego dla wszystkich instancji backendu.

□□ Ćwiczenia

> Ćwiczenia zakładają dostęp do dwóch prostych serwerów HTTP testowych (np. `python3 -m http.server` na dwóch portach) i lokalnej instalacji Nginksa.

Ćwiczenie 1 — obserwacja rozkładu round-robin

```
for i in $(seq 1 6); do curl -s http://localhost/ ; done
```

Uruchamiamy dwa serwery testowe na różnych portach, ustawiamy `upstream` z domyślnym round-robin i sprawdzamy, czy kolejne żądania faktycznie trafiają na przemian do obu serwerów.

Odpowiedź

Przy dwóch serwerach w grupie sekwencja odpowiedzi powinna wyglądać naprzemiennie: serwer A, serwer B, serwer A, serwer B — round-robin nie sprawdza obciążenia, tylko rotuje po kolei.

Pytania kontrolne

1. Dlaczego brak nagłówka X-Forwarded-Proto potrafi powodować pętlę przekierowań w aplikacji wymuszającej HTTPS?

Odpowiedź

Aplikacja widzi połączenie od Nginksa jako zwykłe HTTP (bo Nginx kończy TLS i dalej łączy się nieszyfrowanym połączeniem) i próbuje przekierować klienta na HTTPS — klient trafia z powrotem do Nginksa po HTTPS, Nginx znowu łączy się z aplikacją po HTTP, aplikacja znowu przekierowuje, i tak w kółko.

Powiązane: Bloki server i location · TLS w Nginx · Logi i diagnostyka

Źródła: dokumentacja nginx.org, rozdziały „ngx_http_proxy_module" i „ngx_http_upstream_module" · nginx.com/admin-guide/load-balancer/http-load-balancer

TLS w Nginx

Poziom: średni **Dotyczy: Nginx 1.18+ z OpenSSL 1.1.1+** **Czas: ~20 min**

Certyfikat i klucz to tylko dwa z kilku parametrów, które decydują o tym, jak bezpieczne i jak zgodne z klientami jest połączenie TLS — tę stronę czytamy razem ze stroną Certyfikaty SSL: wystawienie, konwersja, weryfikacja, która opisuje samo pozyskanie certyfikatu.

Skróty użyte na tej stronie (10)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
B2B	business to business: sprzedaż i systemy skierowane do innych firm, nie do klientów indywidualnych
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
HSTS	HTTP Strict Transport Security: nagłówek każący przeglądarce łączyć się tylko po HTTPS
HTTP	Hypertext Transfer Protocol: protokół stron WWW
HTTPS	HTTP zabezpieczony szyfrowaniem TLS
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
OCSP	Online Certificate Status Protocol: sprawdzanie na żywo, czy certyfikat nie został unieważniony
SNI	Server Name Indication: pole w pierwszym komunikacie TLS, w którym klient podaje nazwę serwera, z którym chce rozmawiać, co pozwala HAProxy wybrać backend bez rozszyfrowywania ruchu
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS

Fiszka

TYLKO ODCZYT

Wskazanie certyfikatu i łańcucha pośredniego: ssl_certificate

```
ssl_certificate /etc/nginx/ssl/NazwaDomeny/fullchain.pem;
```

> **Pojęcie: fullchain.** Plik `ssl_certificate` musi zawierać nie tylko certyfikat serwera, ale też certyfikat lub certyfikaty pośrednie urzędu certyfikacji (chain) sklejone jeden pod drugim w tym samym pliku, w kolejności od certyfikatu serwera do certyfikatu pośredniego najbliższego korzeniowi. Podanie samego certyfikatu serwera bez łańcucha pośredniego jest jednym z najczęstszych powodów, dla których przeglądarka desktopowa akceptuje stronę bez ostrzeżenia (bo ma łańcuch pośredni w swoim własnym magazynie), a klient mobilny albo skrypt sprawdzający certyfikat przez `openssl verify` zgłasza błąd niekompletnego łańcucha.

TYLKO ODCZYT

Wskazanie klucza prywatnego: ssl_certificate_key

```
ssl_certificate_key /etc/nginx/ssl/NazwaDomeny/privkey.pem;
```

Wskazujemy plik z kluczem prywatnym odpowiadającym certyfikatowi z `ssl_certificate` — jeśli klucz i certyfikat nie pasują do siebie, `nginx -t` przechodzi (składnia jest poprawna), ale usługa nie startuje albo od razu po starcie zwraca błąd w logu. Sposób sprawdzenia zgodności klucza z certyfikatem opisujemy na stronie Certyfikaty SSL: wystawienie, konwersja, weryfikacja.

SSL = Secure Sockets Layer: dawna nazwa szyfrowania, dziś formalnie TLS

TYLKO ODCZYT

Dozwolone wersje protokołu: ssl_protocols

```
ssl_protocols TLSv1.2 TLSv1.3;
```

Ograniczamy protokół do wersji uznawanych obecnie za bezpieczne — TLS 1.0 i TLS 1.1 są uznane za przestarzałe i większość skanerów bezpieczeństwa (oraz część klientów korporacyjnych) traktuje ich obecność jako błąd konfiguracji, nawet jeśli technicznie serwer nadal je obsługuje.

TLS = Transport Layer Security: szyfrowanie połączeń, np. w HTTPS

TYLKO ODCZYT

Dozwolone zestawy szyfrów: ssl_ciphers

```
ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384;
ssl_prefer_server_ciphers on;
```

> **Pojęcie: zestaw szyfrów (cipher suite).** Kombinacja algorytmów użytych do wymiany klucza, szyfrowania i integralności danych w danej sesji TLS. `ssl_prefer_server_ciphers on` każe serwerowi narzucać własną kolejność preferencji zamiast akceptować pierwszy zestaw zaproponowany przez klienta — istotne, bo starsze klienty potrafią proponować jako pierwsze słabsze zestawy.

□□ **TLS** = Transport Layer Security: szyfrowanie połączeń, np. w HTTPS

□□ TYLKO ODCZYT

Wymuszenie HTTPS przez HSTS: add_header Strict-Transport-Security

```
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
```

> **Pojęcie: HSTS (HTTP Strict Transport Security).** Nagłówek, który każe przeglądarce zapamiętać, że dana domena ma być odwiedzana wyłącznie przez HTTPS przez czas podany w sekundach (`max-age`) — po pierwszej wizycie przeglądarka sama, bez pytania serwera, zamienia każdą próbę wejścia po `http://` na `https://`, zanim jeszcze wyśle jakiekolwiek żądanie sieciowe. Flaga `always` każe dodawać ten nagłówek do każdej odpowiedzi, także tych z kodami błędów.

Uważaj: ustawienie długiego `max-age` na domenie, która później miałaby wrócić do HTTP (np. przy migracji), oznacza, że przeglądarki użytkowników, którzy już odwiedzili stronę, będą odmawiać połączenia po HTTP przez cały zadeklarowany okres — HSTS jest łatwo włączyć i trudno cofnąć.

□□ **HTTPS** = HTTP zabezpieczony szyfrowaniem TLS · **HSTS** = HTTP Strict Transport Security: nagłówek każący przeglądarce łączyć się tylko po HTTPS · **HTTP** = Hypertext Transfer Protocol: protokół stron WWW

□□ TYLKO ODCZYT

Przekierowanie ruchu z portu 80 na 443

```
server {
    listen 80;
    server_name NazwaDomeny;
    return 301 https://$host$request_uri;
}
```

Osobny blok `server` na porcie 80, którego jedynym zadaniem jest przekierowanie każdego żądania na wersję HTTPS tej samej ścieżki, zachowując oryginalny host i pełną ścieżkę wraz z parametrami zapytania (`$request_uri`).

HTTPS = HTTP zabezpieczony szyfrowaniem TLS

TYLKO ODCZYT

OCSP stapling: ssl_stapling

```
ssl_stapling on;
ssl_stapling_verify on;
ssl_trusted_certificate /etc/nginx/ssl/NazwaDomeny/chain.pem;
resolver 1.1.1.1 8.8.8.8 valid=300s;
```

> **Pojęcie: OCSP stapling.** Zamiast każąc przeglądarce klienta samodzielnie odpytywać urząd certyfikacji o status odwołania certyfikatu (co spowalnia pierwsze połączenie i ujawnia urzędowi certyfikacji, jakie strony odwiedza dany klient), serwer sam okresowo pobiera taką odpowiedź od urzędu certyfikacji i „dolepia" ją (staple) do własnego handshake TLS. Wymaga wskazania osobno pliku z certyfikatem pośrednim (`ssl_trusted_certificate`, bez klucza prywatnego) oraz działającego resolvera DNS, bo Nginx musi sam wykonać zapytanie do serwera OCSP urzędu certyfikacji.

OCSP = Online Certificate Status Protocol: sprawdzanie na żywo, czy certyfikat nie został unieważniony · **TLS** = Transport Layer Security: szyfrowanie połączeń, np. w HTTPS · **DNS** = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP

Z życia wzięte

Z ŻYCIA WZIĘTE

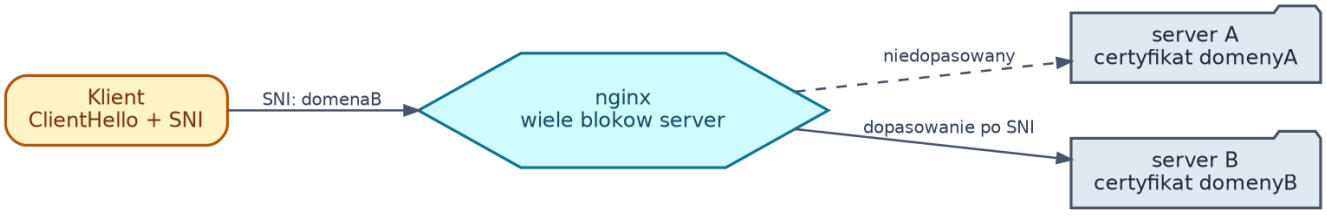
> Pan Wojtek podczas corocznego audytu sprawdza w piątek certyfikat sklepu B2B i zgłasza, że jego skaner widzi TLS 1.0 wciąż włączony na `docker01`, mimo że zespół był pewny, że go wyłączył pół roku temu. Okazuje się, że ktoś podmienił certyfikat przy odnowieniu, ale zapomniał o `nginx -s reload`, więc proces roboczy wciąż działał ze starą konfiguracją sprzed zmiany `ssl_protocols`. Po przeładowaniu `openssl s_client -tls1` faktycznie odmawia połączenia, tak jak powinno być od dawna.

Sprawdzenie, co serwer faktycznie podaje

Napisanie poprawnej konfiguracji nie gwarantuje, że serwer faktycznie ją stosuje — literówka w ścieżce do certyfikatu, stary proces, który nie przeładował się po zmianie, albo konfiguracja w złym bloku `server` potrafią sprawić, że klient dostaje zupełnie inny certyfikat niż się spodziewamy.

```
openssl s_client -connect NazwaDomeny:443 -servername NazwaDomeny </dev/null 2>/dev/null |
openssl x509 -noout -subject -issuer -dates
```

> **Pojęcie: SNI (Server Name Indication).** Parametr `-servername` w powyższej komendzie jest konieczny, gdy jeden adres IP i port obsługuje wiele domen z osobnymi certyfikatami (typowa sytuacja dla reverse proxy) — bez niego serwer nie wie, dla której domeny zwrócić certyfikat, i zwraca certyfikat domyślnego bloku `server` (albo pierwszego zdefiniowanego, jeśli żaden nie jest oznaczony `default_server`), co bywa mylące przy diagnozowaniu problemu.



Klient wysyła nazwę domeny już w ClientHello, a nginx na podstawie SNI wybiera blok server z pasującym certyfikatem zamiast domyślnego (kliknięcie otwiera pełny rozmiar)

Przykład:

```
subject=CN = app01.zielona.example
issuer=C = US, O = Let's Encrypt, CN = R11
notBefore=Aug 12 10:03:11 2026 GMT
notAfter=Nov 10 10:03:10 2026 GMT
```

Sprawdzenie, jakie wersje protokołu i zestawy szyfrów serwer faktycznie oferuje, negocjując po kolei każdą wersję:

```
openssl s_client -connect NazwaDomeny:443 -servername NazwaDomeny -tls1_2 </dev/null 2>&1 |
grep -E 'Protocol|Cipher'
```

Przykład:

```
Protocol  : TLSv1.2
Cipher    : ECDHE-RSA-AES256-GCM-SHA384
```

Odpowiedź `Protocol: TLSv1.2` przy próbie wymuszenia `-tls1` (TLS 1.0) oznacza, że serwer odmówił tej wersji i wynegocjował coś innego albo zerwał połączenie — dokładnie tego oczekujemy po ograniczeniu `ssl_protocols` do wersji 1.2 i 1.3.

Częste błędy

Certyfikat bez łańcucha pośredniego. Strona działa poprawnie w przeglądarce testującej z komputera administratora (bo system ma łańcuch pośredni w swoim magazynie zaufania z wcześniejszych wizyt na innych stronach tego samego urzędu), ale klient mobilny albo integracja API zgłasza błąd `unable to get local issuer certificate` — naprawą jest sklejenie certyfikatu pośredniego do pliku `fullchain.pem` w `ssl_certificate`.

Nieprzeładowany proces po podmianie certyfikatu. Certyfikat na dysku jest już nowy, ale `openssl s_client` nadal pokazuje stary — Nginx trzyma certyfikat wczytany do pamięci procesu roboczego i nie zauważa podmiany pliku bez `nginx -s reload`, dokładnie tak samo jak przy każdej innej zmianie konfiguracji, opisanej na stronie Architektura Nginx: procesy, konteksty, pliki konfiguracyjne.

Brak resolvera przy włączonym OCSP stapling. Usługa startuje, ale klienci dostają ostrzeżenie o nieważnym stapled response albo Nginx w logu zgłasza błąd pobrania odpowiedzi OCSP — `resolver` jest wymagany, bo bez niego proces nie ma jak wykonać zapytania DNS do serwera OCSP urzędu certyfikacji.

□□ Ćwiczenia

> Ćwiczenia zakładają dowolny certyfikat testowy, choćby samopodpisany, zainstalowany w lokalnym Nginksie.

Ćwiczenie 1 — porównanie wyniku dla różnych wersji TLS

```
for v in tls1 tls1_1 tls1_2 tls1_3; do
  echo "== $v =="
  openssl s_client -connect localhost:443 -servername NazwaDomeny -$v </dev/null 2>&1 | grep -E 'Protocol|error'
done
```

Sprawdzamy, które wersje protokołu serwer faktycznie akceptuje po ograniczeniu `ssl_protocols` do `TLSv1.2 TLSv1.3`.

Odpowiedź

Dla `tls1` i `tls1_1` połączenie powinno zakończyć się błędem negocjacji (np. `no protocols available`), a dla `tls1_2` i `tls1_3` komenda zwraca poprawny `Protocol:` z nazwą wynegocjowanej wersji.

Powiązane: Certyfikaty SSL: wystawienie, konwersja, weryfikacja · Reverse proxy i równoważenie obciążenia · Architektura Nginx: procesy, konteksty, pliki konfiguracyjne

Źródła: dokumentacja nginx.org, rozdział „ngx_http_ssl_module" · `man 1 openssl-s_client`