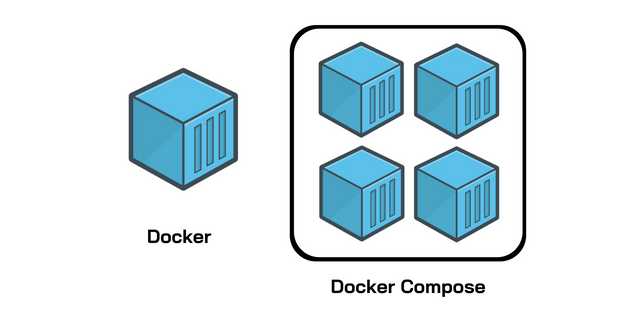
# **Docker Compose - Pierwsze kroki**

## **Wprowadzenie**

**Docker Compose** to narzędzie Docker służące do definiowania i uruchamiania aplikacji złożonych z wielu kontenerów. Dzięki Compose używasz pliku YAML do konfigurowania usług aplikacji i tworzenia wszystkich usług aplikacji na podstawie tej konfiguracji.

Pomyśl o docker-compose jako o zautomatyzowanym przepływie pracy z wieloma kontenerami. Compose to doskonałe narzędzie do programowania, testowania, przepływów pracy CI i środowisk przejściowych.



*Docker vs Docker Compose*

Przykład użycia nad którym się dzisiaj skupimy to utworzenie pliku YAML, w którym skonfigurujemy 2 powiązane ze sobą kontenery. Pierwszy będzie naszym serwerem Apache na otwartym porcie 8080, a drugi będzie bazą danych, z którą będziemy się łączyli z naszej aplikacji.

## **Instalacja**

Jeśli korzystasz z systemu Mac lub Windows z zainstalowanym Docker Desktop, to możesz pominąć ten krok.  
Docker Desktop dla ww systemów zawiera Docker Compose jako część integralną, a samą aplikację można pobrać z oficjalnej strony <https://docs.docker.com/get-docker/>.

W przypadku systemu linux trzeba zainstalować najpierw Docker Engine.

Po zainstalowaniu Docker Engine uruchom następujące polecenie, aby pobrać bieżącą stabilną wersję Docker Compose:

sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

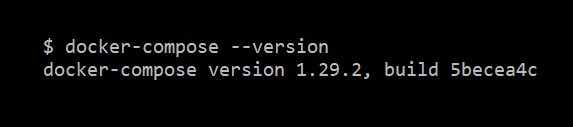
Na dzień tworzenia lekcji była to wersja 1.29.2. Jaka jest obecnie najnowsza wersja możesz sprawdzić tutaj: [Docker Compose release notes](https://docs.docker.com/compose/release-notes/)

Zastosuj uprawnienia dla pliku wykonywalnego:

sudo chmod +x /usr/local/bin/docker-compose

Teraz przetestuj instalację za pomocą następującego polecenia:

docker-compose --version



*Rezultat sprawdzenia wersji docker-compose*

## **Struktura pliku YAML**

Przykład struktury takiego pliku przedstawię w następnym punkcie. Zacznijmy od krótkiego opisu z czego może się składać tak utworzony plik:

**version ‘[wersja, np. 3]’**  
Określamy w ten sposób wersję docker compose, jakiej chcemy użyć. Dzięki temu Docker automatycznie zapewni nam odpowiednie funkcje. W chwili przygotowywania tej lekcji najnowsza wersja to 3.9.

**services**  
W tej sekcji definiujemy wszystkie kontenery, które chcemy utworzyć. W przykładach najpierw napiszemy skrypt tworzący jeden, a następnia dwa kontenery.

**app**  
Pod tym wyrazem umieszczamy nazwy usług, pod którymi mają funkcjonować będą kontenery.

**build**  
Określa lokalizację naszego pliku Dockerfile.

**ports**

Służy do mapowania portów kontenera. W przypadku, gdy np. na komputerze mamy uruchomiony serwer Apache, przez co mamy zajęty port 80, a nasz kontener również nasłuchuje na porcie 80, to możemy zmapować go na inny port, np. 8080 (w ten sposób rozróżniamy port hosta i port kontenera, o których pisaliśmy we wcześniejszym szkoleniu).

**volumes**

Sekcja służąca do montowania dysków w Dockerze. Możemy za jego pomocą z montować nasze katalogi z kodem, czy też z konfiguracją do kontenera.

**depends\_on**

Określa zależność kontenerów. Przydatna, gdy, np. chcemy wprowadzić kolejność uruchamiania kontenerów - najpierw baza danych, później serwer Apache.

**image**

Opcja przydatna, gdy nie mamy przygotowanego pliku Dockerfile, ale chcemy uruchomić usługę przy użyciu gotowego obrazu. Wówczas określamy lokalizację obrazu w sekcji image.

**environment**

Sekcja ta pozwala nam ustawić w kontenerze zmienne środowiskowe. Jest to podobna opcja do argumentu -e podczas uruchamiania kontenera za pomocą Docker-a.

## **Pierwszy skrypt Docker Compose**

Po prawidłowym zainstalowaniu Docker Compose, nadszedł czas nas stworzenie pierwszego skryptu. Na początek w celu zaznajomienia się ze strukturą docker-compose.yml, stwórzmy prosty serwis składający się z jednego kontenera.

W tym celu musimy w dowolnej lokalizacji utworzyć plik skryptu. W tym celu można użyć dowolnego edytora tekstowego. Ja w tym przypadku skorzystam z edytora nano. Kod aplikacji zostanie napisany w oparciu o PHP-a, ale przykład będzie na tyle generyczny, że bez problemu zaimplementujesz funkcjonalność również w innym środowisku programistcznym.

1. Tworzymy plik skryptu:

nano docker-compose.yml

1. Wklejamy do niego poniższy skrypt:

services:

app:

container\_name: app

image: php:7.4-apache

volumes:

- ./public\_html/:/var/www/html

ports:

- 8080:80

Skrypt tworzy nam kontener o nazwie **app** z istniejącego już obrazu **php:7.4-apache.**

Dzięki użyciu pola **container\_name** możemy odwoływać się do naszego kontenera za pomocą nazwy, a niekoniecznie za pomocą jego id (co za chwilę pokażę).

W **volumes** mamy możliwość mapowania katalogów lokalnych na wirtualny system plików Dockera. W przypadku Apache - publiczny katalog znajduje się w lokalizacji ‘**/var/www/html’.** Dlatego w celu publikacji naszego katalogu **’public\_html`** mapujemy go do ww. katalogu.

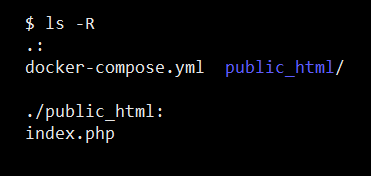
Ostatnie ustawienie w naszym skrypcie dotyczy portów. W naszym przypadku zmapowaliśmy standardowy port http 80 na port 8080. Nie jest to konieczność, ale zrobiłem to w celu zaprezentowania możliwości. Przypomnę Ci tylko, że takie rozwiązanie zapewnia rozróżnienie portów między tymi, które przysługują hostowi a tymi, które dotyczą kontenera.

1. Tworzymy katalog ***public\_html*** i w nim tworzymy plik ***index.php*** z przykładową zawartością:

mkdir public\_html

echo "<?= ‘Hello World' ?>" >> public\_html/index.php

Struktura plików powinna wyglądać następująco:



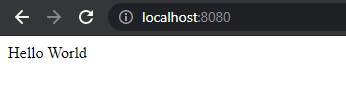
*Struktura plików katalogu*

1. Uruchamiamy nasz skrypt docker-compose:

docker-compose up -d

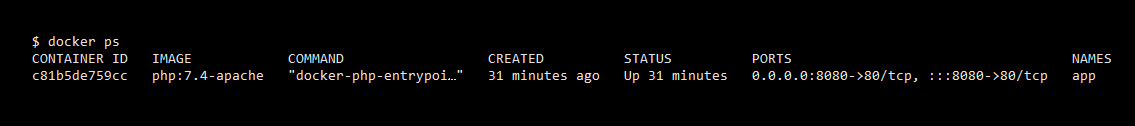
Dzięki opcji **-d** nasz kontener po uruchomieniu będzie działał w tle. Usunięcie tej opcji spowoduje wyświetlanie logów. A po wyjściu z logów za pomocą skrótu **CTRL + C**, kontener automatycznie się zatrzymuje.

1. Po wpisaniu w przeglądarce adresu <http://localhost:8080> powinniśmy ujrzeć nasz przykładowy tekst z pliku index.php:



1. Sprawdźmy listę uruchomionych kontenerów:

docker ps



*Lista uruchomionych koneterów*

Widzimy wyżej, że poza id kontenera mamy też nazwę. Widać również użyty obraz oraz mapowanie portów.

1. Używając wcześniej określonej nazwy kontenera możemy przejść do kontenerowego terminala:  
     
   W przypadku systemu Linux, MAC OS oraz Windows 10 Pro:

docker exec -it **app** bash

W przypadku innej wersji Windows niż Windows 10 Pro:

winpty docker exec -it **app** bash

Po przejściu do dockerowego terminala powinniśmy się znaleźć w katalogu startowym w którym znajduje się nasz plik index.php

Aby wyjść z kontenera wystarczy wpisać polecenie **exit** i zatwierdzić enterem.

1. Aby zatrzymać nasz kontener użyjemy polecenia:

docker-compose down

**Zadanie:**

Zaimplementuj powyższa funkcjonalność programu w dowolnym, wybranym przez siebie, języku.

## **Coś trudniejszego?**

Pierwszą, prostą aplikację już mamy. Nie byłbym jednak sobą, gdybym nie pokazał Ci również nieco bardziej złożonego przykładu aplikacji. Otóż utworzymy teraz dwa kontenery: jeden z web-ową aplikacją napisaną w Pythonie. W drugim natomiast uruchomiona będzie Redis-owa baza danych noSQL. Ponadto do działających kontenerów, dodamy, tzw. volume. Jest to system plików mount-owany bezpośrednio na Docker-owym kontenerze. Dzięki temu będziemy mogli trwale zapisywać dane i współdzielić je pomiędzy kontenerami. Bądź bowiem świadomy tego, iż opierając się na standardowym mechanizmie zapisu-odczytu informacji przez kontener (tzw. **Union File System**), dane zmodyfikowane przez uruchomiony kontener będą zapamiętane tylko do momentu usunięcia kontenera. Jego ewentualny późniejszy rebuild, spowoduje utracenie wszystkich efektów.

Wprowadzenie volumes w obrębie naszych kontenerów rozwiąże właśnie ten problem i dodatkowo umożliwi wprowadzanie zmian “on fly” (zmiana dowolnego pliku i chęć zaaplikowania zmian, nie będzie wymagała od nas przebudowania całego kontenera).

Proces ten przeprowadzę w Pythonie.

Zacznijmy od utworzenia pliku z aplikacją: main.py.

**Utworzenie dowolnego pliku (polecenie terminal)**

| nano <nazwa\_pliku> |
| --- |

**Zawartość pliku main.py**

| import time  import redis  from flask import Flask  app = Flask(\_\_name\_\_)  cache = redis.Redis(host='redis', port=6379)  def hit():  retries = 5  while True:  try:  return cache.incr('hits')  except redis.exceptions.ConnectionError as exc:  if retries == 0:  raise exc  retries -= 1  time.sleep(0.5)  @app.route('/')  def hello():  count = hit()  return f'Hello World! I have been seen {count} times.' |
| --- |

* Redisowy kontener będziemy uruchamiali na porcie 6379
* Funkcja hit służy nam do zwiększania counter-a odwiedzin (cache.incr(‘hits’)) oraz wyrzucania wyjątku aplikacji w momencie, gdy użytkownik dokonał 5-krotnej próby połączenia z aplikacją w odstępach co 0.5 sekundy, ale ani razu nie udało mu się nawiązać komunikacji
* Aplikacja zawiera jeden główny widok hello, który odpowiada za wywoływanie funkcji hit(), zwiększanie licznika odwiedzin strony głównej i wyświetlanie takiego counter-a.

**Plik requirements.txt**

| flask  redis |
| --- |

**Plik Dockerfile**

| FROM python:3.7-alpine # 1  WORKDIR /app # 2  ENV FLASK\_APP=main.py # 3  ENV FLASK\_RUN\_HOST=0.0.0.0 # 4  COPY requirements.txt requirements.txt # 5  RUN pip install --no-cache-dir -r requirements.txt # 6  EXPOSE 5000 # 7  COPY . . # 8  CMD ["flask", "run"] # 9 |
| --- |

**# 1** Naszą aplikację zbudujemy w oparciu o alpine-owy obraz python 3.7

**# 2** Katalog roboczy wewnątrz kontenera ustawiliśmy na ścieżkę /app

**# 3**, **# 4** Z racji, że wewnątrz samego kodu aplikacji nie zdefiniowaliśmy choćby takiej właściwości jak adres hosta, to musimy określić te atrybuty wewnątrz Dockerfile-a (stąd też określenie wartości FLASK\_APP oraz FLASK\_RUN\_HOST)

**# 5** Plik z potrzebnymi do zainstalowania zależnościami kopiujemy do kontenera, do pliku o tej samej nazwie

**# 6** Instalujemy potrzebne zależności z wyżej utworzonego pliku requirements.txt

**# 7** Ustawiamy port hosta…

**# 8** Kopiujemy do oddzielnej warstwy całą zawartość katalogu, w którym się znajdujemy

**# 9** W końcu uruchamiamy naszą aplikację poleceniem flask\_run (nie musimy podawać dodatkowych parametrów z racji ustawienia zmiennych środowiskowych we wcześniejszych krokach).

**Plik docker-compose.yml**

| version: “3.9”  services:  web:  build: .  ports:  - “5000:5000”  redis:  image: “redis:alpine” |
| --- |

A powyżej serce całego przykładu. Aby móc operować na dwóch komunikujących się ze sobą kontenerach, musimy utworzyć plik docker-compose.yml, w którym określimy jego:

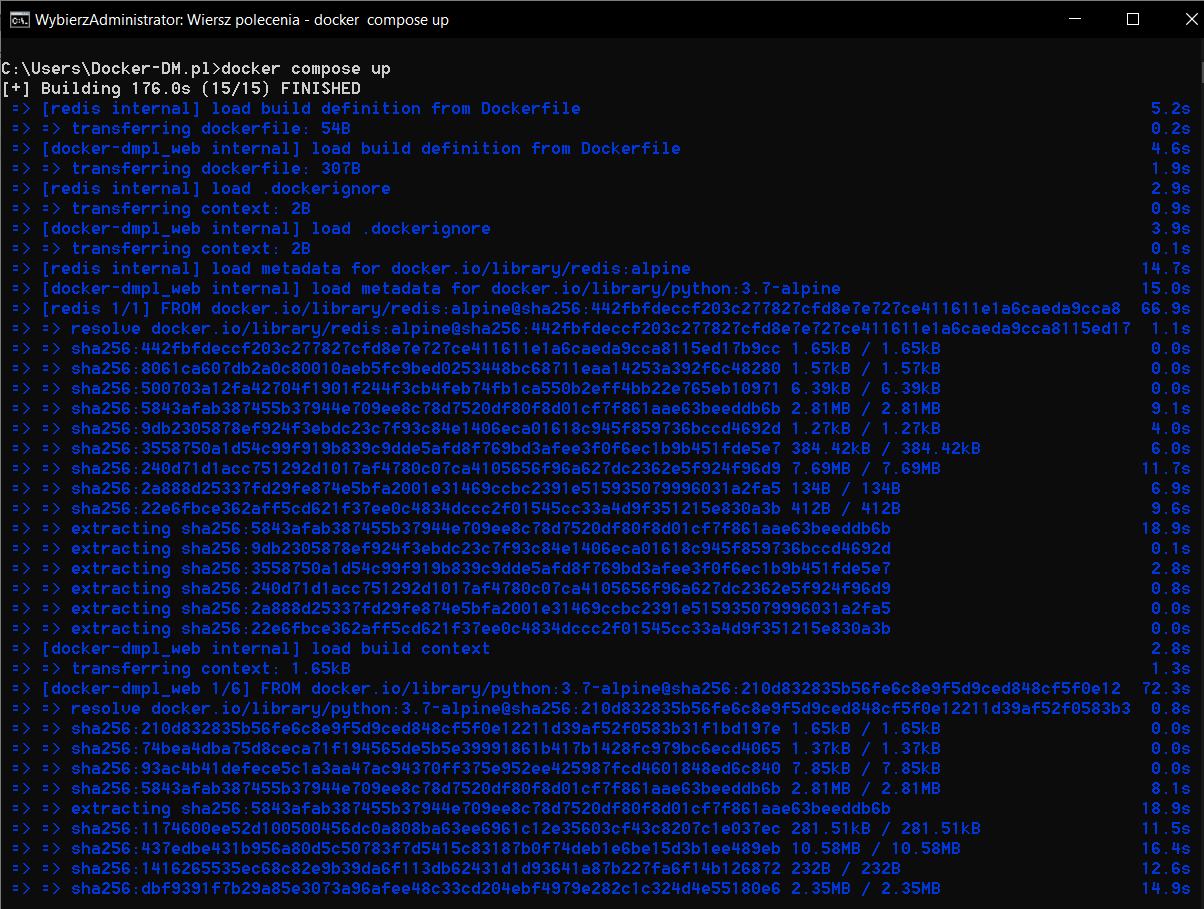
1. version - określenie wersji docker-compose zgodnej z wersją Docker Engine
2. services - klucz umożliwiający zdefiniowanie serwisów (kontenerów) działających w obrębie całego ekosystemu
3. web, redis - to dowolne nazwy naszych kontenerów; web to kontener, w którym uruchomiona jest aplikacja Flask, a redis przechowuje instancję działającej bazy Redis
4. build, ports, image - dodatkowe pola zapisywane w obrębie konkretnych serwisów; określają kolejno, gdzie znajduje się Dockerfile, z którego ma zostać zbudowany obraz aplikacji, na jakich bindowanych portach ma odpalać się aplikacja, z jakich zewnętrznych obrazów skorzystać, aby zainstancjonować kontener.

Spróbujmy uruchomić Nasze kontenery, wpisując w konsolę polecenie docker compose up (pamiętaj, aby znajdować się w katalogu z Dockerfile i całym projektem; w moim przypadku jest to ścieżka C:\Users\Docker-DM.pl\docker-compose).

**Polecenie uruchamiające**

| docker compose up |
| --- |

**Uruchomienie kontenerów**



**Aplikacja**



**Dodanie volume**

Nadeszła pora na dodanie woluminu do naszej aplikacji. Cel takiego rozwiązania przedstawiłem Ci już we wstępie tej sekcji. Przypomnę tylko, że umożliwi on nam nam choćby wprowadzanie zmian w kodzie “on fly” i dzięki temu nie będziemy musieli za każdym razem przebudowywać obrazu, chcąc zmienić działanie kontenera.

Aby dodać wolumin, wystarczy zmodyfikować plik docker-compose.yml, przez dodanie następujących linii wewnątrz serwisu web:

| volumes:  - .:/app  environment:  FLASK\_ENV: development |
| --- |

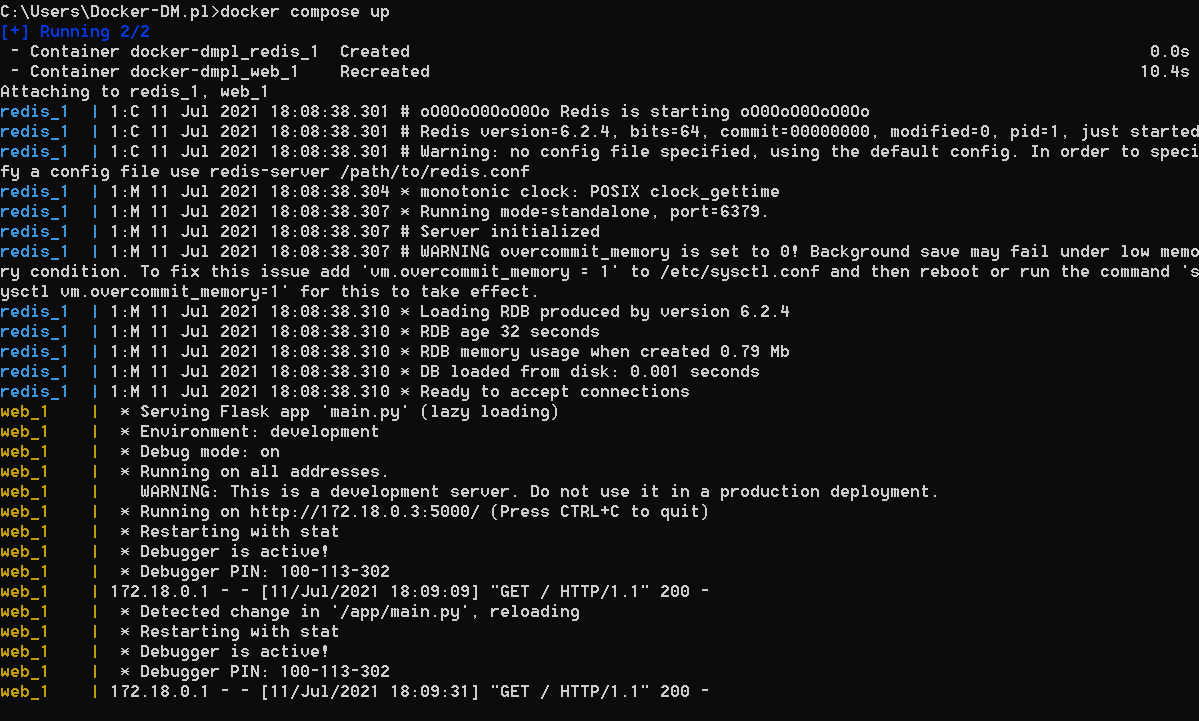
Gdzie /app to nazwa naszego katalogu z aplikacją (który zdefiniowaliśmy wewnątrz Dockerfile).

Pole environment służy do zdefiniowania, w jakim środowisku uruchamiamy aplikację. Pod zmienną FLASK\_ENV umieściliśmy wartość development z racji, iż aplikacja będzie uruchamiana w środowisku projektowym. Pamiętaj, aby przy pushowaniu zmian na serwer produkcyjnym, zmienić ten atrybut z development na production!

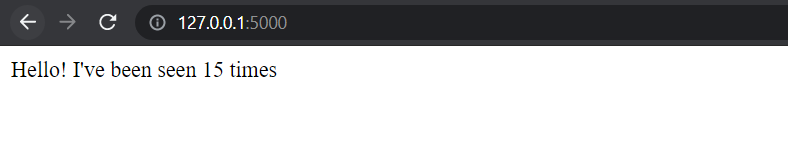
Ostatecznie, nasz zmodyfikowany plik docker-compose.yml będzie wyglądał następująco:

| version: “3.9”  services:  web:  build: .  ports:  - “5000:5000”  volumes:  - .:/app  environment:  FLASK\_ENV: development  redis:  image: “redis:alpine” |
| --- |

Uruchomimy aplikację:



I mając tak uruchomiony kontener, wprowadźmy dowolną zmianę wewnątrz pliku main.py. Ja zamienię komunikat: *’Hello! I've been seen {count} times.’* na *’****Hi****! I have been seen {count} times.'.*



*Aplikacja przed modyfikacją*



*Aplikacja po modyfikacji*

Gdy zapiszesz takie zmiany w pliku i wrócisz do uruchomionej aplikacji, ujrzysz na bieżąco update-ującą się zmianę w napisie. Wszystko odbywa się on live! Gdybyśmy nie dodali woluminu do docker-compose, zaaplikowanie zmian byłoby możliwe dopiero po przebudowaniu całego obrazu, z którego uruchamiany jest kontener.

**Flow, ale w innym języku (przykład PHP)**

Czas na kolejny przykład! Tym razem utworzymy serwis korzystający z serwera Apache, łączący się z bazą danych PostgreSQL. Będzie wymagało to doinstalowanie pewnych rozszerzeń do naszego serwera obsługującego PHP.

1. W tym celu stworzymy osobny plik o nazwie **Dockerfile**, który umieścimy w katalogu głównym naszego projektu:

FROM php:8.0-apache

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

&& docker-php-ext-install pdo\_mysql \

&& docker-php-ext-configure pgsql -with-pgsql=/usr/local/pgsql \

&& docker-php-ext-install pdo pdo\_pgsql pgsql

Dockerfile to tekstowy skrypt służący do tworzenia obrazu kontenera. Plik Dockerfile działa na warstwach. Pierwsza warstwa zaczyna się od słowa FROM i określa, którego gotowego obrazu użyjemy do zbudowania obrazu.

Poza pobraniem gotowego obrazu pozwala na instalowanie odpowiednich rozszerzeń - w naszym przypadku zainstalowaliśmy bibliotekę postgreSQL.

1. Nadpiszmy nasz docker-compose.yml:

services:

app:

container\_name: app

build: ./

volumes:

- ./public\_html/:/var/www/html

ports:

- 8080:80

networks:

- network\_internet

- network\_no\_internet

depends\_on:

- db

db:

image: postgres:11

container\_name: db

ports:

- 5432:5432

environment:

- POSTGRES\_USER=pg\_user

- POSTGRES\_DB=pg

- POSTGRES\_PASSWORD=secret

volumes:

- postgres:/data/postgres

networks:

- network\_no\_internet

networks:

network\_internet:

driver: bridge

network\_no\_internet:

driver: bridge

internal: true

volumes:

postgres:

Opcja **build** określa, w którym miejscu znajduje się plik Dockerfile. Dodany został również nowy kontener o nazwie **db**. Jest to nasza baza danych, do której dostęp ma tylko kontener o nazwie **app.** Dostęp ogranicza sieć wewnętrzna o nazwie **network\_no\_internet**. Jest to jedna z ważnych kwestii dotyczących bezpieczeństwa. Kontener o nazwie **app** korzysta z dodatkowej sieci o nazwie **network\_internet**, która nie ma ograniczenia dostępu, dzięki czemu możemy połączyć się z serwerem Apache.

1. Zmodyfikujmy również nasz plik public\_html/index.php

<?php

if (!$connection = pg\_connect("host=db dbname=pg user=pg\_user password=secret")) {

$error = error\_get\_last();

echo "Nie udało się połączyć z bazą. Napotkano błąd: ". $error['message']. "\n";

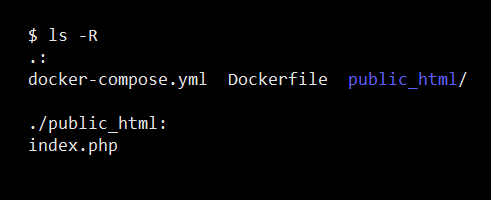
} else {

echo "Połączono z bazą danych";

}

Skrypt łączy się do naszej bazy PostgreSQL i pobiera listę tabel. Jest to szybki test połączenia z bazą. Jak w powyższym przykładzie widać, nazwa kontenera może również służyć jako nazwa hosta **(host=db)**.

W tym momencie struktura plików naszego katalogu powinna wyglądać następująco:

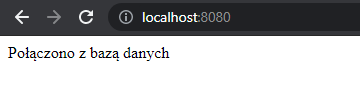


1. Po wszystkich modyfikacjach możemy ponownie uruchomić nasze kontenery:

docker-compose up -d --build

Opcja **--build** wymusza przebudowanie kontenerów.

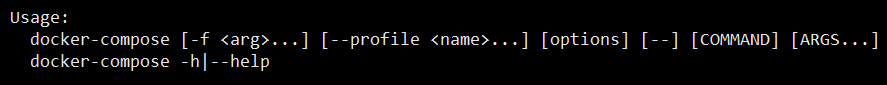
1. Jeśli wszystko zadziała poprawnie to po przejściu do przeglądarki powinniśmy zobaczyć taką informację:



## **Więcej o poleceniach Docker Compose**

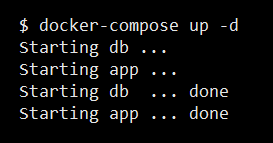
1. **docker-compose -h**

Pomoc, w której znajdziemy opisane wszystkie dodatkowe opcje, oraz jak powinniśmy konstruować polecenia docker-compose :)



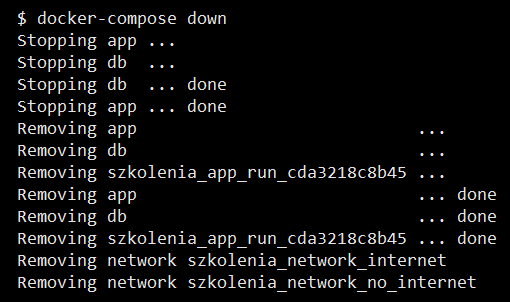
1. **docker-compose up -d**

Buduje obrazy, jeśli nie są zlokalizowane lokalnie i uruchamia kontenery. Jeśli obrazy są już zbudowane, to zostaną uruchomione kontenery.



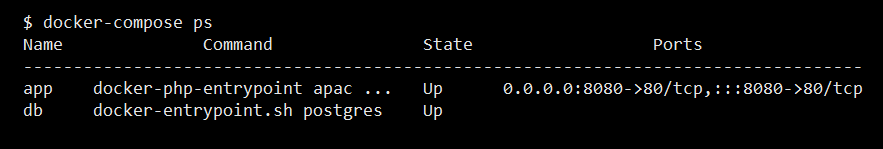
1. **docker-compose down**

Utworzy kontenery z obrazów zbudowanych dla usług wymienionych w pliku docker-compose.yml.



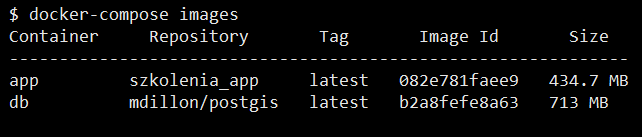
1. **docker-compose ps**

Wyświetla listę kontenerów zdefiniowanych w pliku docker-compose.yml



1. **docker-compose images**

Polecenie listuje użyte obrazy w naszym docker-compose.yml



1. **Opcja -f**

Pozwala na wybranie pliku konfiguracyjnego. Domyślnie plik konfiguracji nazywa się docker-compose.yml, ale za pomocą tej opcji możemy określić nazwę wczytywanego pliku.

docker-compose -f docker-compose.yml -f docker-compose-test.yml up -d

Jak widać w przykładzie, dzięki opcji -f możemy uruchomić kilka skryptów YAML jednocześnie.

7. **Opcja --build**

Dodanie tej flagi do naszego polecenia uruchamiającego kontenery spowoduje, że za każdym razem będzie budowany nowy obraz (z ewentualnie zmienioną strukturą plików). Zwolni nas to z konieczności ręcznego usuwania obrazu, aby został przebudowany, a zmiany wprowadzone lokalnie (bez zamountowanego woluminu) zaaplikowane.

| docker-compose up <pozostałe opcje> –build |
| --- |