Praca z kodem

Pierwsze repozytorium, Gitea Actions i pierwszy kontener wdrożony z pipeline.

- Pierwsze repozytorium: od pustego projektu do pierwszego pusha
- Gitea Actions: runner i pierwszy workflow
- Pierwszy kontener: obraz budowany w Gitea i wdrożony na docker01

Pierwsze repozytorium: od pustego projektu do pierwszego pusha

Poziom: podstawy **Dotyczy: Gitea 1.27, Git dla Windows i Linux** **Czas: ~40 min**

Ta strona prowadzi osobę, która nigdy nie używała Gita, od pierwszego repozytorium w organizacji zespołu Oli do zaakceptowanego pull requesta. Po drodze tłumaczymy każde nowe pojęcie osobno, z analogią.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
CA	Certificate Authority: urząd certyfikacji, który podpisuje certyfikaty
HTTPS	HTTP zabezpieczony szyfrowaniem TLS
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera
SSL	Secure Sockets Layer: dawna nazwa szyfrowania, dziś formalnie TLS
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS

Fiszka

TYLKO ODCZYT

Utworzenie klucza SSH na Windowsie: ssh-keygen

```
ssh-keygen -t ed25519 -C "tomek.nowak@zielona.example"
```

Klucz publiczny i prywatny trafiają domyślnie do `C:\Users\tomek.nowak\.ssh\`. Zawartość pliku `id_ed25519.pub` wklejamy potem w ustawieniach konta Gitea, w sekcji SSH Keys.

□□ **SSH** = Secure Shell: szyfrowane zdalne logowanie do serwera · **ssh-keygen** = wygeneruj parę kluczy SSH albo sprawdź odcisk klucza

□□ TYLKO ODCZYT

Sklonowanie repozytorium przez SSH: git clone

```
git clone git@git.zielona.example:zielona/cennik.git
```

Pobiera pełną kopię repozytorium razem z całą historią commitów na dysk lokalny, do nowego katalogu o nazwie repozytorium.

□□ **SSH** = Secure Shell: szyfrowane zdalne logowanie do serwera · **git** = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

□□ TYLKO ODCZYT

Zapisanie zmiany do śledzenia: git add

```
git add cennik.xlsx
```

Oznacza plik jako gotowy do zapisania w kolejnym commicie. Bez tego kroku Git widzi zmianę na dysku, ale jeszcze jej nie śledzi.

□□ **git** = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

□□ ZMIENIA SYSTEM

Zatwierdzenie zmiany: git commit

```
git commit -m "Aktualizacja cen kawy na czwarty kwartal"
```

Zapisuje oznaczone zmiany jako nowy, nazwany punkt w historii projektu, razem z autorem i znacznikiem czasu.

□□ **git** = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

□□ ZMIENIA SYSTEM

Wysłanie zmian na serwer: git push

```
git push origin nazwa-galezi
```

Przesyła commity zapisane lokalnie do repozytorium na `git.zielona.example`, żeby stały się widoczne dla reszty zespołu.

git = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

TYLKO ODCZYT

Pobranie zmian z serwera: git pull

```
git pull origin main
```

Pobiera i od razu dołącza do lokalnej kopii commity, które ktoś inny wysłał na serwer od czasu ostatniego pobrania.

git = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

TYLKO ODCZYT

Utworzenie nowej gałęzi roboczej: git checkout -b

```
git checkout -b nazwa-galezi
```

Tworzy nową gałąź jako kopię bieżącego stanu i od razu na nią przełącza, żeby dalsze commity trafiały tam, a nie na `main`.

git = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

TYLKO ODCZYT

Przełączenie Gita na magazyn certyfikatów Windows: git config --global http.sslbackend

```
git config --global http.sslbackend schannel
```

Naprawia błąd certyfikatu SSL przy operacjach przez HTTPS na Windowsie, opisany niżej w sekcji o częstych błędach.

SSL = Secure Sockets Layer: dawna nazwa szyfrowania, dziś formalnie TLS · **HTTPS** = HTTP zabezpieczony szyfrowaniem TLS · **git** = system kontroli wersji (nazwa to brytyjski slang: nieznośny

typ)

Z życia wzięte

Z ŻYCIA WZIĘTE

Tomek klonuje repozytorium `cennik` na swój nowy laptop z Windows i przy pierwszym `git clone` przez HTTPS dostaje czerwony komunikat: `SSL certificate problem: unable to get local issuer certificate`. Przegląda tę samą stronę `git.zielona.example` w przeglądarce bez żadnego ostrzeżenia o certyfikacie, więc przez chwilę podejrzewa awarię serwera, zanim Ola pokazuje mu jedną komendę z sekcji Częste błędy.

Jak to działa

Rozwiń: jak to działa pod spodem

Pojęcie: repozytorium. Katalog, w którym Git śledzi każdą zmianę plików, razem z pełną historią, kto i kiedy ją wprowadził. **Analogia:** segregator z każdą wersją dokumentu spiętą osobno i podpisaną datą, zamiast jednego pliku nadpisywanego w kółko.

Pojęcie: commit. Zapisany w historii punkt, opisujący dokładnie, co się zmieniło względem poprzedniego punktu, kto to zrobił i kiedy. **Analogia:** zapis stanu gry: w każdej chwili można wrócić do dowolnego wcześniejszego zapisu, nawet jeśli od tamtej pory wydarzyło się dużo więcej.

Pojęcie: gałąź (branch). Równoległa linia commitów, zaczynająca się od wspólnego punktu z inną gałęzią, na przykład `main`. **Analogia:** kalka nałożona na oryginalny szkic: rysujemy na niej dowolne poprawki, a oryginał pod spodem zostaje nietknięty, dopóki nie zdecydujemy się przenieść poprawek na stałe.

Pojęcie: push. Wysłanie commitów zapisanych lokalnie do repozytorium na serwerze. **Analogia:** wgranie zaktualizowanego pliku na wspólny dysk sieciowy, żeby inni widzieli nową wersję.

Pojęcie: pull. Pobranie i dołączenie do lokalnej kopii commitów, które ktoś inny wcześniej wysłał na serwer. **Analogia:** odświeżenie widoku wspólnego dysku, żeby zobaczyć to, co dopisali inni od naszej ostatniej wizyty.

Pojęcie: klon (clone). Pierwsze pobranie repozytorium z serwera na dysk lokalny, razem z całą jego historią, nie tylko bieżącym stanem plików. **Analogia:** kserokopia całego segregatora ze wszystkimi wcześniejszymi wersjami dokumentów, nie tylko ostatniej kartki.

Pojęcie: organizacja. Kontener grupujący repozytoria i zespoły w Gitea, niezależny od kont osobistych poszczególnych osób. W Zielonej repozytoria zespołu programistów leżą w organizacji `zielona`, a nie na koncie osobistym Oli: odejście Oli z firmy nie zabiera ze sobą repozytoriów zespołu.

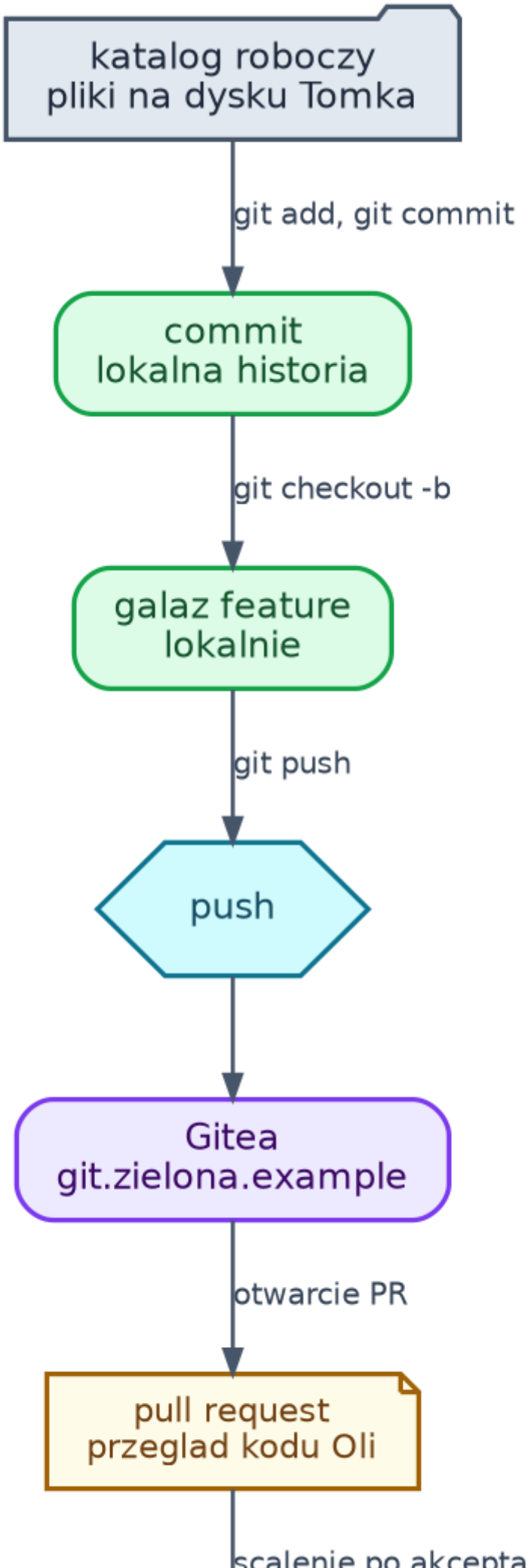
Pojęcie: pull request. Prośba o przeniesienie commitów z jednej gałęzi (na przykład gałęzi Tomka) do drugiej (na przykład `main`), razem z widokiem różnic gotowym do przejrzenia. **Analogia:** podanie z naniesionymi poprawkami, które przełożony przegląda i albo akceptuje w całości, albo odsyła z komentarzem, zanim trafi do oficjalnej dokumentacji.

Pojęcie: przegląd kodu (code review). Etap pull requesta, w którym inna osoba z zespołu, zwykle Ola jako prowadząca, czyta zaproponowane zmiany linijka po linijce, zostawia komentarze i albo prosi o poprawki, albo zatwierdza scalenie.

Pojęcie: scalenie (merge). Ostateczne przeniesienie zaakceptowanych commitów z gałęzi pull requesta do gałęzi docelowej, zwykle `main`. Od tego momentu zmiana staje się częścią głównej linii projektu.

Pojęcie: ochrona gałęzi (branch protection). Reguła w ustawieniach repozytorium blokująca bezpośredni push na wskazaną gałąź, zwykle `main`: każda zmiana musi przejść

przez pull request i przegląd kodu, zamiast trafiać na główną gałąź od razu, bez niczyjej wiedzy.


katalog roboczy
pliki na dysku Tomka
git add, git commit
commit
lokalna historia
git checkout -b
galaz feature
lokalnie
git push
push
Gitea
git.zielona.example
otwarcie PR
pull request
przeglad kodu Oli
scalenie po akceptacji

Przepływ pracy w Gitea: od katalogu roboczego przez commit i push do pull requesta i scalenia z main (kliknięcie otwiera pełny rozmiar)

Praca przez SSH albo przez HTTPS z tokenem

Do codziennej pracy Tomek wybiera jeden z dwóch sposobów łączenia się z serwerem. Klucz SSH, dodany raz w ustawieniach konta, działa bez podawania hasła przy każdej operacji `git clone`/`push`/`pull`. Token dostępu osobistego, generowany w ustawieniach konta (Settings → Applications → Generate New Token), działa jak hasło używane zamiast właściwego hasła konta, z zakresem uprawnień węższym niż całe konto: przydaje się w skryptach automatyzujących i tam, gdzie klucza SSH nie da się użyć.

Pierwszy push z Windowsa i z Linuksa

Sam protokół Git działa identycznie na obu systemach: różni się tylko powłoka, w której wpisujemy komendy (PowerShell albo Git Bash na Windowsie, bash na Linuksie) i domyślny mechanizm weryfikacji certyfikatów, opisany niżej. Kolejność kroków zostaje ta sama: `git clone`, edycja pliku, `git add`, `git commit`, `git checkout -b` na nową gałąź, `git push origin nazwa-galezi`, a na końcu pull request w interfejsie Gitea.

Błąd certyfikatu SSL przy operacjach Git na Windowsie

Objaw. Operacja `git clone`/`push`/`pull` po HTTPS na stacji Windows kończy się błędem `SSL certificate problem: unable to get local issuer certificate`, mimo że certyfikat serwera Gitea jest poprawny i zaufany przez przeglądarkę na tej samej stacji.

Przyczyna. Domyślny mechanizm weryfikacji certyfikatów w Gicie na Windowsie korzysta z własnego magazynu zaufanych certyfikatów, dostarczanego razem z Gitem (backend oparty o OpenSSL), zamiast z systemowego magazynu certyfikatów Windows. Certyfikat zaufany przez system i przeglądarkę nie musi być automatycznie zaufany przez ten wbudowany magazyn Gita.

Naprawa. Przełączamy globalnie backend kryptograficzny Gita na `schannel`, czyli natywny mechanizm Windows:

```
git config --global http.sslbackend schannel
```

Od tej zmiany Git korzysta z tego samego magazynu zaufanych certyfikatów co reszta systemu operacyjnego, więc certyfikat wewnętrznego urzędu certyfikacji rozpoznawany przez system jest rozpoznawany też przez Gita. Alternatywą, gdy z jakiegoś powodu nie chcemy zmieniać backendu globalnie dla wszystkich repozytoriów, jest wskazanie Gitowi wprost pliku z certyfikatem firmowego CA:

```
git config --global http.sslCAInfo "C:\CA\ca-zielona.crt"
```

> □□ Klonowanie przez SSH, opisane wyżej, w ogóle nie korzysta z tego mechanizmu weryfikacji certyfikatów TLS: jeśli błąd certyfikatu przeszkadza, a firmowy CA jeszcze nie trafił do zaufanych źródeł na stacjach, SSH bywa szybszym obejściem na czas naprawy tego pierwszego.

Częste błędy

Objaw	Przyczyna	Co robimy
`SSL certificate problem: unable to get local issuer certificate` na Windowsie	Git na Windowsie nie ufa firmowemu CA we własnym magazynie certyfikatów	`git config --global http.sslbackend schannel` albo `http.sslCAInfo` ze ścieżką do certyfikatu CA
`remote: Repository not found` mimo poprawnego adresu	literówka w nazwie organizacji albo repozytorium, albo brak uprawnień do niego	sprawdzamy dokładny adres w interfejsie Gitea, sekcja Clone
`! [remote rejected] main -> main (protected branch hook declined)`	próba pushu bezpośrednio na chronioną gałąź `main`	tworzymy gałąź roboczą i wysyłamy zmiany przez pull request
Konflikt przy `git pull` albo scaleniu pull requesta	ta sama linia pliku zmieniona równolegle na dwóch gałęziach	Git zaznacza konfliktujące fragmenty w pliku, poprawiamy ręcznie, `git add` , `git commit` kończący scalenie

□□ Ćwiczenia

1. Użytkownik na stacji Windows zgłasza błąd `unable to get local issuer certificate` przy `git push` do wewnętrznego serwera Gitea, mimo że przeglądarka na tej samej stacji otwiera adres serwera bez ostrzeżenia o certyfikacie. Wyjaśniamy przyczynę rozbieżności.

Odpowiedź

Przeglądarka korzysta z systemowego magazynu zaufanych certyfikatów Windows, w którym certyfikat wewnętrznego urzędu certyfikacji jest już zaufany. Git na Windowsie domyślnie korzysta z własnego, wbudowanego magazynu certyfikatów niezależnego od systemowego, więc certyfikat zaufany przez system nie musi być automatycznie zaufany przez Gita, dopóki nie przełączymy backendu na `schannel`.

1. Tomek pracuje na gałęzi `dodanie-rabatow` i próbuje wysłać zmiany komendą `git push origin main`. Wyjaśniamy, co się prawdopodobnie stanie i jak powinna wyglądać poprawna komenda.

Odpowiedź

Jeśli `main` jest gałęzią chronioną, push zostanie odrzucony niezależnie od nazwy podanej po `origin`, bo dotyczy gałęzi docelowej `main`, a nie gałęzi, na której faktycznie pracuje Tomek. Poprawna komenda to `git push origin dodanie-rabatow`, a przeniesienie zmian na `main` odbywa się przez pull request, nie przez bezpośredni push.

Powiązane: Logowanie kontem z Active Directory · Instalacja Gitea w Dockerze · Gitea Actions: runner i pierwszy workflow

Źródła: docs.gitea.com, rozdziały „Basic Usage" i „Command Line" · `git help config` (sekcje `http.sslbackend` i `http.sslCAInfo`)

Gitea Actions: runner i pierwszy workflow

Poziom: podstawy **Dotyczy: Gitea 1.27, Gitea Runner 1.0.0** **Czas: ~30 min**

Gitea Actions uruchamia automatyczne kroki (budowanie, testy, wdrożenie) przy każdym zdarzeniu w repozytorium, na przykład przy każdym pushu. Sama Gitea tylko zleca pracę, wykonuje ją osobny program: runner. Tu instalujemy runnera i uruchamiamy pierwszy, najprostszy workflow.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
CD	Continuous Delivery/Deployment: automatyczne wdrażanie zbudowanego kodu
CI	Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie; w ITIL także configuration item, czyli element konfiguracji: każdy składnik usługi zapisany w CMDB
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

KLIKAMY W INTERFEJSIE

Włączenie Actions dla repozytorium: ustawienia repozytorium

```
Ustawienia repozytorium → Actions → General → Enable Repository Actions
```

Od Gitea 1.21 mechanizm Actions jest włączony w całej instancji domyślnie, ale dla każdego repozytorium z osobna trzeba go jeszcze zaznaczyć w ustawieniach, zanim workflow z tego repozytorium w ogóle zacznie się uruchamiać.

TYLKO ODCZYT

Wygenerowanie tokenu rejestracji runnera z poziomu CLI: gitea actions generate-runner-token

```
docker exec -u git gitea gitea actions generate-runner-token
```

Token służy do jednorazowej rejestracji runnera, nie do bieżącego uwierzytelniania. Ten sam token da się pobrać bez CLI w interfejsie, patrz niżej.

Przykład:

```
9f1c2b7e4a6d8f0c3b5a7e9d1f3c5b7a9e1d3f5c
```

CLI = Command Line Interface: praca przez wpisywanie komend w terminalu · **docker** = narzędzie do kontenerów

ZMIENIA SYSTEM

Uruchomienie runnera w Dockerze: docker compose up -d

```
docker compose up -d
```

Wykonujemy w `/docker/compose/gitea-runner/`, po uzupełnieniu pliku `.env` adresem instancji i tokenem rejestracji. Runner sam rejestruje się w Gitei przy pierwszym starcie i zapisuje potwierdzenie w pliku `.runner` w katalogu danych.

Przykład:

```
[+] Running 1/1
 ✔ Container gitea-runner  Started
```

docker = narzędzie do kontenerów

KLIKAMY W INTERFEJSIE

Podgląd zarejestrowanych runnerów: Site Administration → Actions → Runners

```
Site Administration → Actions → Runners
```

Ten sam ekran, na poziomie organizacji albo repozytorium zamiast całej instancji, pokazuje runnery widoczne tylko dla tego zakresu.

TYLKO ODCZYT

Wygenerowanie pliku konfiguracyjnego runnera: gitea-runner generate-config

```
docker exec gitea-runner gitea-runner generate-config
```

Wypisuje domyślną konfigurację (limit równoległych zadań, etykiety, ścieżka do cache) do skopiowania i dostosowania w pliku `config.yaml`, zamiast trzymać wszystko w zmiennych środowiskowych.

docker = narzędzie do kontenerów

TYLKO ODCZYT

Sprawdzenie logu runnera: docker compose logs -f

```
docker compose logs -f gitea-runner
```

Pierwsze miejsce diagnozy, gdy workflow nie startuje albo runner nie łączy się z instancją.

Przykład:

```
gitea-runner  | INFO Runner registered successfully
gitea-runner  | INFO Listening for tasks...
```

docker = narzędzie do kontenerów

Z życia wzięte

Z ŻYCIA WZIĘTE

Ola, wtorek, 10:15. Włącza Actions dla repozytorium `cennik`, ale workflow po pierwszym pushu wisi w statusie „Waiting" bez końca. W logu runnera pusto, bo runner jeszcze nie istnieje: samo włączenie Actions w repozytorium nie uruchamia żadnego runnera, trzeba go dopiero postawić i zarejestrować.

Jak to działa

Rozwiń: jak to działa pod spodem

Pojęcie: CI. Skrót od ciągłej integracji: automatyczne budowanie i testowanie kodu przy każdej zmianie w repozytorium, zamiast robienia tego ręcznie od czasu do czasu. Im częściej sprawdzamy, tym szybciej łapiemy błąd, zanim narośnie na niego kolejny kod.

Pojęcie: CD. Skrót od ciągłego dostarczania (albo wdrażania): automatyczne przekazanie zbudowanej i przetestowanej wersji dalej, aż do działającego środowiska, znów bez ręcznego klikania. CI sprawdza, że kod działa, CD dba, żeby ta sprawdzona wersja trafiła tam, gdzie ma działać.

Analogia: linia montażowa. Wyobraźmy sobie fabrykę. Zdarzenie na hali (dostawa nowej partii materiału) uruchamia linię: kolejne stanowiska wykonują po kolei swoje zadania, a na końcu gotowy produkt trafia na magazyn. Gitea Actions działa tak samo, tylko materiałem jest kod, a stanowiskami komputer, który go buduje i sprawdza.

Pojęcie: workflow. Cały przepis na linię montażową dla jednego repozytorium, zapisany w jednym pliku YAML w katalogu `.gitea/workflows/`. Jedno repozytorium może mieć kilka workflow, każdy uruchamiany przez inne zdarzenie albo robiący coś innego.

Pojęcie: zdarzenie (trigger). To, co uruchamia workflow: push do gałęzi, otwarcie pull requesta, utworzenie tagu, a nawet ręczne kliknięcie w interfejsie albo harmonogram czasowy. Bez zdarzenia workflow leży bezczynnie, nawet jeśli jest poprawnie zapisany w repozytorium.

Pojęcie: zadanie (job). Jeden odcinek linii montażowej: zestaw kroków wykonywanych razem, w jednym kontenerze roboczym. Workflow może mieć kilka zadań, domyślnie uruchamianych równolegle, chyba że jawnie każemy jednemu czekać na drugie.

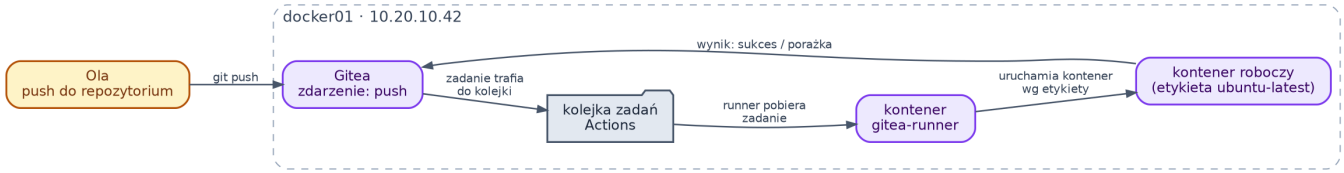
Pojęcie: krok (step). Pojedyncza czynność w zadaniu: uruchomienie komendy, pobranie kodu, zbudowanie obrazu. Kroki w jednym zadaniu wykonują się po kolei, w tym samym kontenerze roboczym, więc plik zapisany w jednym kroku widzi kolejny krok tego samego zadania.

Pojęcie: runner. Program, który stoi i czeka na nowe zadania z Gitei, a gdy jakieś się pojawi, sam pobiera je z kolejki i wykonuje w osobnym, jednorazowym kontenerze roboczym. To runner, nie sama Gitea, robi całą pracę: buduje, testuje, wdraża. Bez uruchomionego i zarejestrowanego runnera workflow czeka w kolejce bezczynnie.

Pojęcie: etykieta runnera. Nazwa (na przykład `ubuntu-latest`), którą zadanie w polu `runs-on` wskazuje, jakiego środowiska potrzebuje. Runner ma zdefiniowany zestaw etykiet, którym odpowiada obraz kontenera do uruchomienia (`ubuntu-latest:docker://node:20-bullseye`): zadanie czeka w kolejce, dopóki żaden dostępny runner nie obsługuje pasującej etykiety.

Pojęcie: sekret. Wartość (hasło, token, klucz) zapisana w ustawieniach repozytorium, dostępna w workflow przez `${{ secrets.NAZWA }}`, ale nigdy niewidoczna w samym pliku ani w logu builda. Odwrotność sekretu to zmienna repozytorium: wartość jawna, przydatna do rzeczy niewrażliwych jak nazwa środowiska.

Droga jednego zdarzenia od pushu do wyniku wygląda tak:



Push trafia do kolejki zadań, runner pobiera zadanie i uruchamia kontener roboczy, wynik wraca do Gitei (kliknięcie otwiera pełny rozmiar)

Nazwa się zmieniła: act_runner to dziś Gitea Runner

> ⚠ Do 5 maja 2026 ten sam program nazywał się `act_runner`, z obrazem `gitea/act_runner` i binarką `act_runner`. Od wersji 1.0.0 projekt przeniósł się do repozytorium `gitea.com/gitea/runner`, binarka nazywa się `gitea-runner`, a

> obraz `gitea/runner`. Instrukcje ze starszych poradników w internecie wciąż używają starych nazw: `act_runner` czytamy jako `gitea-runner`, `GITEA_INSTANCE_URL` i `GITEA_RUNNER_REGISTRATION_TOKEN` zostały bez zmian, więc te dwie zmienne środowiskowe da się przepisać bez modyfikacji.

Instalacja runnera w Dockerze

> □□ W naszym środowisku dane aplikacji trzymamy w `/docker/apps/<nazwa>`, a plik Compose w `/docker/compose/<nazwa>/compose.yaml`. To konwencja, nie wymóg Dockera: można ją zmienić, byle w całej firmie była jedna. Katalog `/docker` to osobny wolumen `vgDocker/lvDocker`, żeby pełny dysk aplikacji nie zatrzymał systemu.

`/docker/compose/gitea-runner/compose.yaml`:

```
services:
  gitea-runner:
    image: gitea/runner:1.0.0
    container_name: gitea-runner
    restart: unless-stopped
    environment:
      GITEA_INSTANCE_URL: ${GITEA_INSTANCE_URL}
      GITEA_RUNNER_REGISTRATION_TOKEN: ${GITEA_RUNNER_REGISTRATION_TOKEN}
      GITEA_RUNNER_NAME: docker01-runner
    volumes:
      - /docker/apps/gitea-runner:/data
      - /var/run/docker.sock:/var/run/docker.sock
```

`/docker/compose/gitea-runner/.env`:

```
GITEA_INSTANCE_URL=https://git.zielona.example
GITEA_RUNNER_REGISTRATION_TOKEN=WstawTokenZRejestracji
```

>

> ⚠ Wolumen `/var/run/docker.sock` daje kontenerowi runnera pełną władzę nad demonem Dockera hosta docker01, a więc w praktyce nad wszystkimi pozostałymi kontenerami na tym hoście i pośrednio nad samym hostem. Każdy workflow, który ten runner wykona, mógłby (celowo albo przez pomyłkę w cudzym kodzie z otwartego repozytorium) zatrzymać, podmienić albo odczytać dane dowolnego innego kontenera na docker01. Runnera z dostępem do gniazda Dockera podłączamy tylko do repozytoriów, którym ufamy, nigdy do publicznie dostępnej instancji z anonimowymi zgłoszeniami.

Rejestracja jest jednorazowa: przy pierwszym starcie kontener zapisuje potwierdzenie w pliku `.runner` w `/docker/apps/gitea-runner`. Kolejne `docker compose up -d` (na przykład po aktualizacji obrazu) nie rejestruje runnera ponownie, dopóki ten plik istnieje.

Instalacja runnera bez Dockera: binarka i usługa systemd

Pobieramy binarkę i rejestrujemy runnera z uprawnieniami zwykłego konta systemowego, nie roota:

```
wget -O /usr/local/bin/gitea-runner https://dl.gitea.com/gitea-runner/1.0.0/gitea-runner-1.0.0-linux-amd64
chmod +x /usr/local/bin/gitea-runner
```

```
sudo -u nazwa_usera gitea-runner register --no-interactive \
  --instance https://git.zielona.example \
  --token WstawTokenZRejestracji \
  --name NazwaRunnera
```

Jednostka systemd:

```
[Unit]
Description=Gitea Runner
After=network.target docker.service

[Service]
User=nazwa_usera
WorkingDirectory=/opt/gitea-runner
```

```
ExecStart=/usr/local/bin/gitea-runner daemon
Restart=always

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl enable --now gitea-runner
```

Ten wariant nadal potrzebuje dostępu do Dockera (grupa `docker` dla konta usługi), jeśli zadania mają uruchamiać kontenery robocze: runner sam w sobie jest tylko programem sterującym, wykonawcą kroków pozostaje kontener.

Pierwszy workflow: „Witaj, Zielona"

`.gitea/workflows/witaj.yaml`:

```
name: Witaj, Zielona
on: [push]

jobs:
  powitanie:
    runs-on: ubuntu-latest
    steps:
      - name: Pobranie kodu
        uses: actions/checkout@v4
      - name: Powitanie
        run: echo "Czesc z Gitea Actions, Zielona!"
```

Zapisujemy plik w repozytorium i wykonujemy zwykły push: samo dodanie pliku w `.gitea/workflows/` jest już zdarzeniem `push`, które go uruchamia.

> □□ Zrzut do uzupełnienia: zakładka „Actions" repozytorium `zielona/cennik` po pierwszym pushu z workflow „Witaj, Zielona", widoczny zielony status ukończonego zadania „powitanie" i rozwinięty log kroku „Powitanie" z linią „Czesc z Gitea Actions, Zielona!".

Sekrety i zmienne repozytorium

```
Ustawienia repozytorium → Actions → Secrets → Add Secret
Ustawienia repozytorium → Actions → Variables → Add Variable
```

Sekret wpisany raz nie da się już podejrzeć w interfejsie, tylko nadpisać nowym, dokładnie jak hasło w większości systemów. W workflow sięgamy po niego przez `${{ secrets.NAZWA_SEKRETU }}`, po zmienną przez `${{ vars.NAZWA_ZMIENNEJ }}`.

Gitea Actions kontra Jenkins kontra TeamCity

	Gitea Actions	Jenkins	TeamCity
Gdzie żyje definicja	plik YAML w repozytorium, `.gitea/workflows/`	osobny serwer, `Jenkinsfile` w repozytorium (Pipeline) albo konfiguracja w UI	osobny serwer, konfiguracja głównie w UI, opcjonalnie zapisana jako kod w Kotlinie
Instalacja	jeden kontener obok Gitei, runner rejestruje się sam	osobny serwer kontrolera plus agenci, więcej ruchomych części	osobny serwer, zwykle cięższy proces wdrożenia niż Jenkins
Kiedy sięgamy	mały zespół, już ma Gitea, potrzebuje prostego CI/CD bez uczenia się nowego narzędzia	duże, złożone pipeline'y, mnóstwo wtyczek, wieloletnia baza zadań do utrzymania	zespół ceniący gotowe raporty i UI bez ręcznego pisania YAML/Groovy
Składnia workflow	zgodna z GitHub Actions, YAML	Groovy (deklaratywny albo skryptowy), patrz Pipeline: Jenkinsfile, etapy, artefakty	głównie konfiguracja przez UI, mniej tekstu do utrzymania w repozytorium
Słaba strona	młodszy projekt, mniejszy ekosystem gotowych akcji niż GitHub Actions	najwięcej ruchomych części do utrzymania spośród trzech	licencja komercyjna powyżej darmowego progu liczby konfiguracji budowania

Częste błędy

Objaw	Przyczyna	Co robimy

Workflow w ogóle się nie uruchamia po pushu	Actions nie włączone dla tego repozytorium	Ustawienia repozytorium → Actions → General → Enable Repository Actions
Zadanie wisi w statusie „Waiting" bez końca	brak zarejestrowanego runnera z pasującą etykietą	sprawdzamy Site Administration → Actions → Runners, stawiamy albo rejestrujemy runnera
Runner rejestruje się przy każdym starcie kontenera od nowa	brak trwałego wolumenu na `/data`, plik `.runner` znika razem z kontenerem	montujemy `/docker/apps/gitea-runner:/data` w compose.yaml
Krok `docker build` w workflow kończy się błędem braku dostępu do Dockera	kontener runnera bez zamontowanego `/var/run/docker.sock`	dokładamy wolumen z gniazdem Dockera, ze świadomością ryzyka opisanego wyżej

Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy różnicę między zdarzeniem, zadaniem i krokiem, na przykładzie workflow „Witaj, Zielona".

Odpowiedź

Zdarzenie to `push` do repozytorium, ono uruchamia cały workflow. Zadanie to `powitanie`, jeden zestaw kroków wykonywany w jednym kontenerze roboczym. Kroki to `Pobranie kodu` i `Powitanie`, wykonywane po kolei wewnątrz tego zadania.

1. Workflow nowego repozytorium wisi w statusie „Waiting" od dziesięciu minut. Wskazujemy dwa najbardziej prawdopodobne powody i jak je sprawdzić.

Odpowiedź

Po pierwsze: Actions może nie być włączone dla tego repozytorium (Ustawienia repozytorium → Actions → General). Po drugie: żaden zarejestrowany runner nie obsługuje etykiety, której żąda pole `runs-on` (sprawdzamy Site Administration → Actions → Runners i porównujemy etykiety).

Powiązane: Gitea: Git z filiżanką herbaty · Instalacja Gitea w Dockerze · Pierwsze repozytorium: od pustego projektu do pierwszego pusha · Docker Compose · Pipeline: Jenkinsfile, etapy, artefakty

Źródła: docs.gitea.com, rozdział „Usage → Actions" · blog.gitea.com, „Release of Runner 1.0.0" · `gitea-runner --help`

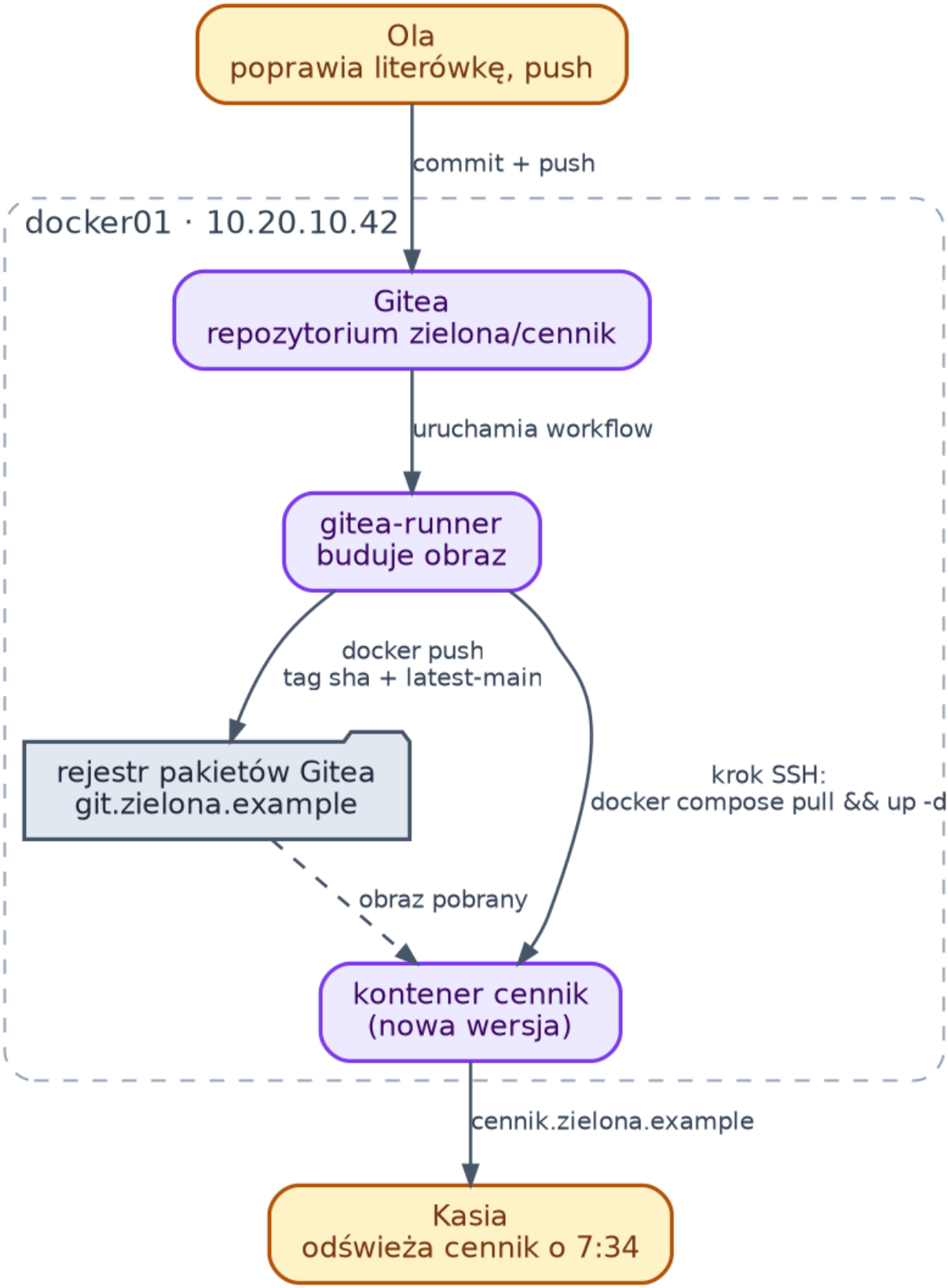
Pierwszy kontener: obraz budowany w Gitea i wdrożony na docker01

Poziom: podstawy **Dotyczy: Gitea 1.27, Gitea Runner 1.0.0, Docker na docker01**

Czas: ~45 min

Doprowadzamy drogę od pushu do działającego kontenera: repozytorium `zielona/cennik` dostaje workflow, który sam buduje obraz, wypycha go do wbudowanego rejestru Gitei i wdraża na serwerze docker01. Efekt: zmiana w kodzie trafia na produkcję bez ręcznego `docker build` i `scp`.

Cała droga od commitu do działającego kontenera, zanim rozbijemy ją na kroki:



Commit trafia do Gitei, workflow buduje obraz i wypycha go do rejestru, krok SSH wdraża go na docker01, Kasia widzi nową wersję (kliknięcie otwiera pełny rozmiar)

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
HTTP	Hypertext Transfer Protocol: protokół stron WWW
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera

Zanim zaczniemy

> **Z ŻYCIA WZIĘTE**
>
> Poniedziałek, 7:30. Kasia, w terenie, otwiera cennik na telefonie przed spotkaniem z klientem i widzi wczorajszą cenę na jeden z produktów, literówkę w opisie. Dzwoni do Oli. Ola poprawia jedną linijkę w repozytorium `zielona/cennik`, robi push z laptopa i nie robi nic więcej: reszta dzieje się sama.

Wymaganie	Uwaga
Runner z tego rozdziału, zarejestrowany i uruchomiony	patrz Gitea Actions: runner i pierwszy workflow
Organizacja `zielona` w Gitei, z repozytorium `cennik`	zespół Oli trzyma tu małą aplikację z cennikiem dla handlowców
Docker na docker01, katalogi `/docker/apps` i `/docker/compose` gotowe	patrz Instalacja Gitea w Dockerze po konwencję ścieżek
Klucz SSH do konta wdrożeniowego na docker01	konto z prawem uruchomić `docker compose` w katalogu `/docker/compose/cennik`, nic więcej

Uważaj: krok wdrożenia w tej procedurze łączy się po SSH z kontenera roboczego runnera na sam host docker01. Prywatny klucz do tego połączenia trzymamy wyłącznie jako sekret repozytorium, nigdy w treści workflow ani w logu.

Przebieg

1. Repozytorium z minimalną aplikacją

Zakładamy repozytorium `zielona/cennik` w organizacji `zielona`, tak jak opisuje strona Pierwsze repozytorium: od pustego projektu do pierwszego pusha. Aplikacja na start to statyczna strona

serwowana przez `nginx:alpine`, żeby cały przykład zmieścił się w dwóch plikach.

`cennik.html`:

```
<!doctype html>
<html lang="pl">
<head><meta charset="utf-8"><title>Cennik Zielona</title></head>
<body>
  <h1>Cennik Zielona</h1>
  <p>Kubek miodu: 24,50 zł</p>
</body>
</html>
```

`Dockerfile`:

```
FROM nginx:1.27-alpine
COPY cennik.html /usr/share/nginx/html/index.html
```

> **Pojęcie: obraz.** Gotowy, złożony z warstw zestaw plików, z którego Docker uruchamia kontener: system plików aplikacji, jej zależności i sposób jej startu, zapisane raz i uruchamiane wielokrotnie, identycznie na każdym hoście.

> **Pojęcie: rejestr.** Magazyn obrazów, z którego Docker je pobiera (`docker pull`) i do którego je wypycha (`docker push`), odpowiednik repozytorium Git, tylko dla gotowych obrazów zamiast kodu źródłowego. Gitea ma wbudowany rejestr pakietów, więc nie potrzebujemy osobnej instalacji Docker Registry czy Harbor obok.

> **Pojęcie: tag.** Etykieta doklejona do nazwy obrazu po dwukropku (`cennik:1.4.2`, `cennik:latest-main`), rozróżniająca kolejne wersje tego samego obrazu w rejestrze. Jeden i ten sam obraz może mieć naraz kilka tagów, tak jak w tej procedurze: stały tag z numeru commitu i zmienny `latest-main` wskazujący zawsze na najnowszą wersję z gałęzi `main`.

2. Token do rejestru pakietów i docker login

Token dostępu osobistego z uprawnieniem do pakietów zakładamy w profilu konta, którym będzie logował się workflow: `Ustawienia → Aplikacje → Wygeneruj nowy token`, z zaznaczonym zakresem

`write:package`.

```
docker login git.zielona.example -u nazwa_usera
```

Przykład:

```
Password:
Login Succeeded
```

W workflow logowanie dzieje się automatycznie, tokenem z sekretu repozytorium, nie ręcznie z linii poleceń: powyższe polecenie służy tylko do sprawdzenia z własnej stacji, że token w ogóle działa, zanim wpiszemy go do sekretu.

3. Sekret z tokenem rejestru

```
Ustawienia repozytorium → Actions → Secrets → Add Secret
Nazwa: REJESTR_TOKEN
```

4. Workflow budujący obraz i wypychający go do rejestru

`.gitea/workflows/build-i-publikacja.yaml`:

```
name: Budowa i publikacja obrazu
on:
  push:
    branches: [main]

jobs:
  buduj:
    runs-on: ubuntu-latest
    steps:
      - name: Pobranie kodu
        uses: actions/checkout@v4

      - name: Logowanie do rejestru Gitea
        run: echo "${{ secrets.REJESTR_TOKEN }}" | docker login git.zielona.example -u
ola.wisniewska --password-stdin
```

```
      - name: Budowa obrazu
        run: |
          docker build \
            -t git.zielona.example/zielona/cennik:${{ gitea.sha }} \
            -t git.zielona.example/zielona/cennik:latest-main \
            .

      - name: Wypchnięcie obrazu do rejestru
        run: |
          docker push git.zielona.example/zielona/cennik:${{ gitea.sha }}
          docker push git.zielona.example/zielona/cennik:latest-main
```

`${{ gitea.sha }}` to skrócony identyfikator commitu, który wywołał workflow: dzięki niemu każdy zbudowany obraz ma jeden niepowtarzalny, niezmienny tag, niezależnie od tego, że `latest-main` przesuwa się z każdym pushem na nowy obraz.

5. Compose na docker01 korzystający z obrazu z rejestru

> W naszym środowisku dane aplikacji trzymamy w `/docker/apps/<nazwa>`, a plik Compose w `/docker/compose/<nazwa>/compose.yaml`. To konwencja, nie wymóg Dockera: można ją zmienić, byle w całej firmie była jedna. Katalog `/docker` to osobny wolumen `vgDocker/lvDocker`, żeby pełny dysk aplikacji nie zatrzymał systemu.

`/docker/compose/cennik/compose.yaml`:

```
services:
  cennik:
    image: git.zielona.example/zielona/cennik:${TAG_OBRAZU}
    container_name: cennik
    restart: unless-stopped
    ports:
      - "8090:80"
```

`/docker/compose/cennik/.env`:

```
TAG_OBRAZU=latest-main
```

Zmienna `TAG_OBRAZU` w `.env` wskazuje tag, po który sięga `docker compose pull`: na co dzień `latest-main`, przy wycofaniu zmian w kroku 8 podmieniamy ją na konkretny, niezmienny tag z numeru commitu.

6. Krok wdrożenia: SSH do docker01

```
      - name: Wdrożenie na docker01
        run: |
          mkdir -p ~/.ssh
          echo "${{ secrets.KLUCZ_WDROZENIOWY }}" > ~/.ssh/id_ed25519
          chmod 600 ~/.ssh/id_ed25519
          ssh-keyscan -H 10.20.10.42 >> ~/.ssh/known_hosts
          ssh -i ~/.ssh/id_ed25519 nazwa_usera@10.20.10.42 \
            'cd /docker/compose/cennik && docker compose pull && docker compose up -d'
```

Uważaj: ten krok dokłada do repozytorium prywatny klucz SSH jako sekret `KLUCZ_WDROZENIOWY`, ważny na koncie z prawem uruchomić dokładnie jedną komendę w jednym katalogu na docker01, nic ponadto. Kompromitacja tego sekretu (na przykład przez `echo` kroku wypisany w logu przez pomyłkę) daje napastnikowi tyle, na ile pozwala to konto, nie całego hosta.

> Alternatywa bez klucza SSH: skoro runner i tak działa jako kontener na docker01, z zamontowanym gniazdem Dockera (patrz Gitea Actions: runner i pierwszy workflow), krok wdrożenia mógłby wywołać `docker compose` bezpośrednio przez to gniazdo, bez wychodzenia przez sieć. Unikamy wtedy trzymania klucza SSH jako osobnego sekretu, kosztem tego, że każdy workflow tego runnera dostaje tę samą, pełną władzę nad hostem, o której ostrzegaliśmy przy montowaniu gniazda. To nie jest wybór bezpieczniejszy, tylko inny rozkład tego samego ryzyka: SSH ogranicza je do jednego konta i jednego katalogu, gniazdo Dockera nie ogranicza niczego.

Weryfikacja

```
curl -s -o /dev/null -w "%{http_code}\n" https://cennik.zielona.example
```

Przykład:

```
200
```

```
ssh nazwa_usera@10.20.10.42 'docker compose -f /docker/compose/cennik/compose.yaml images'
```

Przykład:

```
CONTAINER   REPOSITORY                              TAG           IMAGE ID       SIZE
cennik      git.zielona.example/zielona/cennik   latest-main   4f6a2b1c9e3d   23.4MB
```

Otwieramy `https://cennik.zielona.example` w przeglądarce i sprawdzamy, czy poprawiona treść jest widoczna: to jest ostateczne potwierdzenie, kod odpowiedzi HTTP mówi tylko, że serwer w ogóle odpowiada.

Wycofanie zmian

Jeśli nowa wersja psuje coś na produkcji, wracamy do poprzedniego, znanego dobrego obrazu przez podmianę tagu w `.env`, nie przez cofanie commitów w repozytorium:

```
ssh nazwa_usera@10.20.10.42
```

```
sed -i 's/TAG_OBRAZU=.*/TAG_OBRAZU=PoprzedniTagZCommita/' /docker/compose/cennik/.env
docker compose -f /docker/compose/cennik/compose.yaml pull
docker compose -f /docker/compose/cennik/compose.yaml up -d
```

Poprzedni tag odczytujemy z listy pakietów organizacji `zielona` w Gitei (`Organizacja → Pakiety`) albo z historii workflow w zakładce Actions, gdzie każdy udany bieg pokazuje, jaki `gitea.sha` wtedy wypchnął. Tag `latest-main` nie nadaje się do wycofania, bo zawsze wskazuje na najnowszy obraz, nigdy na konkretną, wcześniejszą wersję.

Gdy coś nie zadziała

Objaw	Przyczyna	Co robimy
Krok `docker login` w workflow kończy się błędem `unauthorized`	token w sekrecie bez uprawnienia `write:package`	generujemy nowy token z zaznaczonym zakresem pakietów, aktualizujemy sekret
`docker build` działa, `docker push` odrzucony	nazwa obrazu nie zaczyna się od nazwy organizacji zgodnej z repozytorium (`git.zielona.example/zielona/...`)	poprawiamy nazwę obrazu w kroku budowy tak, żeby pasowała do organizacji `zielona`
Krok SSH kończy się `Permission denied (publickey)`	klucz w sekrecie niekompletny (brakujący nowy wiersz na końcu) albo klucz publiczny nie dodany na docker01	porównujemy klucz publiczny w `~/.ssh/authorized_keys` konta wdrożeniowego z kluczem prywatnym w sekrecie

Objaw	Przyczyna	Co robimy
Kontener `cennik` wstaje ze starą treścią mimo udanego pusha	`docker compose pull` pobrał obraz, ale `up -d` nie wymusił ponownego utworzenia kontenera, bo Compose uznał konfigurację za niezmienioną	`docker compose up -d --force-recreate` na docker01, a potem sprawdzenie tagu w `.env`

Powiązane: Gitea Actions: runner i pierwszy workflow · Instalacja Gitea w Dockerze · Pierwsze repozytorium: od pustego projektu do pierwszego pusha · Docker Compose

Źródła: docs.gitea.com, rozdział „Packages → Container" · `docker help build` · `docker help push`