

Projekt DEVOPS

Mateusz Para 14595

Informatyka stosowana niestacjonarna

Wstęp.

Cel projektu: Celem projektu jest wykorzystanie narzędzi DevOps w praktyce poprzez stworzenie i zarządzanie prostą aplikacją webową. Projekt ma obejmować kluczowe aspekty zarządzania kodem źródłowym, konteneryzacji aplikacji oraz procesu CI.

Pierwszym krokiem jaki wykonałem podczas realizacji projektu był wybór języka którym posłużę się do stworzenia prostej aplikacji webowej. Moim wyborem został flask. Czym jest flask? Flask to mikroframework webowy dla języka Python, który jest lekki, elastyczny i łatwy w użyciu. Został zaprojektowany, aby umożliwić szybkie tworzenie aplikacji internetowych i API. Jego kluczowe cechami są między innymi:

- **Lekkość:** Flask jest minimalistyczny i nie narzuca z góry konkretnej struktury aplikacji, dzięki czemu daje dużą swobodę programistom.
- **Rozszerzalność:** Posiada wiele wtyczek i rozszerzeń, które można łatwo zintegrować, np. Flask-SQLAlchemy dla obsługi baz danych.
- **Wsparcie dla Jinja2:** Używa potężnego systemu szablonów Jinja2, który pozwala generować dynamiczne strony HTML.
- **Wbudowany serwer developerski:** Idealny do testowania i debugowania aplikacji podczas rozwoju.
- **Obsługa routing:** Pozwala na definiowanie tras i obsługę żądań HTTP (GET, POST itd.) w prosty sposób.

Kolejnym elementem mojego stacku technologicznego jest nginx.

NGINX to wszechstronny, wysokowydajny serwer HTTP oraz serwer reverse proxy, który często jest również używany jako load balancer (rozpływ obciążenia) i serwer proxy dla protokołu e-mail (IMAP/POP3/SMTP). Został stworzony w celu obsługi dużej liczby jednoczesnych połączeń z wysoką wydajnością i niskim zużyciem zasobów.

Głównym powodem tego wyboru była jedna z cech charakterystycznych aplikacji a mianowicie fakt że NGINX działa jako pośrednik między klientem (np. przeglądarką) a serwerem aplikacji, przekazując żądania HTTP do serwerów w tle (np. **Flask**, Django czy Node.js). Flask posiada wbudowany automatyczny serwer developerski jednak nie służy on do użytku w środowiskach deweloperskich z takich powodów jak:

- Nie jest zoptymalizowany pod kątem wydajności.
- Nie radzi sobie dobrze z dużą liczbą równoczesnych połączeń.
- Nie oferuje wsparcia dla wielu istotnych funkcji produkcyjnych, takich jak zarządzanie ruchem czy certyfikaty SSL/TLS.

Nginx jest również przydatny z innych powodów takich jak bezpieczeństwo oraz *Load Balance* (równoważenie obciążenia między kilkoma serwerami w środowisku skalowalnym).

Czym jest CI/CD ?

CI/CD oznacza w skrócie **Continuous Integration (CI)** i **Continuous Deployment/Delivery (CD)**. Jest to praktyka oraz zestaw narzędzi stosowanych w inżynierii oprogramowania, które umożliwiają automatyzację procesu budowy, testowania i wdrażania aplikacji.

Dlaczego CI/CD jest ważny i jakie są jego zalety?

- CI/CD zapewnia szybsze dostarczanie oprogramowania.

Automatyzacja procesów pozwala na częstsze i szybsze wdrażanie zmian. Dzięki temu nowe funkcje i poprawki mogą być dostępne dla użytkowników w krótkim czasie.

- Wczesne wykrywanie błędów.

CI/CD integruje ciągłe testowanie, które pozwala wykrywać błędy już na wczesnych etapach procesu deweloperskiego. Zapobiega to eskalacji problemów na późniejszych etapach produkcji.

- Lepsza współpraca zespołu.

Ułatwia współpracę w zespołach programistycznych, ponieważ kod od wielu deweloperów jest regularnie integrowany i testowany. Redukuje problemy z tzw. "merge conflicts" (konflikty scalania kodu).

Mój wybór technologiczny podczas tworzenia oraz konfigurowania CI/CD

Do wykonania projektu wybrałem git hub actions. GitHub Actions to narzędzie oferowane przez GitHub, które umożliwia automatyzację przepływów pracy (workflowów) bezpośrednio w repozytorium pozwalająca na budowanie, testowanie i wdrażanie aplikacji automatycznie, na podstawie różnych zdarzeń w repozytorium (np. push, pull request, czy utworzenie nowej gałęzi).

Uzasadnieniem wyboru Git hubs actions jako silnika napędowego mojego projektu była wbudowana integracja z githubem oraz łatwość w konfiguracji.

Zabezpieczenie obrazu Docker

Jako sposób zabezpieczenia wybrałem trivy. Trivy to narzędzie typu open-source. Służy do skanowania luk w zabezpieczeniach oraz zarządzania konfiguracją w różnych komponentach oprogramowania. Trivy jest lekkim, szybkim i wszechstronnym rozwiązaniem, które automatyzuje identyfikację podatności na różne zagrożenia. W moim projekcie trivy jest wykorzystywane do skanowania podatności obrazu docker w trakcie CI przed dokonaniem procesem CD.

Dlaczego Trivy jako zabezpieczenie obrazu Docker?

Obrazy Docker zawierają wszystkie zależności i pliki wymagane do uruchomienia aplikacji, dlatego ich bezpieczeństwo ma kluczowe znaczenie. Trivy pozwala wykrywać potencjalne luki w takich obrazach, co zapobiega wykorzystaniu tych słabości przez atakujących.

Wymagania:

Projekt wymaga jedynie zainstalowanego dockera na systemie operacyjnym oraz połączenia z siecią umożliwiające zaciągnięcie obrazu o ramiarze ok 700MB

Instalacja:

Obraz zaciągniemy przy użyciu polecenia -

docker pull bozios11/flask-nginxpara

Uruchomienie:

Projekt możemy uruchomić przy użyciu polecenia -

docker run --name nazwaKontenera -p 8080:80 bozios11/flask-nginxpara

Założenia projektu

Aplikacja pisana w falsku zostanie uruchomiona automatycznie wraz z nginxem który przy użyciu reversed proxy przechwyci aplikacje oraz udostępni ją pod swoim domyślnym adresem na skonfigurowanym porcie, w tym przypadku jest to port 80.

Aplikacja posiada zautomatyzowany proces CI/CD. Proces CI posiada 2 *workflows* każdy z inną funkcją.

- Pierwszy proces zbuduje aplikacje oraz uruchomi specjalne testy napisane w języku aplikacji które potwierdzą prawidłowe działanie aplikacji. Ten proces uruchamia się podczas dodania pull requestu pod warunkiem modyfikacji lub edycji w folderze w którym znajdują się dane aplikacji dla oszczędności czasu jak i mocy obliczeniowej serwisu.
- Drugi proces buduje obraz dockera. Po prawidłowym zbudowaniu obrazu trivy dokona skanu aplikacji pod względem podatności oraz wyświetli w konsoli informacje o statusie aplikacji. W przypadku odnalezienia zagrożenia wygeneruje automatycznie raport. Następnie w ramach testu uruchomi obraz na lokalnym serwerze githuba oraz wyświetli przykładowe dane dla potwierdzenia prawidłowego działania aplikacji.

Procesy Continuous Implementation postanowiłem podzielić na 2 oddzielne operacje dla przejrzystości danych oraz ułatwienia debugowania aplikacji jak i również samych procesów.

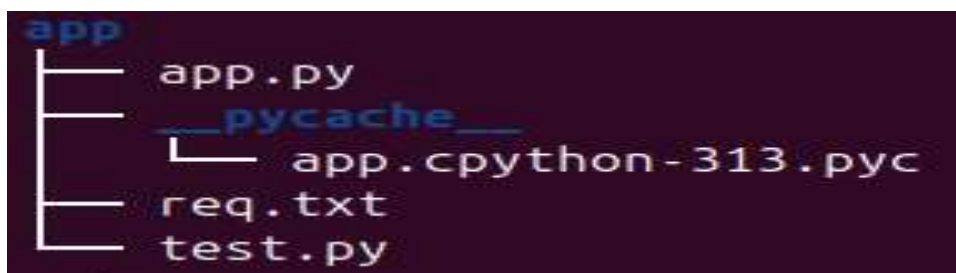
Aplikacja posiada także zautomatyzowany proces CD który uruchomi się automatycznie po zaakceptowaniu pull requestu co spowoduje zmargowanie zmian z główną gałęzią projektu. Continues Development zbuduje obraz dockera a następnie udostępni go na stronie docker.io we właściwym repozytorium. Postanowiłem nie wykonywać dodatkowych kroków odnośnie testowanie oraz skanowania obrazu ponieważ został już one wykonane w trakcie trwania CI.

Przebieg oraz budowa projektu

Przed stworzeniem oraz implementacją CI/CD potrzebna jest aplikacja którą to wyżej wymienione procesy będą obejmować.

Aplikacja ma za zadanie wyświetlać proste powitanie dla użytkownika pod głównym adresem. Aplikacja wraz z przebiegiem developmentu będzie rozwijana o dodatkowe funkcjonalności.

Struktura aplikacji



screenshot: (Zdjęcie wykonane przy użyciu linuxowej komendy tree)

Zawartość:

- app.py zawiera główny kod źródłowy aplikacji webowej:
- req.txt zawiera liste frameworków potrzebnych do uruchomienia aplikacji plik sluzi do budowania obrazu dla procesu CI oraz do ultawienia piasania Dockerfiles.
- test.py zawiera prosty test napisany przy użyciu **unittest**

Kod źródłowy oraz terminologia:

app.py

```
from flask import Flask

# Create a Flask app instance
app = Flask(__name__)

# Define a route
@app.route('/')
def hello_world():
    return 'Hello, World!'

# Run the app on port 5000
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

w pierwszej linijce kodu importujemy flaska przy użyciu polecenia „import”
następnie tworzymy obiekt aplikacji inicjujący strukturę flaska w linijce `app = Flask(__name__)`

Definiujemy ścieżkę / (główna ścieżka strony) przy użyciu dekoratora linijka `@app.route('/')` sprawi że pod głównym adresem strony w tym przypadku `localhost:5000` ukaże się nam zawartość zwracana przez funkcję `hello_world()`.

W ostatniej linijce tworzymy główną inicjalizuje blok główny aplikacji w którym uruchamiamy aplikację pod adresem lokalnym (0.0.0.0) pod portem 5000.

test.py

```
import unittest

from app import app

class TestApp(unittest.TestCase):

    def setUp(self):
        self.client = app.test_client()

    def test_app(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'Hello, World!', response.data)

if __name__ == '__main__':
    unittest.main()
```

W pierwszej linii importujemy moduł unittest, który służy do pisania testów w Pythonie. Następnie importujemy obiekt aplikacji app z pliku app.py, aby można było testować jej działanie.

Definiujemy klasę TestApp, która dziedziczy po unittest.TestCase. W tej klasie definiujemy testy naszej aplikacji.

W metodzie „setUp(self)” ustawiamy środowisko testowe dla aplikacji Flask. Tworzymy klienta testowego za pomocą „app.test_client()” i przypisujemy go do zmiennej „self.client”, aby móc z niego korzystać w kolejnych testach. Klient testowy symuluje żądania HTTP do aplikacji bez konieczności jej uruchamiania na serwerze.

Test punktu końcowego /:

W metodzie test_app testujemy działanie głównej strony aplikacji:

- `response = self.client.get('/')`
Wysyłamy symulowane żądanie HTTP GET na główną ścieżkę aplikacji (/).
Odpowiedź zapisujemy w zmiennej response.
- `self.assertEqual(response.status_code, 200)`
Sprawdzamy, czy kod statusu odpowiedzi wynosi 200 – to oznacza, że żądanie zakończyło się sukcesem.
- `self.assertIn(b'Hello, World!', response.data)`
Sprawdzamy, czy w danych zwróconych przez aplikację (w treści odpowiedzi) znajduje się tekst Hello, World!. Dodane b oznacza, że sprawdzamy tekst w postaci bajtów, ponieważ odpowiedź aplikacji jest zwracana jako bajty.

Finalne podsumowanie:

Test ten sprawdza podstawowe działanie aplikacji Flask:

- Czy główna strona (/) zwraca poprawny kod odpowiedzi HTTP (200).
- Czy w treści odpowiedzi znajduje się oczekiwany tekst Hello, World!.

Req.txt

Flask==3.1.0

Zawiera wyłącznie frameworki w zdefiniowanej wersji wymagane do uruchomienia aplikacji.

Jest to plik konfiguracyjny dla pliku tworzącego oraz budującego docker image.

Sposób uruchomienia aplikacji i testu oraz wyniki

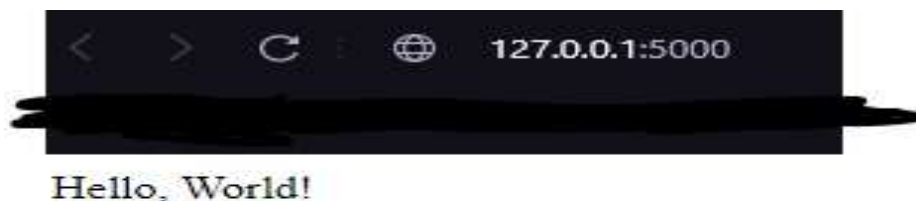
aby uruchomić aplikację wystarczy użyć polecenia „python app.py” w folderze /app

Wynik polecenia:

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania\app> python app.py
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.112:5000
Press CTRL+C to quit
```

Jak widać na załączonym zrzucie z terminalu aplikacja prawidłowo się uruchomiła a flask utworzył serwer deweloperski pod przypisanym adresem oraz portem.

Wygląd strony w przeglądarce:



test aplikacji uruchamiamy tym samym sposobem:

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania\app> python test.py
.
-----
Ran 1 test in 0.009s
OK
```

Jak widać test potwierdził prawidłowe działanie strony oraz zgodność treści strony głównej poprzez zwrócenie komunikatu „Ok”

Plik konfiguracyjny NGINX

Nginx aby przechwytywać oraz przesyłać aplikacje potrzebuje być odpowiednio skonfigurowany. W tym celu, w głównym katalogu aplikacji, utworzyłem plik konfiguracyjny dla Nginx .

Aplikacja wymaga pliku konfiguracyjnego o rozszerzeniu „.conf ”

Plik dla czytelności został nazwany nginx

Mode	LastWriteTime		Length	Name
----	-----		-----	----
dar--l	08.01.2025	21:14		app
-a----	08.01.2025	21:59	96621	Dokumentacja Mateusz Para 14595.odt
-a---l	08.01.2025	22:04	1171	nginx.conf
-a---l	02.01.2025	00:55	89191	Projekt zaliczeniowy.pdf

Figure 1: Główny folder repozytorium jak widać na załączonym obrazku zawiera on plik konfiguracyjny "nginx.conf"

Plik posłuży do skonfigurowania nginx podczas tworzenia obrazu docker.

Zawartość:

```
# Main section for configuring events

events {

# Options related to connection handling (e.g., worker_connections) can be set here.

}


# Main section for HTTP server configuration

http {

    # Server configuration

    server {

        # Listening on port 80 (HTTP)

        listen 80;

        # Server name - localhost

        server_name localhost;

        # Configuration for the root path '/'

        location / {

# Forward requests to the Flask app running on port 5000

            proxy_pass http://127.0.0.1:5000;

            # Set the "Host" header to the current hostname

            proxy_set_header Host $host;

            # Set the "X-Real-IP" header to the client's IP address

            proxy_set_header X-Real-IP $remote_addr;

            # Set the "X-Forwarded-For" header to the chain of proxy IPs

            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

            # Set the "X-Forwarded-Proto" header to the current scheme (http/https)

            proxy_set_header X-Forwarded-Proto $scheme;

        }

    }

}
```

Opis

events {}: W tej sekcji konfiguruje się opcje dotyczące obsługi połączeń w Nginx, takie jak liczba dozwolonych jednoczesnych połączeń (`worker_connections`). W tym przypadku sekcja jest pusta, ale można tu umieścić ustawienia, które wpływają na sposób zarządzania połączeniami. Ta sekcja jest wymagana do prawidłowego uruchomienia aplikacji.

Http {} . Sekcja HTTP to główny blok konfiguracji dla serwera HTTP, w którym definiujemy ustawienia dotyczące serwera i trasowania żądań HTTP.

Server {}. Sekcja serwera konfiguruje wirtualny serwer Nginx, który obsługuje ruch przychodzący na określonym porcie (w tym przypadku port 80).

listen 80. Nginx nasłuchuje na porcie 80, co oznacza, że będzie obsługiwał ruch HTTP (domyślny port dla protokołu HTTP).

server_name localhost. Określa nazwę hosta, która jest powiązana z tym serwerem. W tym przypadku jest to localhost, co oznacza, że serwer będzie obsługiwał żądania wysyłane do lokalnego hosta.

Location / {}. Konfiguracja ścieżki głównej (/): Ta sekcja definiuje, jak serwer Nginx ma obsługiwać żądania wysyłane do głównej ścieżki aplikacji (czyli /).

proxy_pass <http://127.0.0.1:5000>. Wszystkie przychodzące żądania na ścieżkę / są przekazywane do aplikacji Flask działającej lokalnie na porcie 5000. Jest to kluczowe, aby Nginx pełnił rolę proxy, przekazując żądania do aplikacji backendowej.

proxy_set_header Host \$host. Ustawia nagłówek Host na wartość bieżącego hosta. Dzięki temu aplikacja Flask otrzymuje poprawną wartość nagłówka Host, co jest istotne dla prawidłowej obsługi żądań.

proxy_set_header X-Real-IP \$remote_addr. Ustawia nagłówek X-Real-IP na adres IP klienta. Jest to przydatne, gdy Nginx działa jako proxy, aby aplikacja backendowa mogła poznać rzeczywisty adres IP klienta, a nie adres IP serwera proxy.

proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for. Ustawia nagłówek X-Forwarded-For, który zawiera łańcuch adresów IP, przez które żądanie przeszło (np. jeśli było używane proxy przed serwerem Nginx). Dzięki temu aplikacja backendowa wie, przez jakie proxy przeszło żądanie.

proxy_set_header X-Forwarded-Proto \$scheme. Ustawia nagłówek X-Forwarded-Proto na wartość schematu protokołu (HTTP lub HTTPS) używanego przez klienta. Jest to pomocne, jeśli aplikacja backendowa musi wiedzieć, czy żądanie przyszło przez protokół HTTPS, mimo że Nginx działa jako proxy.

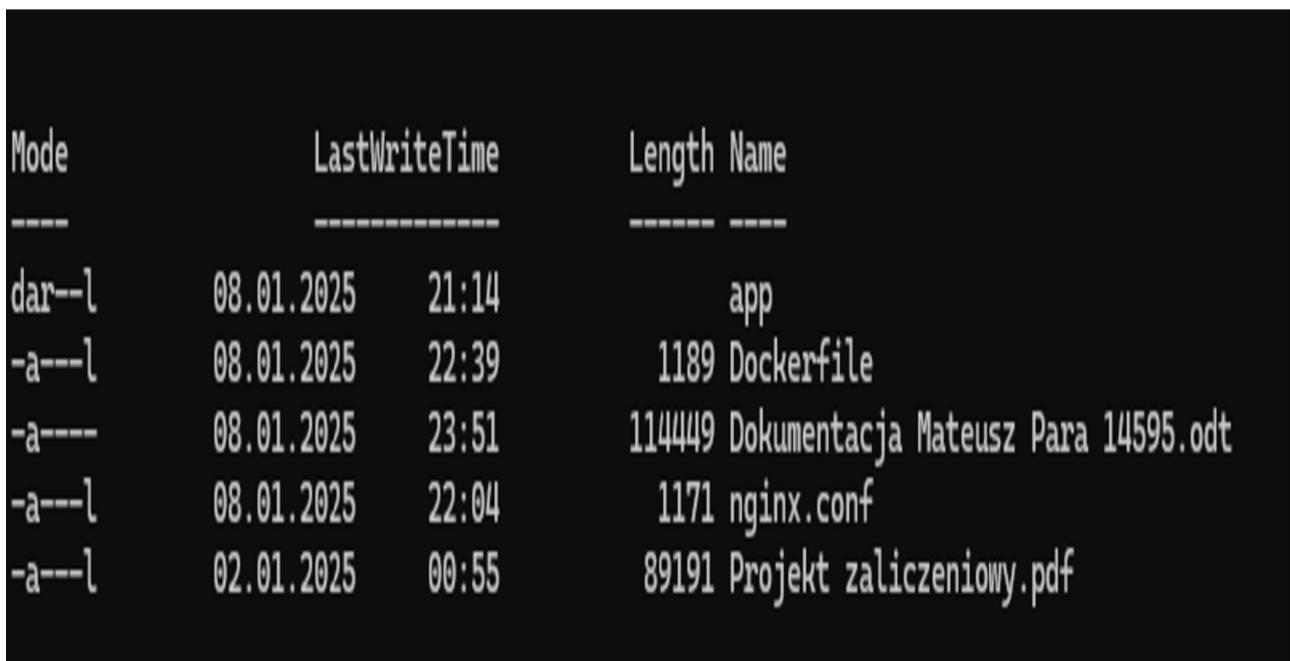
Dockerfile

Dockerfile to plik tekstowy, który zawiera instrukcje potrzebne do stworzenia obrazu kontenera. Obraz Dockera jest z kolei "szablonem", z którego można uruchamiać kontenery, czyli odizolowane środowiska do uruchamiania aplikacji. Plik Dockerfile pozwala w prosty i automatyczny sposób definiować, co znajdzie się w obrazie, np. system operacyjny, biblioteki, pliki aplikacji czy komendy do jej uruchomienia.

Domyślnie plik konfiguracyjny dockera powinien nazywać się „Dockerfile” jednak nie jest to wymagane. W przypadku innej nazwy niż domyślna podczas budowania trzeba podać nazwę pliku jako argument polecenia budującego „Docker build” pod flagą „-f”.

Przykład: „ docker build -f custom_name . ”

Plik dockerfile w przypadku mojego projektu musi znajdować się w głównym katalogu repozytorium lub w podfolderze zawierającym aplikacje. Jest to spowodowane zabezpieczeniem wbudowanym w dockera które uniemożliwia budowanie obraz z aplikacji które wybiegają poza zasięg katalogu kontekstowego. Dla przejrzystości aplikacji umieściłem go w głównym katalogu repozytorium wraz z plikiem konfiguracyjnym nginx.

A terminal window with a black background and white text showing the output of a directory listing command. The output is a table with three columns: Mode, LastWriteTime, and Length Name. The files listed are 'app', 'Dockerfile', 'Dokumentacja Mateusz Para 14595.odt', 'nginx.conf', and 'Projekt zaliczeniowy.pdf'.

Mode	LastWriteTime		Length Name
----	-----	-----	----
dar--l	08.01.2025	21:14	app
-a---l	08.01.2025	22:39	1189 Dockerfile
-a----	08.01.2025	23:51	114449 Dokumentacja Mateusz Para 14595.odt
-a---l	08.01.2025	22:04	1171 nginx.conf
-a---l	02.01.2025	00:55	89191 Projekt zaliczeniowy.pdf

Figure 1: Główny katalog projektu oraz repozytorium po dodaniu dockerfile

Zawartość

#Use a base image with Linux

FROM debian:bullseye-slim

#Update package and install Python and Nginx

RUN apt-get update && apt-get install -y \

python3 \

python3-pip \

nginx \

&& rm -rf /var/lib/apt/lists/*

#Set up a workdir for app

WORKDIR /app2

#Copy application code to workdir

COPY ./app /app2

#Install Python dependencies

RUN pip3 install --no-cache-dir -r /app2/req.txt

#Remove the default Nginx site configuration

RUN rm /etc/nginx/sites-enabled/default

#Copy the Nginx config file

COPY ./nginx.conf /etc/nginx/nginx.conf

#Expose port 80 for Nginx which will proxy flask

EXPOSE 80

#Command to start Flask and Nginx

CMD ["sh", "-c", "python3 /app2/app.py & exec nginx -g 'daemon off;']

OPIS

FROM debian:bullseye-slim - budujemy obraz na bazie lekkiej wersji systemu Debian (wersja *bullseye-slim* wybrany z powodu lekkości).

RUN apt-get update

**&& apt-get install -y \ python3 \ python3-pip \ nginx **

&& rm -rf /var/lib/apt/lists/* -

apt-get update: Aktualizuje listę dostępnych pakietów.

Instalacja: **&& apt-get install -y \ python3 \ python3-pip \ nginx **

- **python3:** Instalujemy język Python 3.
- **python3-pip:** Instalujemy narzędzie do zarządzania bibliotekami (pip).
- **nginx:** Instalujemy serwer WWW, posłuży jako proxy.

Czyszczenie: **&& rm -rf /var/lib/apt/lists/***

Usuwanie tymczasowe pliki (**/var/lib/apt/lists/***), aby zmniejszyć rozmiar obrazu.

WORKDIR /app2 - Tworzymy i ustawiamy katalog roboczy w obrazie.

COPY ./app /app2 - Kopiujemy lokalny katalog **app** do katalogu **/app2** w obrazie.

RUN pip3 install --no-cache-dir -r /app2/req.txt -

Instaluje zależności aplikacji z pliku **req.txt** za pomocą **pip3**.

Opcja **--no-cache-dir** zapobiega przechowywaniu tymczasowych plików, co zmniejsza rozmiar obrazu.

RUN rm /etc/nginx/sites-enabled/default - Usuwamy domyślną konfigurację Nginx, żeby można było użyć własnego pliku konfiguracyjnego.

COPY ./nginx.conf /etc/nginx/nginx.conf - Kopiujemy własny plik konfiguracyjny `nginx.conf` do katalogu `/etc/nginx/`.

EXPOSE 80 - Deklarujemy, że kontener będzie nasłuchiwał na porcie 80, przez który użytkownicy będą łączyć się z serwerem Nginx.

CMD ["sh", "-c", "python3 /app2/app.py & exec nginx -g 'daemon off;'] -

- **python3 /app2/app.py**: Uruchamiamy aplikację Flask w tle.
- **exec nginx -g 'daemon off;'**: Uruchamiamy serwer Nginx w trybie pierwszoplanowym (nie jako proces w tle).

Podsumowanie:

Ten plik Dockerfile tworzy obraz oparty na lekkiej wersji systemu **Debian (bullseye-slim)**, który jest zoptymalizowany pod aplikację w Pythonie z serwerem **Nginx** jako reverse proxy.

Oto, co się dzieje:

1. **Bazowy obraz**: Używany jest minimalistyczny obraz Debiana.
2. **Instalacja Python i Nginx**: Aktualizujemy pakiety i instalujemy **Python 3**, **pip** oraz **Nginx**.
3. **Katalog roboczy**: Ustawiony jest `/app2`, gdzie znajdzie się kod aplikacji.
4. **Kopiowanie aplikacji**: Kod aplikacji z katalogu `./app` jest kopiowany do kontenera.
5. **Instalacja zależności**: Z pliku `req.txt` instalowane są biblioteki wymagane do działania aplikacji w Pythonie.
6. **Konfiguracja Nginx**: Usuwany jest domyślny plik konfiguracji Nginx, a własna konfiguracja (`nginx.conf`) jest kopiowana do odpowiedniego miejsca.
7. **Ekspozowanie portu 80**: Aplikacja będzie dostępna na porcie 80.
8. **Uruchomienie aplikacji**: Po starcie kontenera jednocześnie uruchamiana jest aplikacja Flask i serwer Nginx.

Demonstracja:

Poniżej znajduje się przebieg budowania obrazu docker lokalnie oraz uruchomienie go na prywatnym komputerze.

1. Pierwszym krokiem jest zbudowanie obrazu, wykonam to za pomocą polecenia

„**docker build -t test-lokalny .**” gdzie:

-t test-lokalny nadaje nazwę naszego zbudowanego obrazu w tym przypadku test-lokalny

. oznacza obecną lokalizację w której znajduje się terminal

```
PS C:\Users\bozio> cd C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania"
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> docker build -t test-lokalny .
[+] Building 7.1s (12/12) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile                0.0s
=> => transferring dockerfile: 791B                                0.0s
```

Wynik:

```
[+] Building 7.1s (12/12) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile                0.0s
=> => transferring dockerfile: 791B                                0.0s
=> [internal] load metadata for docker.io/library/debian:bullseye-slim 1.2s
=> [internal] load .dockerignore                                  0.0s
=> => transferring context: 2B                                       0.0s
=> [internal] load build context                                  0.0s
=> [7/7] COPY ./nginx.conf /etc/nginx/nginx.conf                 0.0s
=> exporting to image                                             0.5s
=> => exporting layers                                              0.3s
=> => exporting manifest sha256:c22cb5e726d9d4f81c6f04b9d4ab3a5cc6bcf8cae3bd47dc2549908a831eb9b7 0.0s
=> => exporting config sha256:a03bb0420890b721331136f1b51cd1440a7169a84566ed18c34c930550231bb7 0.0s
=> => exporting attestation manifest sha256:643c6d730d66bc4a336c42ddbacf6a6eec0bf7af615265b0161b7a5c0f 0.0s
=> => exporting manifest list sha256:73b5af9f8825c824ca4505fb4c8de03524d37e58869e6a109d9cf7374a554afa 0.0s
=> => naming to docker.io/library/test-lokalny:latest              0.0s
=> => unpacking to docker.io/library/test-lokalny:latest           0.1s
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> ^C
```

Jak widać obraz docker prawidłowo się zbudował oznacz to że obraz powinien być widoczny oraz rozpoznawalny przez docker. Sprawdźmy to poleceniem

„**docker images**”.

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
test-lokalny        latest          73b5af9f8825    6 minutes ago   684MB
flask-nginx??       latest          f185f6ah376a    25 hours ago    684MB
```

Figure 2: Jak widać nasz utworzony obraz pojawił się w liście dostępnych obrazów naszego dockera

Sprawdzamy czy aplikacja prawidłowo uruchomi się oraz udostępni pythonową stronę, aby to wykonać użyjemy polecenia „***docker run -p 8080:80 test-lokalny***”.

Wyjaśnienie polecenia:

1. docker run

Uruchamia kontener na podstawie obrazu Dockera o nazwie **test-lokalny** utworzony we wcześniejszym etapie.

2. -p 8080:80

Mapuje porty między komputerem hosta (Twój lokalny komputer) a kontenerem:

- **8080** (na hoście) — port, na którym będzie dostępna aplikacja w przeglądarce (`http://localhost:8080`).
- **80** (w kontenerze) — port, na którym działa **Nginx** wewnątrz kontenera.

3. test-lokalny

Nazwa obrazu Dockera

Wynik:

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> docker run -p 8080:80 test-lokalny
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
Press CTRL+C to quit
```

Figure 3: Pogląd z perspektywy terminalu

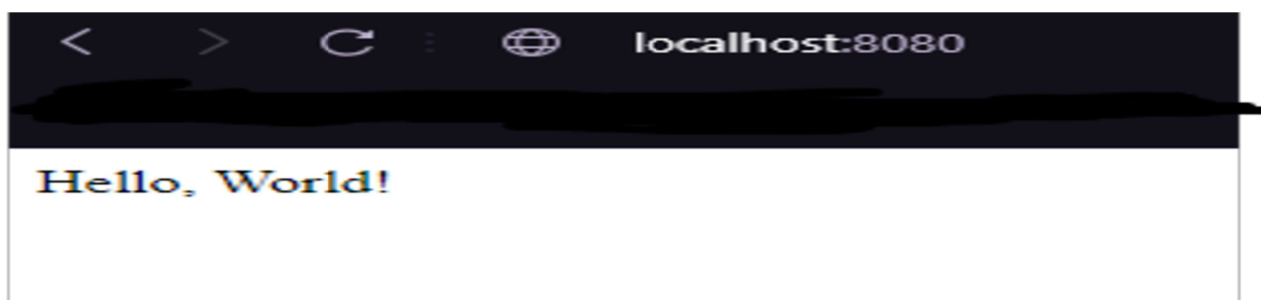


Figure 4: Pogląd z perspektywy przeglądarki

Jak widać na powyższych zrzutach ekranu obraz docker uruchomił się prawidłowo wraz z naszą aplikacją webową.

Posiadamy teraz wszystkie elementy potrzebne do utworzenia oraz przetestowania CI/CD. Przed przejściem do następnego etapu wykonamy szybki commit do gałęzi głównej naszego repozytorium github.

Continues Implementation

CI dla aplikacji python

zawartość pliku yaml:

```
name: Flask CI

on:
  pull_request:
    branches: [main]
    paths: 'app/**'

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: 3.13.1

      - name: Create virtual environment
        run: |
          python -m venv venv
          source venv/bin/activate

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          python -m pip install -r app/req.txt

      - name: Print debugging information
        run: |
          echo "Python Version: $(python --version)"
          echo "Working Directory: $(pwd)"
          echo "Contents of Working Directory: $(ls -l)"
          echo "Contents of site-packages: $(ls -l venv/lib/python*/site-packages)"

      - name: Run tests
        run: python app/test.py
```

Opis:

Ten plik to GitHub Actions Workflow o nazwie "Flask CI", który automatycznie uruchamia się przy pull requestach do gałęzi main, jeśli zmieniono pliki w folderze app. Wykonuje build na systemie Ubuntu oraz uruchamia zbudowany wcześniej test aplikacji. Najpierw pobiera kod z repozytorium za pomocą actions/checkout. Następnie ustawia wersję Pythona na 3.13.1. Tworzy wirtualne środowisko w folderze venv i aktywuje je. Instalowane są zależności z pliku req.txt znajdującego się w folderze app. Dla celów debugowania wypisuje informacje o wersji Pythona, ścieżce roboczej i zawartości folderów. Na końcu uruchamiane są testy aplikacji za pomocą skryptu test.py. Workflow pomaga automatycznie sprawdzić, czy zmiany w kodzie działają poprawnie.

name: Flask CI Ustawia nazwę workflow na "Flask CI". To po prostu etykieta, dzięki której łatwiej zrozumieć, co robi ten plik.

on: pull_request: Ten workflow uruchamia się automatycznie, kiedy ktoś tworzy pull request (propozycję zmian) do gałęzi main.

- branches: [main] — Działa tylko, gdy pull request jest skierowany do gałęzi main.
- paths: 'app/**' — Sprawdza tylko zmiany w folderze app. Jeśli zmiany są gdzie indziej, workflow się nie uruchomi.

jobs: build: Tworzy zadanie o nazwie build, które zawiera kroki do wykonania.

- runs-on: ubuntu-latest — Workflow działa na najnowszym systemie Ubuntu w chmurze GitHub.

steps:

To lista kroków, które workflow wykonuje. Każdy krok to coś konkretnego.

1. Checkout code

Copy code

```
- name: Checkout code
  uses: actions/checkout@v2
```

Pobiera najnowszy kod z repozytorium, aby można było na nim pracować w kolejnych krokach.

2. Set up Python

Copy code

```
- name: Set up Python
  uses: actions/setup-python@v2
  with:
    python-version: 3.13.1
```

Ustawia wersję Pythona na 3.13.1. Dzięki temu w kolejnych krokach wszystkie komendy będą działały na tej wersji Pythona.

3. Create virtual environment

Copy code

```
- name: Create virtual environment
run: |
  python -m venv venv
  source venv/bin/activate
```

Tworzy wirtualne środowisko w folderze venv, aby instalować zależności w odizolowanym środowisku. Następnie aktywuje je, dzięki czemu dalsze polecenia (np. instalacja bibliotek) dotyczą tylko tego środowiska.

4. Install dependencies

Copy code

```
- name: Install dependencies
run: |
  python -m pip install --upgrade pip
  python -m pip install -r app/req.txt
```

Najpierw aktualizuje menedżera pakietów pip do najnowszej wersji. Potem instaluje wszystkie potrzebne biblioteki, które są wymienione w pliku req.txt znajdującym się w folderze app.

5. Print debugging information

Copy code

```
- name: Print debugging information
run: |
  echo "Python Version: $(python --version)"
  echo "Working Directory: $(pwd)"
  echo "Contents of Working Directory: $(ls -l)"
  echo "Contents of site-packages: $(ls -l venv/lib/python*/site-packages)"
```

Wypisuje kilka przydatnych informacji:

- Wersję Pythona.
 - Aktualną ścieżkę roboczą (gdzie w systemie działa workflow).
 - Listę plików w folderze roboczym.
 - Listę zainstalowanych bibliotek w wirtualnym środowisku.
- Te informacje pomagają debugować problemy, jeśli coś pójdzie nie tak.

6. Run tests

Copy code

```
- name: Run tests
run: python app/test.py
```

Uruchamia testy aplikacji za pomocą skryptu test.py, który powinien znajdować się w folderze app. Jeśli testy przejdą pomyślnie, wszystko działa poprawnie. Jeśli nie, workflow się zatrzyma i pokaże błąd.

Prezentacja działania

Aby zaprezentować działanie CI oraz jego sprawność utworze pull request który automatycznie uruchomi workflow. Aby to zrobić utworze nową gałąź w repozytorium w której wykonam edycje kodu aplikacji. Przebieg:

Utworzenie nowej gałęzi

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git checkout -b "flask"
Switched to a new branch 'flask'
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> branch
branch : The term 'branch' is not recognized as the name of a cmdlet, function, script file, or o
the spelling of the name, or if a path was included, verify that the path is correct and try aga
At line:1 char:1
+ branch
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (branch:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git branch
* flask
main
```

Dodanie zawartości do kodu aplikacji

```
from flask import Flask

# Create a Flask app instance
app = Flask(__name__)

# Define a routes for application
@app.route('/')
def hello_world():
    return 'Hello, World!'

# added rout for testing python CI
@app.route('/python')
def python():
    return 'Python CI is very cool'

# Run the app on port 5000
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```


Stworzenie commitu z nową zawartością na utworzonej gałęzi oraz

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git add .
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git commit -m "test CI"
[flask d8bc099] test CI
 3 files changed, 7 insertions(+), 2 deletions(-)
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git push origin flask
Enumerating objects: 53, done.
Counting objects: 100% (53/53), done.
Delta compression using up to 12 threads
Compressing objects: 100% (51/51), done.
Writing objects: 100% (53/53), 505.80 KiB | 15.81 MiB/s, done.
Total 53 (delta 17), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (17/17), done.
remote:
remote: Create a pull request for 'flask' on GitHub by visiting:
remote:   https://github.com/Pogromcamyszy/DEVOPS-projekt/pull/new/flask
remote:
To https://github.com/Pogromcamyszy/DEVOPS-projekt.git
 * [new branch]      flask -> flask
```

Po wykonaniu powyższych kroków w portalu github utworzy się prośba o utworzenie pull requestu.

Dopiero wtedy github actions uruchomi proces CI.Przebieg:



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#). [Learn more about diff comparisons here](#).



base: main



compare: flask

✓ **Able to merge.** These branches can be automatically merged.



Add a title

test CI

Reviewers

No reviews

Assignees

No one—[assign yourself](#)

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Use [Closing keywords](#) in the description to automatically close issues

Helpful resources

[GitHub Community Guidelines](#)

Add a description

Write

Preview

H

B

I

≡

<>

🔗

📋

📋

📋

📋

📋

📋

📋

📋

📋

📋

📋

📋

📋

📋

Add your description here...

Markdown is supported

Paste, drop, or click to add files

Create pull request

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#)

No description provided.



test CI

d8bc099



Require approval from specific reviewers before merging

[Rulesets](#) ensure specific people approve pull requests before they're merged.

Add rule



All checks have passed

1 successful check

[Hide all checks](#)



Flask CI / build (pull_request) Successful in 16s

[Details](#)



This branch has no conflicts with the base branch

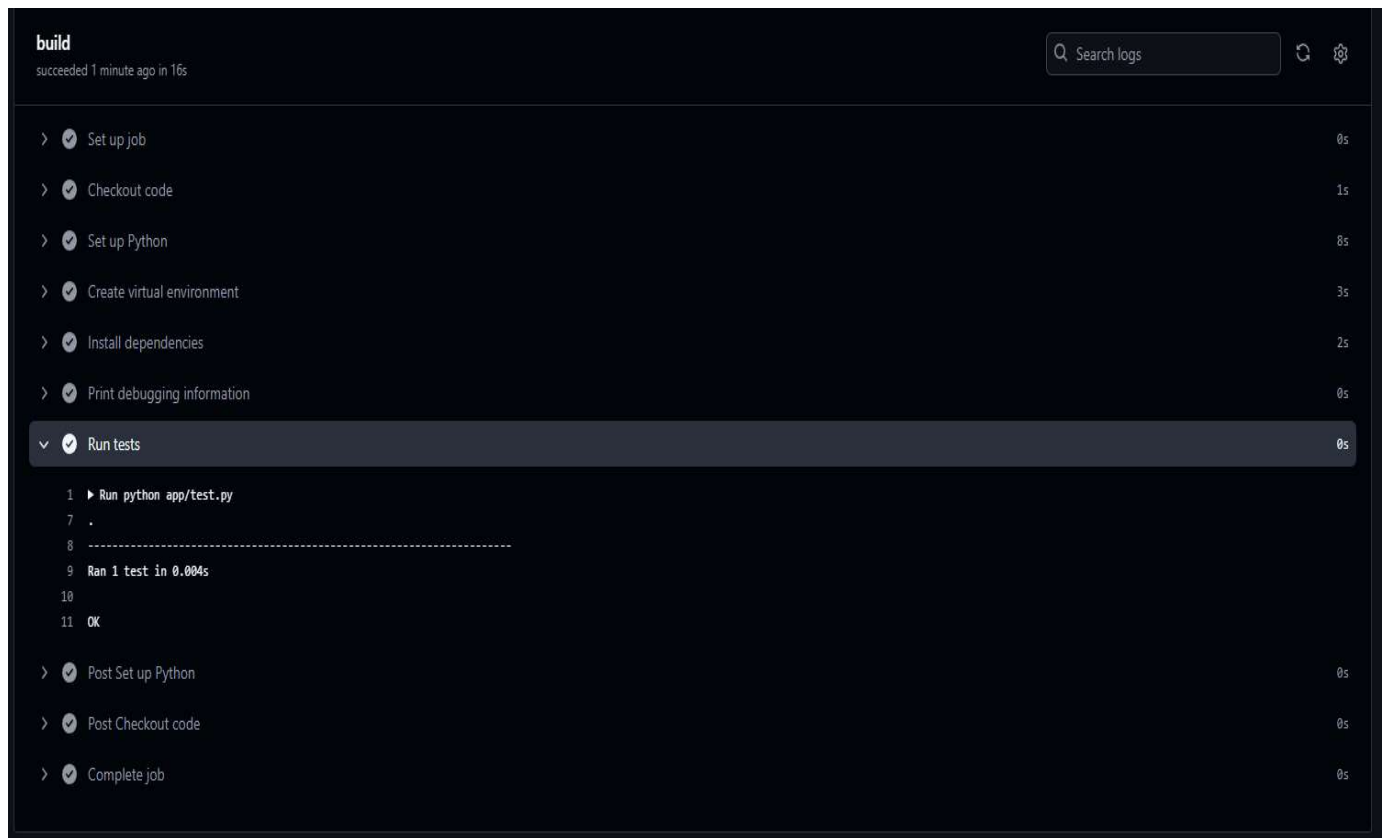
Merging can be performed automatically.

Merge pull request



You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Try the new merge experience



Jak widać proces uruchomił się prawidłowo oraz potwierdził działanie procesu CI.

Potwierdza to że gałąź może bezkolizyjnie zostać zmergowana z mainem co również uczyniłem.

CI dla obrazu Docker

zawartość pliku yaml:

```
name: Docker Image CI

on:
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
      - name: Build docker image
        run: |
          docker build . --file Dockerfile -t image13

      - name: Scan Docker Image with Trivy
        run: |
          # Install Trivy
          curl -sL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh

          # Scan the Docker image
          ./bin/trivy image --format sarif --output trivy-report.sarif --exit-code 0 --quiet --severity CRITICAL,HIGH image13

      - name: Upload SARIF Report to GitHub Security
        uses: github/codeql-action/upload-sarif@v3
        with:
          sarif_file: trivy-report.sarif

      - name: Show docker images
        run: docker images

      - name: Run Docker container in detached mode
        run: docker run -d --name my-container -p 8080:80 image13

      - name: Close container
        run: docker stop my-container
```

OPIS:

Ten workflow o nazwie Docker Image CI automatycznie sprawdza buduje i testuje obraz Dockera w trakcie pull requestów do gałęzi main. Najpierw pobiera kod repozytorium za pomocą actions/checkout. Następnie buduje obraz

Dockera na podstawie pliku Dockerfile i zapisuje go pod nazwą image13. Potem instalowany jest Trivy, który skanuje obraz Dockera pod kątem problemów bezpieczeństwa i generuje raport w formacie SARIF. Raport jest wysyłany do GitHub Security przy użyciu wtyczki `github/codeql-action/upload-sarif`. Workflow. Następnie następuje uruchomienie oraz testowanie Dockera oraz sprawdza jego bezpieczeństwo. Na samym końcu obraz Dockera jest zamykany.

1. **actions/checkout@v4**

Pobiera kod z repozytorium, aby można było z nim pracować w kolejnych krokach.

2. **docker build . --file Dockerfile -t image13**

Buduje obraz Dockera na podstawie pliku Dockerfile w bieżącym katalogu i nadaje mu nazwę image13.

3. **Trivy - Instalacja i skanowanie**

- **curl -sL https://... | sh** — Pobiera i instaluje narzędzie Trivy, które służy do skanowania obrazów Dockera pod kątem podatności.
- **./bin/trivy image ...** — Skanuje obraz image13 i generuje raport o krytycznych i wysokich problemach bezpieczeństwa w pliku `trivy-report.sarif`.

4. **github/codeql-action/upload-sarif@v3**

Wysyła raport bezpieczeństwa (plik SARIF) do GitHub Security, aby można go było przejrzeć bezpośrednio w interfejsie GitHub.

5. **docker images**

Wyświetla listę wszystkich obrazów Dockera dostępnych na maszynie, aby upewnić się, że obraz image13 został zbudowany.

5. **docker run -d --name my-container -p 8080:80 image13**

Uruchamia kontener o nazwie my-container w trybie odłączonym (-d) na porcie 8080, używając obrazu image13.

6. **docker stop my-container**


Zatrzymuje uruchomiony kontener my-container, aby zwolnić zasoby.

Prezentacja działania

Analogicznie do CI flask edytujemy kod aplikacji, tworzymy nową gałąź a następnie tworzymy pull request.

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git checkout -b "docker"
Switched to a new branch 'docker'
```

```
app.py X
C: > Users > bozio > OneDrive > Pulpit > Projekt studia do wydania > app > app.py > ...
1  from flask import Flask
2
3  # Create a Flask app instance
4  app = Flask(__name__)
5
6  # Define a routes for application
7  @app.route('/')
8  def hello_world():
9      return 'Hello, World!'
10
11 # added rout for testing python CI
12 @app.route('/python')
13 def python():
14     return 'Python CI is very cool'
15
16 #added rout for testin docker CI
17 @app.route('/docker')
18 def docker():
19     return 'Docker Ci is super cool'
20
21 # Run the app on port 5000
22 if __name__ == '__main__':
23     app.run(host='0.0.0.0', port=5000)
```

 docker had recent pushes 1 minute ago

[Compare & pull request](#)

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#). [Learn more about diff comparison](#)



base: main



compare: docker

✓ Able to merge. These branches can be automatically merged.



Add a title

testing docker CI

Add a description

Write

Preview

H

B

I

≡

<>

🔗

⋮

⋮

⋮

📎

@

📎

↩

🗑

Add your description here...

Markdown is supported

Paste, drop, or click to add files

Create pull request



Require approval from specific reviewers before merging

[Rulesets](#) ensure specific people approve pull requests before they're merged.

Add rule



All checks have passed

3 successful checks

[Hide all checks](#)



Docker Image CI / build (pull_request) Successful in 1m

[Details](#)



Flask CI / build (pull_request) Successful in 15s

[Details](#)



Code scanning results / Trivy Successful in 5s — No new alerts in code changed by this pull request

[Details](#)



This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request



You can also [open this in GitHub Desktop](#) or view [command line instructions](#).



Try the new merge experience

testing docker CI #2

Summary

Jobs

build

Run details

Usage

Workflow file

Annotations

1 warning

build

succeeded 1 minute ago in 1m 4s

Set up job

Run actions/checkout@v4

Build docker image

Scan Docker Image with Trivy

Upload SARIF Report to GitHub Security

Show docker images

Run Docker container in detached mode

Close container

Post Upload SARIF Report to GitHub Security

Post Run actions/checkout@v4

Complete job

build

succeeded 6 minutes ago in 1m 4s

Search logs

🔄

⚙️

Set up job7s

Run actions/checkout@v41s

Build docker image29s

Scan Docker Image with Trivy14s

1 ▶ Run # Install Trivy

8 aquasecurity/trivy info checking GitHub for latest tag

9 aquasecurity/trivy info found version: 0.58.1 for v0.58.1/Linux/64bit

10 aquasecurity/trivy info installed ./bin/trivy

Upload SARIF Report to GitHub Security8s

1 ▶ Run github/codeql-action/upload-sarif@v3

8 ▶ Uploading results

15 ▶ Waiting for processing to finish

Show docker images0s

Run Docker container in detached mode0s

Close container0s

Post Upload SARIF Report to GitHub Security0s

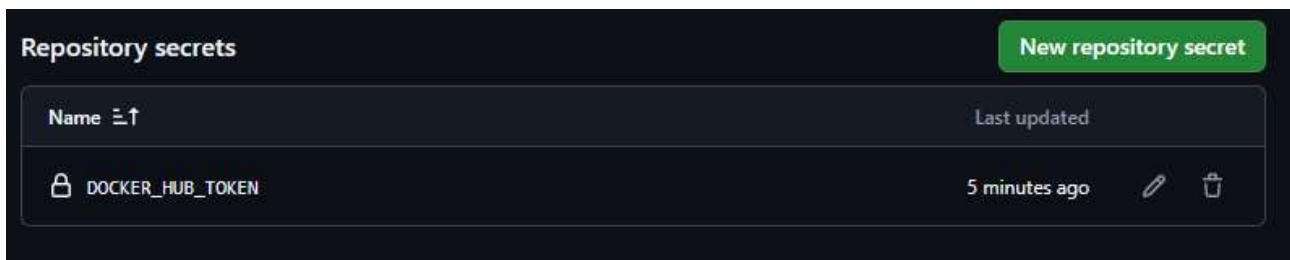
Post Run actions/checkout@v40s

Complete job0s

Jak widać CI docker uruchomiło się oraz zadziałało prawidłowo podczas pull requestu oznacza to że posiadamy już wszystkie elementy mojego CI.

Continuous Deployment

Pierwszym krokiem w implementacji CD było utworzenie tokenu autentykacyjnego do portalu docker dla mojego konta oraz dodatnie go do github secrets pod nazwa DOCKER_HUB_TOKEN.



Zawartość pliku „Docker CD publish to dockerIO.yaml”

name: Build and Publish image to Docker Hub

on:

push:

branches:

- main

jobs:

publish_images:

runs-on: ubuntu-latest

steps:

- name: checkout

uses: actions/checkout@v4

- name: build docker image image

run: |

docker build . --file Dockerfile -t bozios11/flask-nginxpara:latest

- name: push image to docker hub

run: |

docker login -u bozios11 -p \${{ secrets.DOCKER_HUB_TOKEN }}

docker push bozios11/flask-nginxpara:latest

Opis

Plik ten który automatycznie buduje obraz Dockera i publikuje go na Docker Hub. Po każdym pushu na gałąź "main", pipeline wykonuje checkout kodu, buduje obraz Dockera na podstawie Dockerfile i wypycha go na konto Docker Hub za pomocą tokena przechowywanego w sekrecie.

Nagłówek

name: Build and Publish image to Docker Hub

Nadaje nazwę workflow. W tym przypadku jest to "Build and Publish image to Docker Hub".

Uruchamianie workflow

```
on:
  push:
    branches:
      - main
```

Workflow zostanie uruchomiony, gdy wykonasz **push** (wypchnięcie zmian) na gałąź main. Możesz to zmienić, dodając inne zdarzenia lub gałęzie.

Definicja zadania (jobs)

```
jobs:
  publish_images:
    runs-on: ubuntu-latest
```

- Workflow ma jedno zadanie o nazwie publish_images.
- Zadanie działa na maszynie wirtualnej z systemem **Ubuntu (latest version)**.

Kroki zadania (steps)

1. Checkout kodu z repozytorium

```
- name: checkout
  uses: actions/checkout@v4
```

- Używa akcji actions/checkout w wersji 4.
- Pobiera kod źródłowy z repozytorium, co jest konieczne do budowy obrazu Dockera.

2. Budowa obrazu Dockera

```
- name: build docker image
  run: |
    docker build . --file Dockerfile -t bozios11/flask-nginxpara:latest
```

- Buduje obraz Dockera na podstawie pliku Dockerfile znajdującego się w katalogu głównym repozytorium.
- Obraz otrzymuje tag bozios11/flask-nginxpara:latest, co oznacza, że zostanie zapisany w repozytorium Dockera użytkownika bozios11 z wersją latest.

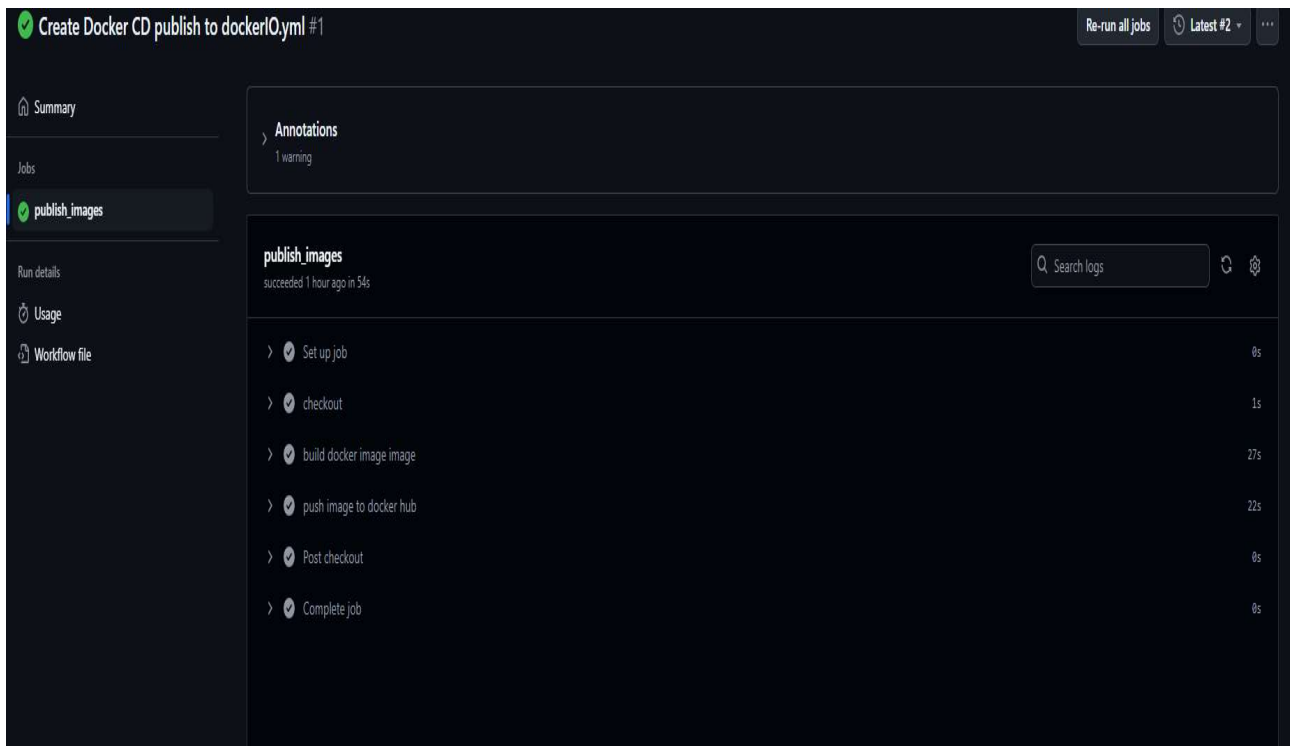
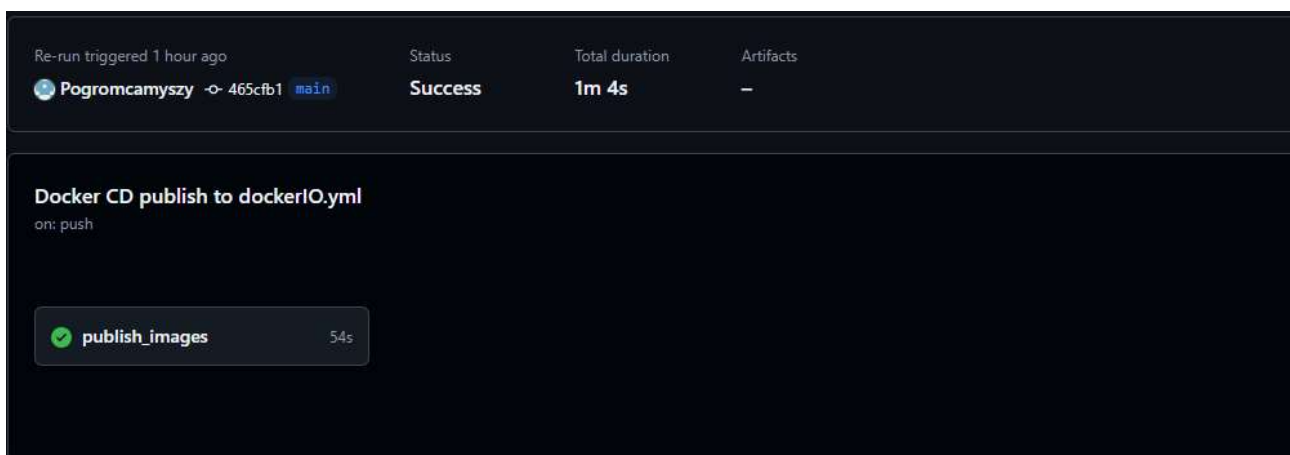
3. Publikacja obrazu na Docker Hub

```
- name: push image to docker hub
run: |
  docker login -u bozios11 -p ${ secrets.DOCKER_HUB_TOKEN }}
  docker push bozios11/flask-nginxpara:latest
```

- Loguje się do Docker Hub za pomocą nazwy użytkownika bozios11 i tokena (przechowywanego jako sekret DOCKER_HUB_TOKEN w ustawieniach repozytorium GitHub).
- Wypycha obraz bozios11/flask-nginxpara:latest na Docker Hub.

Prezentacja oraz demonstracja:

Jako iż workflow zawierający CD został z commitowant na „main” spowodowało to że samoczynnie uruchomił swój proces.



Jak widać na powyższych zrzutach ekranu proces powinien zbudować obraz naszej aplikacji na podstawie obecnych plików w repozytorium zawartych w gałęzi głównej oraz wysłać go na docker.io do mojego publicznego repozytorium pod nazwą.

bozios11/flask-nginxpara

Last pushed about 2 hours ago

Img of my flask app with nginx as server included and installed app is ready to start from go

Add a category [INCOMPLETE](#)


Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
latest		Image	2 minutes ago	2 hours ago

[See all](#)

Obraz pojawił się w repozytorium docker hub co oznacza prawidłowy transfer.



bozios11/flask-nginxpara

By [bozios11](#) · Updated about 2 hours ago


Img of my flask app with nginx as server included and installed app is ready to start from go

[IMAGE](#)

☆0 ↓36

[Manage Repository](#)

Overview **Tags**



No overview available

This repository doesn't have an overview

Docker Pull Command

```
docker pull bozios11/flask-nginxpara
```

[Copy](#)

Test

```
Terminal

latest: Pulling from bozios11/flask-nginxpara
48627b50d5db: Download complete
189d5913c199: Download complete
f5f32e357a4b: Download complete
ae5be469489f: Download complete
27c667c7d20f: Download complete
0d224e80cc11: Download complete
Digest: sha256:3f96d001ddc81f6f07f9f93feb2e9d077857a1c409eb109114e0d7687cb5fd9c
Status: Downloaded newer image for bozios11/flask-nginxpara:latest
docker.io/bozios11/flask-nginxpara:latest
PS C:\Users\bozio> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
bozios11/flask-nginxpara  latest             3f96d001ddc8       2 hours ago        684MB
```

Figure 5: Zaciągamy obraz z publicznego repozytorium na lokalne urządzenie

```
PS C:\Users\bozio> docker run --name pokaz -p 8080:80 bozios11/flask-nginxpara
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server
instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
Press CTRL+C to quit
[]
```

Figure 6: Uruchomienie obrazu docker

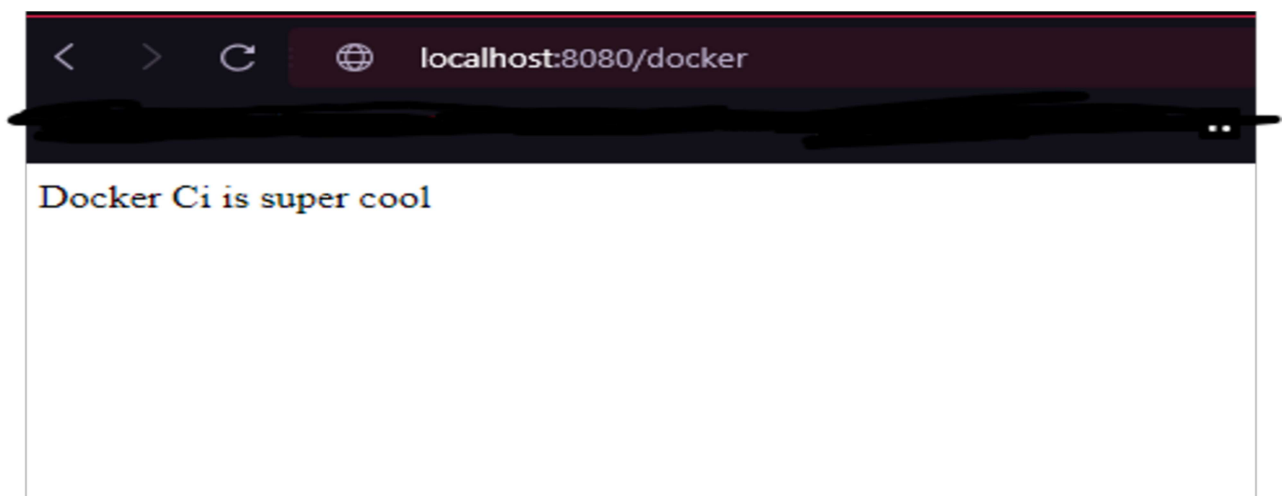


Figure 7: Podgląd z perspektywy przeglądarki internetowej

Install the latest PowerShell for new features and improvements! <https://aka.ms/PSWindows>

```
PS C:\Users\bozio> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
4edf2b0dc4ee	bozios11/flask-nginxpara	"sh -c 'python3 /app..."	3 minutes ago	Up 3 minutes	0.0.0.0:8080

```
PS C:\Users\bozio>
```

Figure 8: Status procesów docker uruchomianych w tle jak widać uruchomiony jest tylko obraz zaciągnięty z docker huba

Przedstawienie ciągłości CI/CD podczas pracy.

Aby zaprezentować działanie zamierzam edytować kod główny app.py na nowo utworzonej gałęzi tworząc nową ścieżkę strony.

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git checkout -b "demo"
Switched to a new branch 'demo'
```

```
from flask import Flask

# Create a Flask app instance
app = Flask(__name__)

# Define a routes for application
@app.route('/')
def hello_world():
    return 'Hello, World!'


# added rout for testing python CI
@app.route('/python')
def python():
    return 'Python CI is very cool'

#added rout for testin docker CI
@app.route('/docker')
def docker():
    return 'Docker Ci is super cool'


#last update testing CD process
@app.route('/deploy')
def deploy():
    return 'I hope i will get 5 for this funny project'


# Run the app on port 5000
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git checkout -b "demo"
Switched to a new branch 'demo'
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git add .
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git commit -m "final countdown"
[demo f60e0c8] final countdown
 1 file changed, 5 insertions(+)
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania> git push origin demo
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 12 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 472 bytes | 472.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'demo' on GitHub by visiting:
remote:   https://github.com/Pogromcamyszy/DEVOPS-projekt/pull/new/demo
remote:
To https://github.com/Pogromcamyszy/DEVOPS-projekt.git
 * [new branch]      demo -> demo
PS C:\Users\bozio\OneDrive\Pulpit\Projekt studia do wydania>
```


 demo had recent pushes 1 minute ago

[Compare & pull request](#)

 base: main

 compare: demo

✓ Able to merge. These branches can be automatically merged.














Add a title

final countdown


Add a description


Write

Preview

H B I           



Add your description here...

 Markdown is supported


 Paste, drop, or click to add files


Create pull request


final countdown #5

 Open Pogromcamyszy wants to merge 1 commit into `main` from `demo` 

 Conversation 0

 Commits 1

 Checks 0

 Files changed 1




Pogromcamyszy commented 1 minute ago

Owner ...

No description provided.



 `final countdown`

✓ f60e0c8



Require approval from specific reviewers before merging

[Rulesets](#) ensure specific people approve pull requests before they're merged.

Add rule



All checks have passed

3 successful checks

[Show all checks](#)



This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request




You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

 Try the new merge experience

← Build and Publish image to Docker Hub

✓ Merge pull request #5 from Pogromcamyszy/demo #3


 Summary

Jobs



✓ publish_images

Run details

 Usage

 Workflow file

Triggered via push 2 minutes ago

 Pogromcamyszy pushed  ea9e05d `main`

Status

Success

Total duration

1m 7s

Artifacts

—

Docker CD publish to dockerIO.yml

on: push

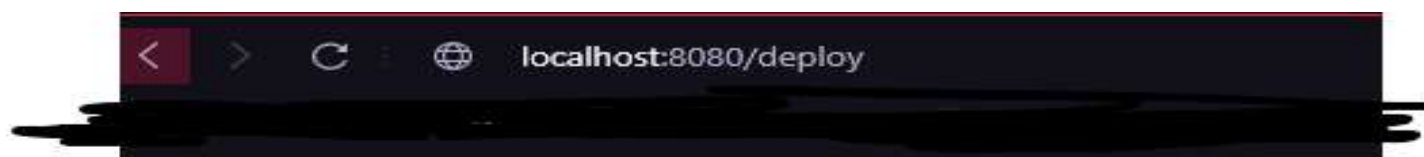


publish_images

57s


```
Terminal
docker.io/bozios11/flask-nginxpara:latest
PS C:\Users\bozio> docker run --name pokaz2 -p 8080:80 bozios11/flask-nginxpara
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server
instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
Press CTRL+C to quit
█
```

Figure 9: Przed uruchomieniem obrazu wykonałem ponownie pull request



I hope i will get 5 for this funny project

```
Terminal
PS C:\Users\bozio> docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED        STATUS        PORTS                    NAMES
e5cfc3491e72   bozios11/flask-nginxpara  "sh -c 'python3 /app..." 3 minutes ago  Up 3 minutes  0.0.0.0:8080->80/tcp      pokaz2
PS C:\Users\bozio>
```

Jak widać cały proces działa prawidłowo a rozszerzenia aplikacji zostały wysłane oraz zaktualizowane na docker hubie. Dostępne są dla każdego użytkownika za pomocą polecenia **docker pull bozios11/flask-nginxpara**

DZIĘKUJE ZA UWAGĘ :)