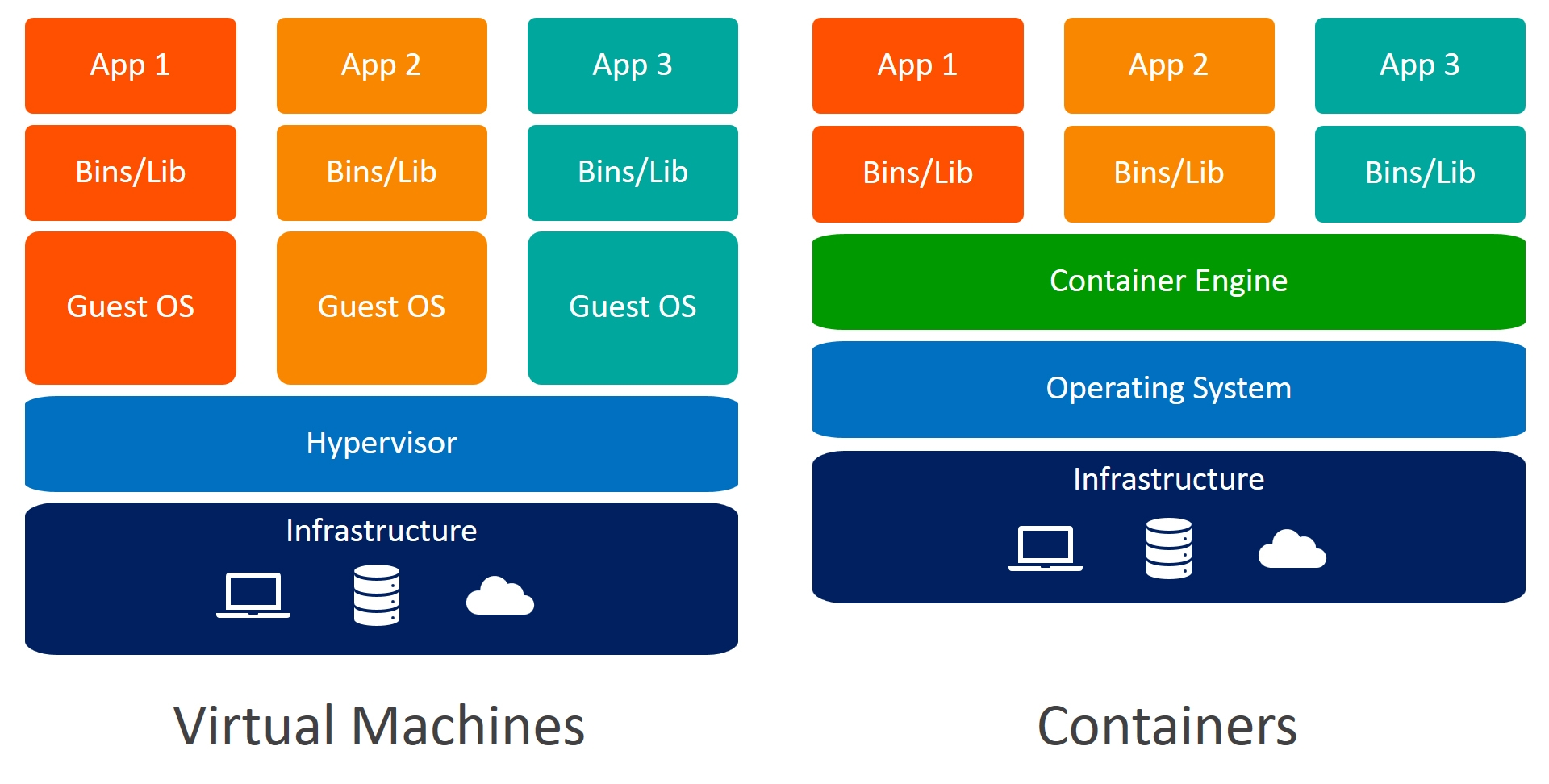
**Docker - czym jest? Pierwsza aplikacja.**

**Czym jest Docker?**

Szkolenie to zacznijmy od postawienia sobie prostego pytania - jak sprawić, aby dana aplikacja działała niezależnie od komputera/hosta, na którym jest uruchamiana? Wyobraź sobie sytuację, w której korzystasz ze środowiska z zainstalowaną wersją Pythona, np. 3.9 i serwera Apache. Nagle okazuje się, że Twoją aplikację musisz przenieść na inny serwer, na którym zainstalowane zostały nieco starsze wersje modułów Pythona, czy Apache-a. Po zmigrowaniu, Nasza aplikacja z oczywistych powodów się nie uruchomi i konieczna będzie rekonfiguracja środowiska uruchomieniowego. Niewątpliwie będzie to mozolny proces i żaden z developerów nie chciałby go doświadczyć. Pytanie wówczas jakie się pojawia, to jak zaradzić temu problemowi?

Jednym z rozwiązań jest stworzenie maszyny wirtualnej w pełni zgodnej ze środowiskiem, na którym budowaliśmy aplikację. Nie jest to jednak uniwersalne rozwiązanie i czasami może być nieefektywnym wyjściem. Na jedną wirtualną maszynę jest bowiem przydzielane oddzielne jądro systemowe, co zużywa więcej zasobów systemowych i czasami powoduje, że takie rozwiązanie staje się nieoptymalne.

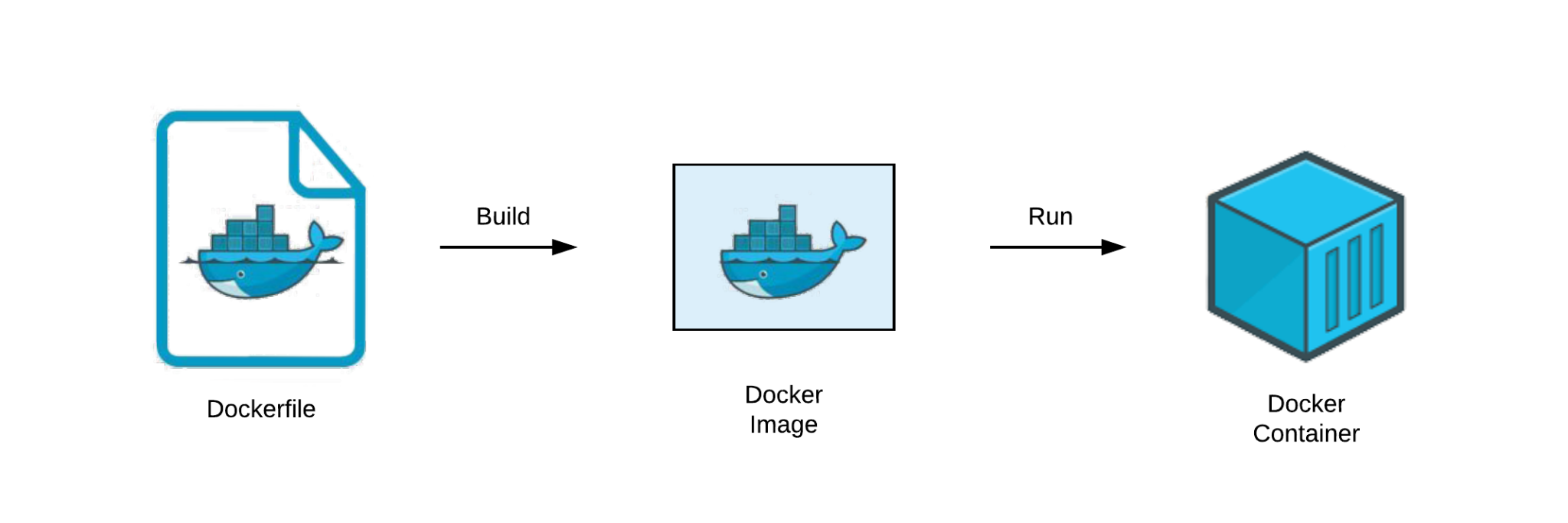
I wtedy, ubrany cały na biało, wchodzi do gry konteneryzacja, która również zapewnia izolację między hostem a utworzonym środowiskiem, ale wszystko to odbywa się na jednym jądrze systemu. Konteneryzacja więc nie wykorzystuje tzw. hypervisor-a (narzędzia niezbędnego do zarządzania procesami wirtualizacji między wirtualnymi maszynami), a więc środowisko działa z dużo większą prędkością i dostępnością. Dzieje się tak, ponieważ czas procesora nie jest wykorzystywany na obsłużenie dodatkowej warstwy abstrakcji, jaką jest wspomniany hypervisor. Uruchamianie wirtualnych środowisk (kontenerów) jest traktowane tak, jakby włączane były pojedyncze aplikacje. Niesamowite udogodnienie, które możemy realizować przy użyciu, np. Dockera.



**Więcej szczegółów o Dockerze**

Zagłębiając się w temat Dockera, musimy zacząć od zrozumienia, czym są takie pojęcia jak Docker Image, Docker Container oraz Dockerfile.

Gdybyśmy chcieli zobrazować, jakie zależności występują między wymienionymi aspektami Dockera, to możemy wykorzystać następujący schemat:

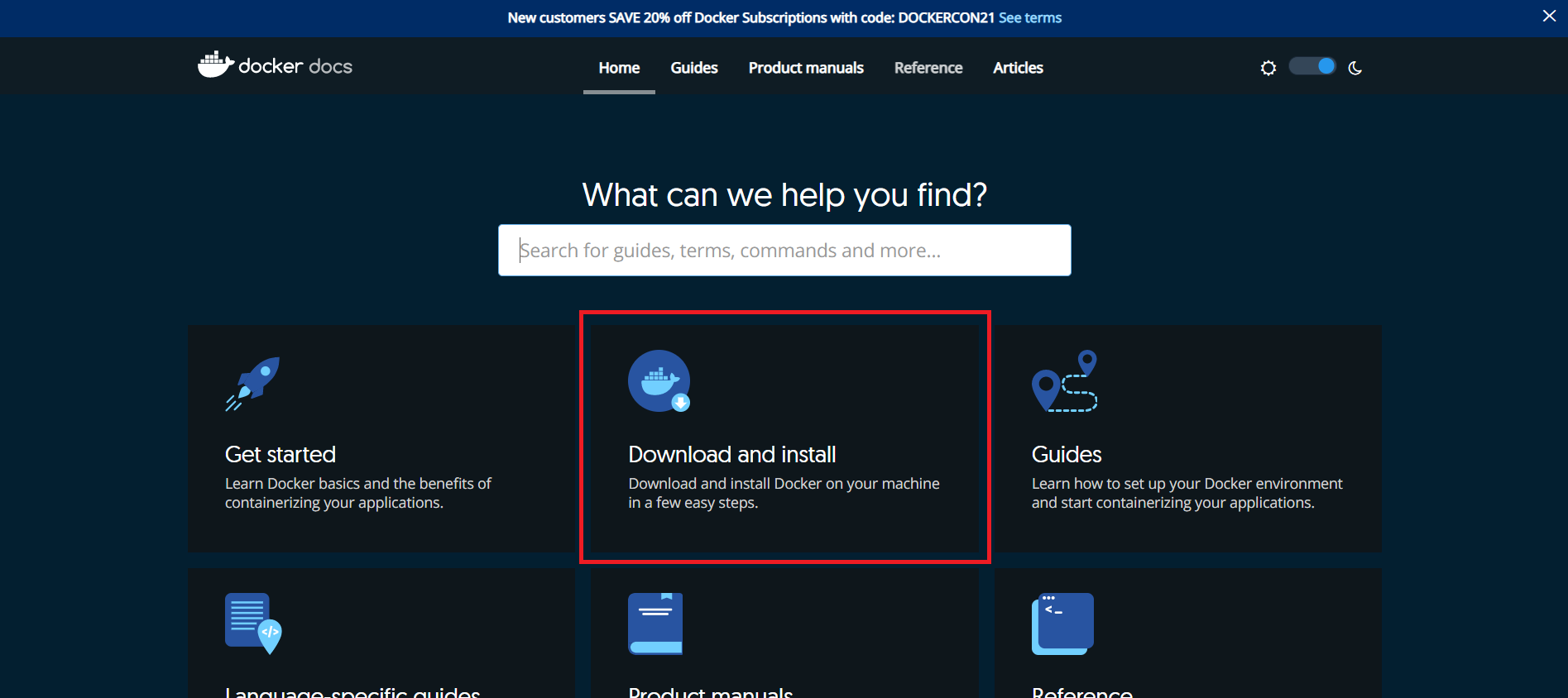


1. Dockerfile - to plik będący bazą dla Dockera. Jest to niejako plik konfiguracyjny, w którym określamy m.in jakie gotowe obrazy (poznasz niżej, czym jest obraz) chcemy wykorzystać w ramach uruchomionego kontenera, jak mają być skonfigurowane instancje uruchamianej bazy danych, na jakim porcie ma nasłuchiwać aplikacja etc.
2. Docker Image to niejako szablon, przepis, z którego możemy tworzyć jeden lub więcej kontenerów. Obrazy Dockera powstają na podstawie Dockerfile-a. Jeżeli natomiast nie chcemy budować własnego obrazu, mozemy również wykorzystać gotowce z tzw. Docker Hub-a (o którym więcej za chwilę). W odniesieniu do omawianych obrazów, możemy mówić również o tzw. warstwach. Najprościej ujmując - kolejne warstwy obrazu tworzą się wraz z wprowadzaniem nowych modyfikacji do Naszej aplikacji. Przez modyfikację rozumiem, np. dodanie katalogu, w którym znajduje się cała konfiguracja i zależności serwera Apache. Takie rozwiązanie umożliwia budowanie obrazów w o wiele szybszy sposób i choćby efektywne przywracanie wprowadzonych zmian bez naruszania działającego systemu zależności. Temat warstw rozszerzę w dalszych punktach tego szkolenia.
3. Docker Container to działająca instancja obrazu Dockerowego, która jest izolowana i wyodrębniona od pozostałych kontenerów. W ramach każdego kontenera możemy uruchamiać dowolny zestaw procesów.

**Instalacja Dockera**

Uzbrojeni w taką dawkę wiedzy, możemy przejść już do praktyki i zainstalować Dockera. Miej na uwadze fakt, iż proces instalacji będzie wyglądał różnie - w zależności od tego, z jakiego systemu operacyjnego korzystasz. Na szczęście oficjalna dokumentacja Dockera przeprowadzi Nas krok po kroku przez ten proces i dzięki temu jestem zwolniony z dogłębnego tłumaczenia Ci, jak prawidłowo przeprowadzić ten proces.

1. Przejdźmy na oficjalną stronę dokumentacji Dockera: <https://docs.docker.com/> i kliknijmy w zaznaczoną opcję:



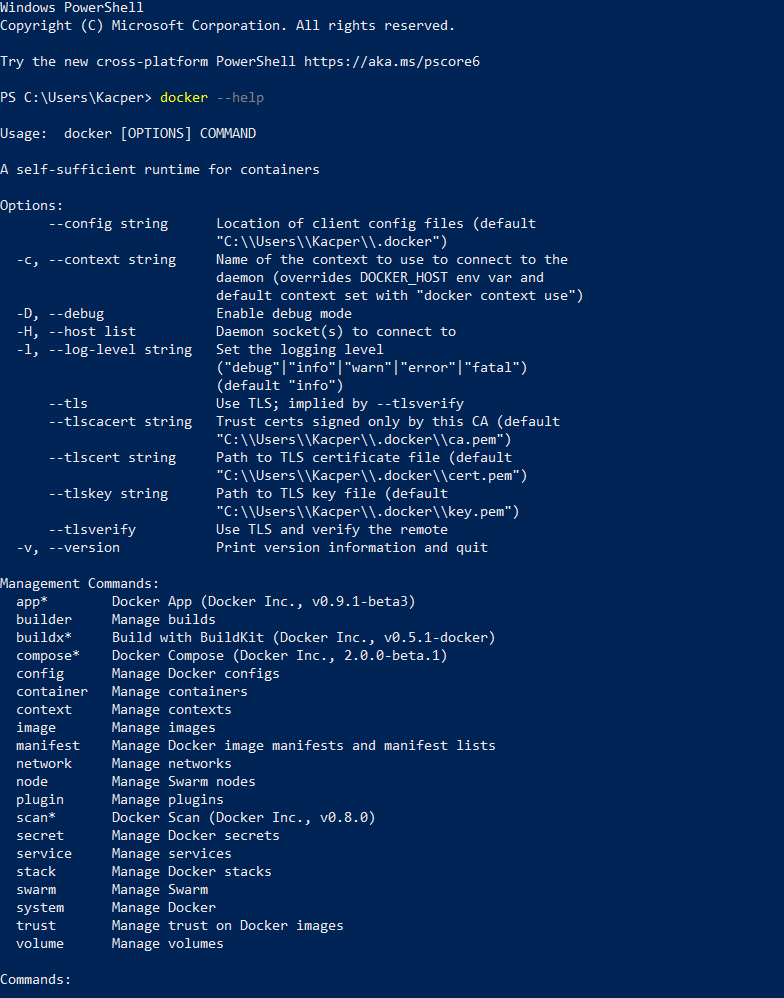
Następnie, w zależności od systemu, wybierz właściwą instrukcję instalacji. Ja operuję na Windowsie, dlatego też na tym systemie będę wykonywał wszystkie operacje.



Po pobraniu Docker Desktop, przejdźmy do jego instalacji. Nie będę Cię przeprowadzał przez ten proces, ponieważ jest on dość intuicyjny i nie wymaga dużego obeznania.

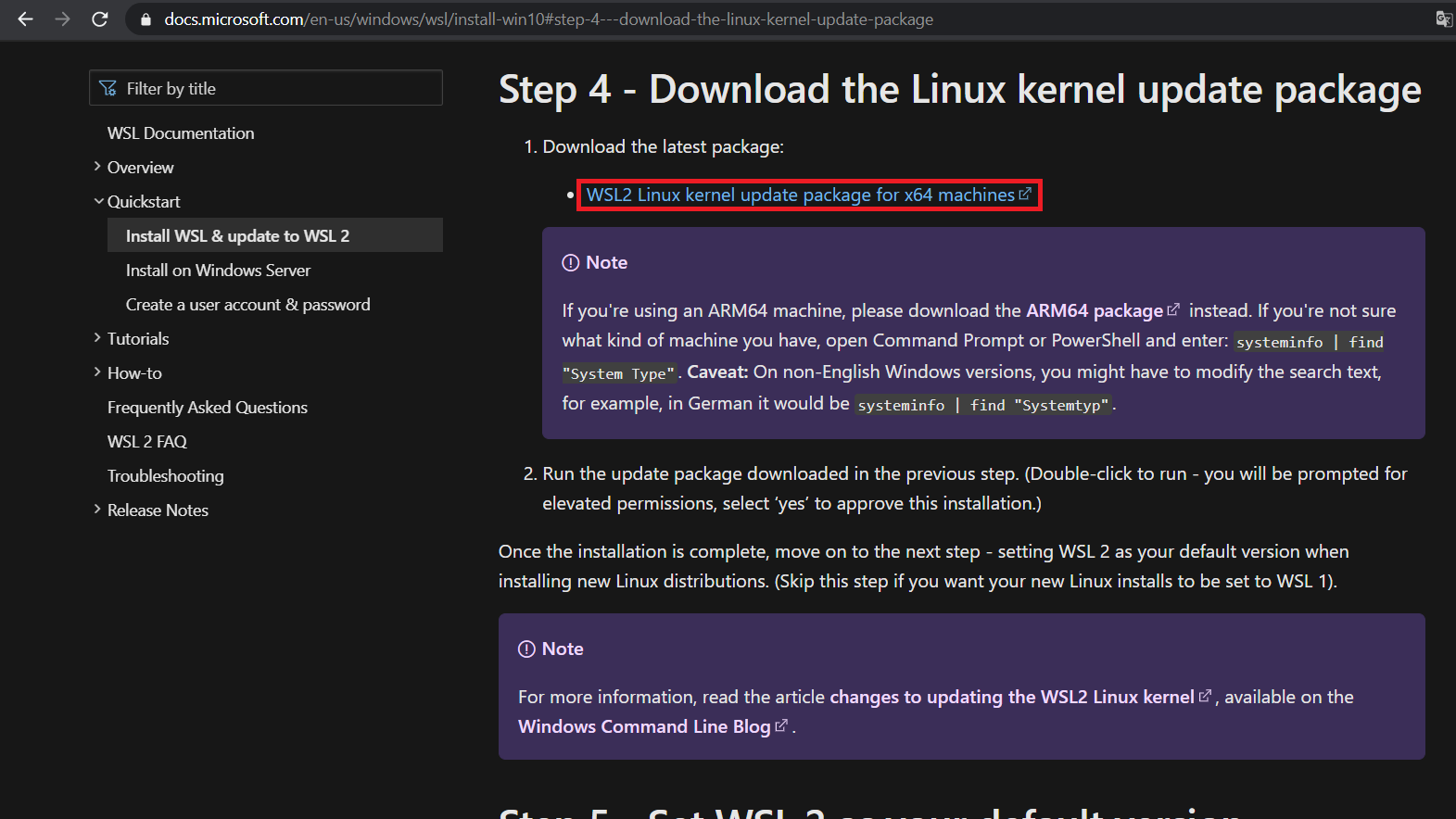
Powinieneś wiedzieć, że Docker Desktop dostarcza nam zestaw kilku narzędzi, bo oprócz samego Dockera, zyskujemy również dostęp do: Docker Engine, Docker CLI client, Docker Compose, Notary, Kubernetes, and Credential Helper. Nazwy te mogą być na ten moment dość enigmatyczne, jednak wiedz, że są to toolsy znacznie usprawniające pracę DevOpsa i szeroko wykorzystywane przy deploymentu aplikacji i budowaniu efektywnej architektury serwisu.

Po zainstalowaniu wszystkich zależności, możesz sprawdzić, czy docker został poprawnie zainstalowany, wpisując w PowerShell-u polecenie docker --help. Jeżeli otrzymałeś efekt taki jak ja, to znaczy, że wszystko zostało prawdopodobnie zainstalowane:



UWAGA:

1. Często w celu poprawnej instalacji konieczny jest restart systemu.
2. Jeżeli polecenia, które za chwilę wykonamy nie będą się poprawnie wykonywały na Twoim systemie, to może być konieczne doinstalowanie paczki spod linku: [Install WSL on Windows 10](https://docs.microsoft.com/en-us/windows/wsl/install-win10#step-4---download-the-linux-kernel-update-package)



**Architektura Dockera i pierwsze komendy**

Do tej pory nie omówiliśmy jeszcze, jak przeprowadzana jest komunikacja w obrębie systemu Docker-owego. Jak się okazuje, Docker związany jest z architekturą klient-serwer. Możemy w niej wyróżnić takie komponenty jak: Docker Client, Docker Daemon oraz Docker Registry. Pierwsze dwa działają na tym samym systemie lub Docker Client może być zdalnie podłączony do Docker Daemon-a.

Zadaniem Docker Clienta jest interakcja z użytkownikiem za pomocą poleceń i przekazywanie zadań do wykonania do Docker Daemon-a. Komponent ten zarządza obrazami, kontenerami i serwisami Dockerowymi.

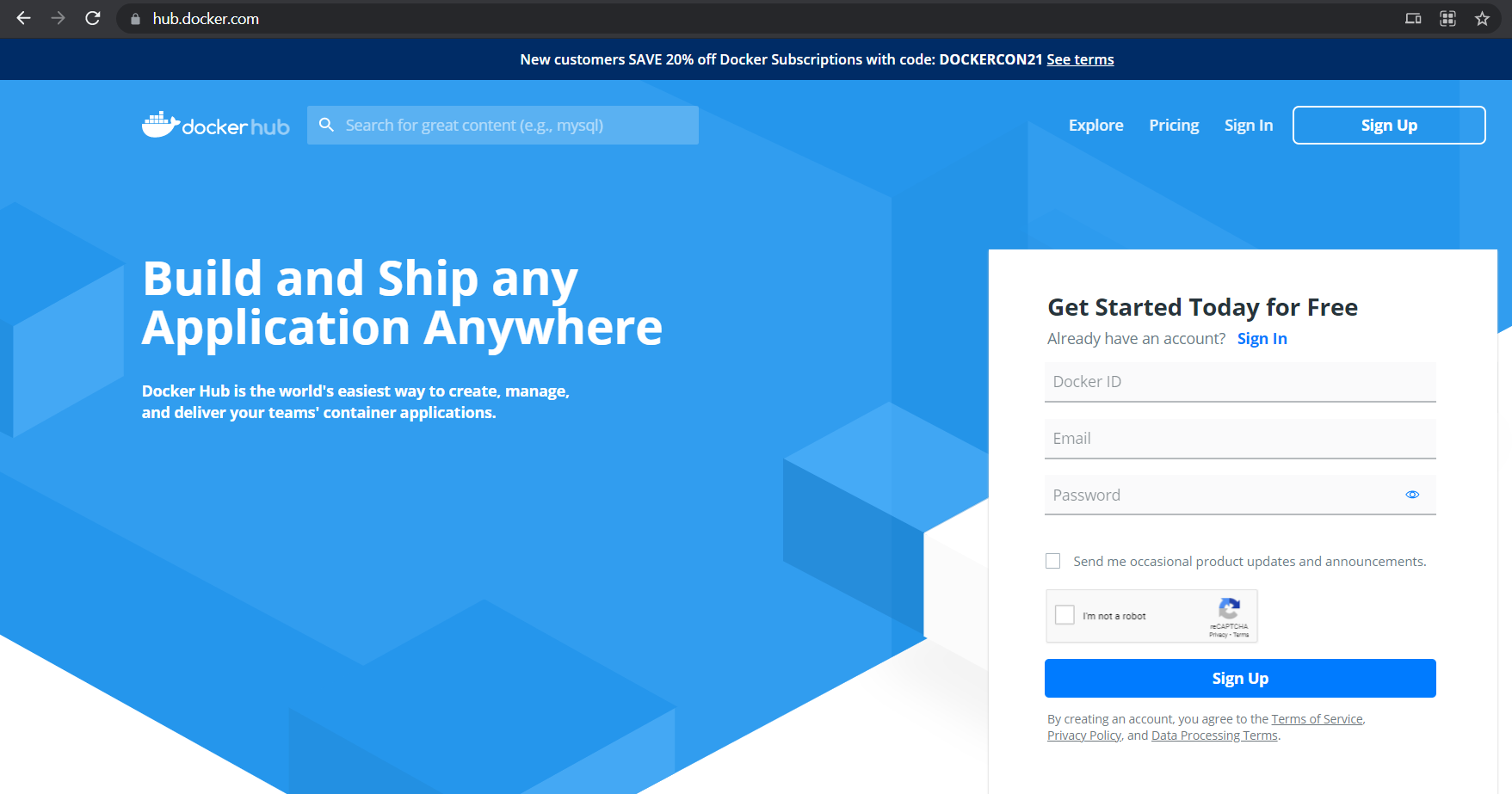
Docker Registry to natomiast storage obrazów dockerowych - przykładem jest Docker Hub będący publicznym serwisem, z którego możesz pobierać gotowe obrazy.

Z klientem Dockera, możesz się komunikować za pomocą tak podstawowych poleceń jak:

* **docker pull <*nazwa\_obrazu*>** - pobranie obrazu
* **docker run <*nazwa\_obrazu lub id\_obrazu*>** - zainstancjonowanie kontenera na podstawie obrazu
* **docker ps** - wyświetlenie uruchomionych kontenerów
* **docker start <*id\_kontenera*>** - uruchomienie kontenera (kontener musi już być zainstancjonowany poleceniem docker run)
* **docker stop <*id\_kontenera*>** - zatrzymanie działającego kontenera
* **docker images** - wyświetlenie dostępnych obrazów

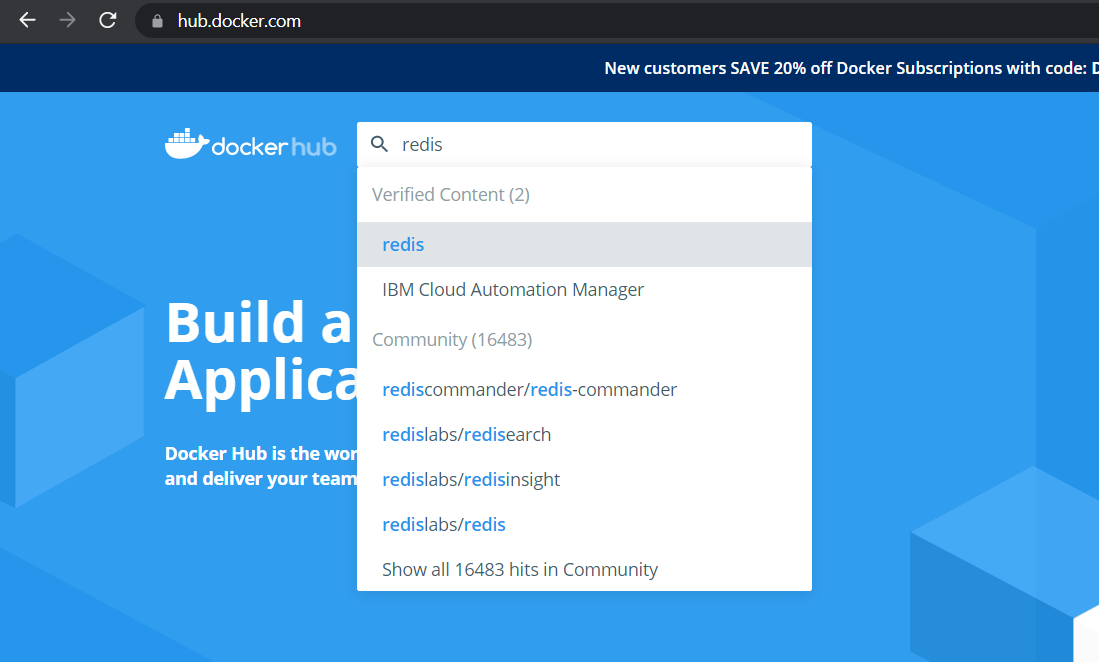
Zacznijmy od przetestowania polecenia docker pull. Posłuży Nam ono do pobrania gotowego obrazu z serwisu Docker Hub. Po pobraniu, zbudujemy z obrazu kontener i go uruchomimy.

Zanim wykonamy polecenia, przyjrzyjmy się jeszcze, jak wygląda serwis Docker Hub: <https://hub.docker.com/>

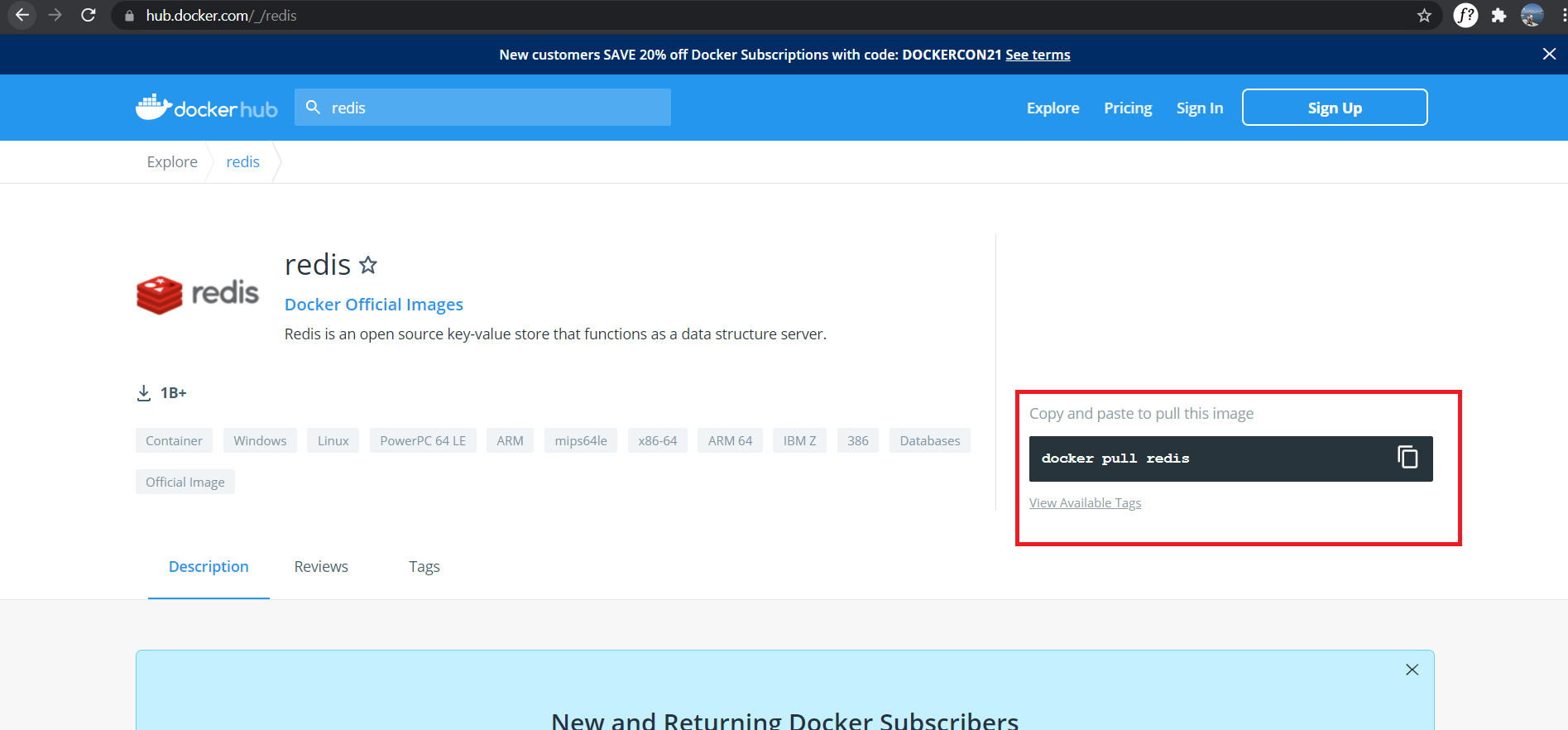


Docker Hub to zbiór gotowych obrazów, które możemy łatwo przeglądać i wybierać z nich dowolny. Pierwszym obrazem, który pobierzemy i na podstawie którego uruchomimy kontener będzie **redis**.

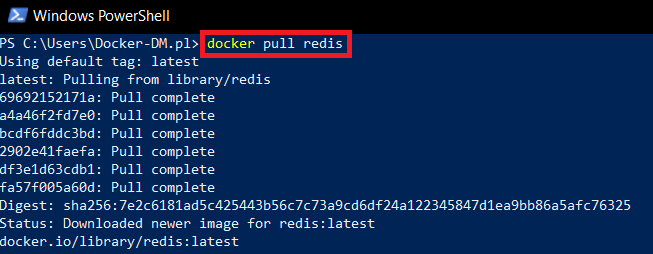
Możemy go znaleźć, wpisując w zakładkę *Search for great content* słowo redis:



Po zatwierdzeniu wyboru, mamy dokładną dokumentacją dla wspomnianego obrazu, a nawet polecenie, za pomocą którego możemy pobrać dockerowy obraz (docker pull redis):

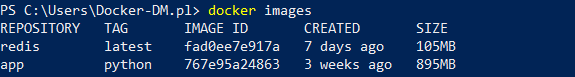


Przejdźmy więc zatem do Naszego terminala (ja akurat pracuję na Windows PowerShell-u) i wpiszmy wyżej wskazane polecenie:



*Pobranie obrazu redis*

Po pobraniu obrazu, możemy sprawdzić listę wszystkich dostępnych, wykonując polecenie docker images.



*Lista wszystkich lokalnie dostępnych obrazów*

Na powyższym screenshot-cie, możesz zauważyć dostępność dwóch obrazów - redis (tego, który pobraliśmy przed chwilą) oraz app (obraz Pythonowej aplikacji, którą stworzyłem jakiś czas temu). Z każdym obrazem mamy powiązane następujące informacje:

**REPOSITORY** - czyli nazwa repozytorium, którego dany obraz dotyczy

**TAG** - informacje o danej wersji obrazu

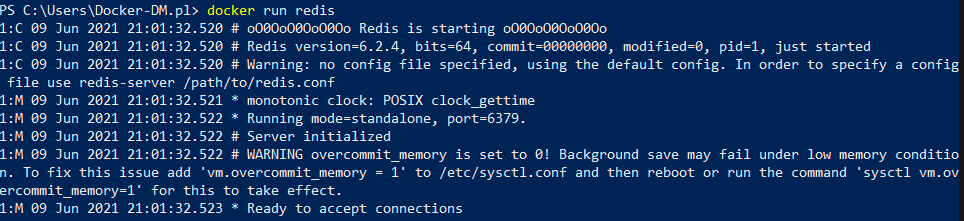
**IMAGE ID** - unikalny hash ID przypisany do obrazu

**CREATED** - data utworzenia obrazu

**SIZE** - rozmiar, jaki zajmują wszystkie warstwy obrazu

**Uruchomienie kontenera**

Następnym krokiem będzie zainstancjonowanie kontenera z pobranego obrazu redis. W tym celu wykorzystamy polecenie docker run redis:

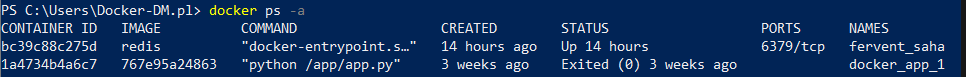


Zgodnie z ostatnim komunikatem: *Ready to accept connections*, widzimy, że uruchomienie kontenera przebiegło prawidłowo. Aby sprawdzić, czy faktycznie znajduje się on na liście działających kontenerów, uruchomimy nowy terminal i wpiszemy polecenie docker ps:



Kontener znajduje się na liście, a więc został prawidłowo uruchomiony i faktycznie działa w tle. Zauważ, że możemy wyłuskać z niego takie informacje jak: **CONTAINER ID** (unikalny identyfikator kontenera), **IMAGE** (z jakiego obrazu został zainstancjonowany kontener), **COMMAND** (), **CREATED** (kiedy został uruchomiony kontener), **STATUS** (jaki jest jego status działania), **PORTS** (na jakim porcie działa uruchomiony kontener), **NAMES** (nazwa kontenera; jeżeli nie zdefiniujemy jego nazwy przy uruchomieniu, to Docker Daemon sam zadba o to, aby przydzielić jakąś losową; przykładowe nazwy kontenerów możesz znaleźć tutaj: <https://frightanic.com/goodies_content/docker-names.php>).

Należy również zastanowić się, jak wyświetlić wszystkie kontenery, nie tylko te uruchomione w danym momencie. Należy wówczas do polecenia docker ps, dodać flagę -a:

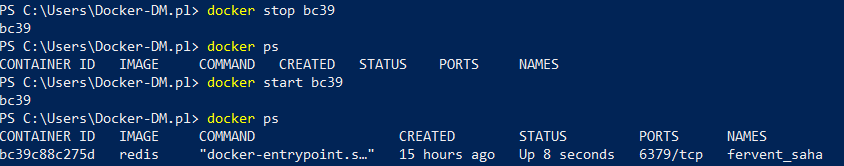


Jak widzisz, teraz wyświetlone zostały dwa kontenery, z czego jeden z nich jest uruchomiony (kontener powstały z obrazu redis), a drugi wyłączony (kontener dla aplikacji Pythonowej).

**Restart kontenera**

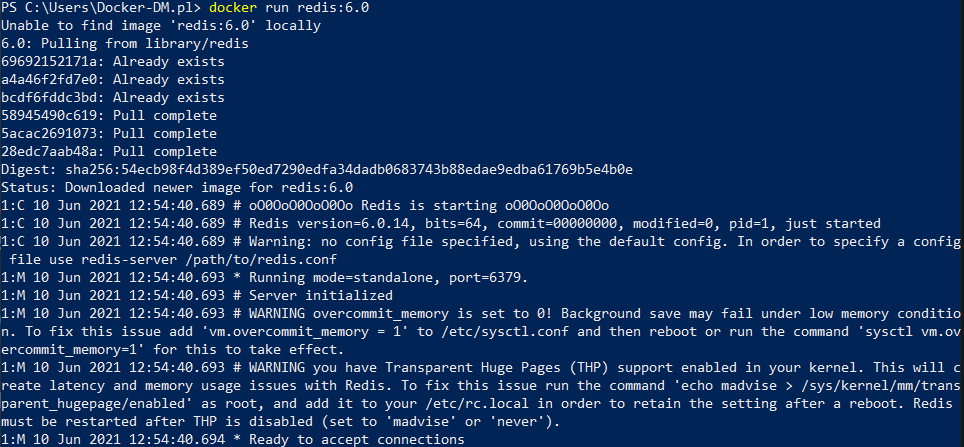
Załóżmy, że chcielibyśmy teraz zrestartować działający kontener. Powodów restartu może być wiele - crash aplikacji, nieprawidłowo uruchomiony kontener itp. W tym celu wykorzystamy dwa polecenia: docker stop <*id\_kontenera*> oraz docker start <*id\_kontenera*>.

Zacznijmy zatem od zatrzymania działającego kontenera (id możemy odczytać za pomocą polecenia docker ps -a; zwróć również uwagę na fakt, że nie musimy podawać całego ID, wystarczą jego kilka pierwszych znaków). Po zatrzymaniu kontenera, uruchomimy go na nowo poleceniem docker start <*id*> - tak, jak jest to widoczne na poniższym screenie.

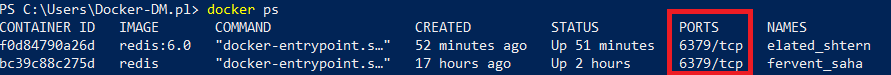


**Zmiana wersji redisa**

Załóżmy, że chcemy teraz uruchomić kolejny kontener z nowo pobranego obrazu - innej wersji redisa, np. 6.0 (do tej pory wykorzystywaliśmy najnowszą wersję, czyli 6.2.4). Ten kontener uruchomimy w troszkę inny sposób niż ostatnio - otóż chciałbym Ci zaprezentować, jak w łatwy sposób pobrać i zainstancjonować dany obraz - bez rozdrabniania się na komendę docker pull <*nazwa\_obrazu*>, a następnie docker run <*nazwa\_obrazu*>. Okazuje się bowiem, że wystarczy tak naprawdę wykonać polecenie docker run <*nazwa\_obrazu*>. Mimo że Nasz obraz nie będzie dostępny lokalnie, to przy próbie jego uruchomienia, zostanie automatycznie pobrany z Docker Hub.

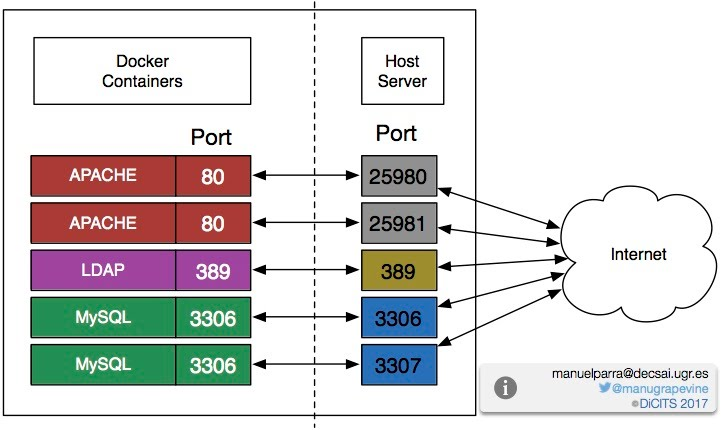


W tym momencie mamy uruchomione dwa kontenery (redis:6.0 oraz redis):



Zwróć jednak uwagę na zaznaczoną na czerwono sekcję PORTS. Jak można zauważyć, oba kontenery nasłuchują domyślnie na tym samym porcie - 6379. Gdybyśmy chcieli w tym momencie skorzystać, z któregoś z nich, napotkamy błąd i niemożliwość skorzystania z usługi. A to dlatego właśnie, że request nie jest w stanie dotrzeć do konkretnego kontenera przez wielokrotne wykorzystanie tego samego portu.

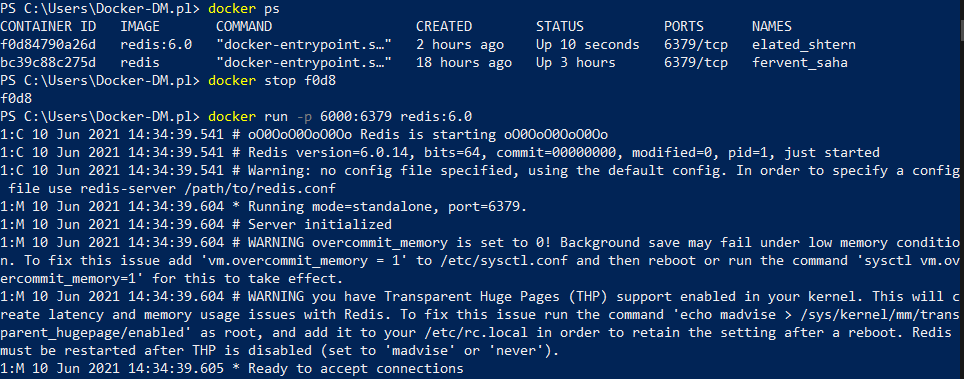
Funkcjonowanie dwóch kontenerów na tych samych portach aplikacji nie jest problematyczne, pod warunkiem że **porty hosta** dla tych dwóch aplikacji są niezależne. Dobrze odzwierciedla to poniższy obraz:



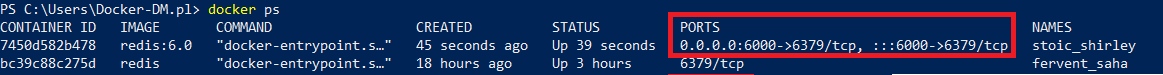
Zwróć uwagę, że każdy z portów Hosta jest niezależny i wartość każdego różni się od pozostałych. Dzięki temu **porty kontenerów** mogą się powtarzać (2 x 80, 389 i 2 x 3306).

Wracając zatem do Naszego przykładu z redisem, aby Nasze aplikacje poprawnie działały, wystarczy któryś z kontenerów uruchomić na innym porcie Hosta (np. 6000) i przekierować go do odpowiedniego portu aplikacji (6379).

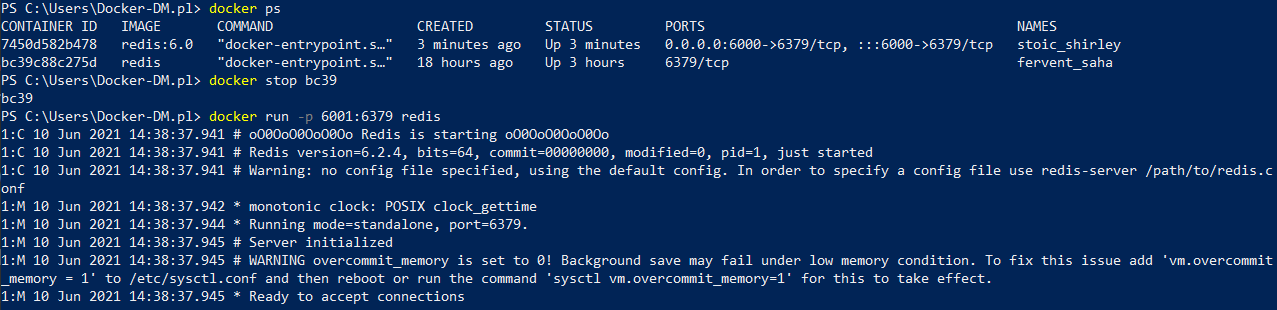
Wyłączmy zatem kontener redis:6.0 i zainstancjonujmy go raz jeszcze, tym razem z parameterem **-p 6000:6379**, który określa zbindowany port hosta - 6000 do portu kontenera - 6379.



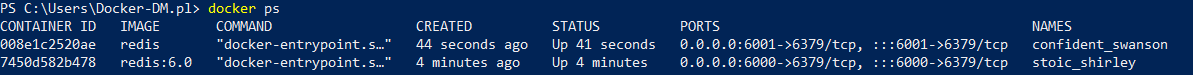
I teraz, uruchamiając w nowym terminalu polecenie docker ps, ujrzymy:



Pozostało teraz wykonanie tego samego z kontenerem redis (port hosta ustawimy na 6000, a kontenera pozostawimy bez zmian):



Po wylistowaniu:



**UWAGA:**

Choć nazwy tego nie odzwierciedlają, to różnica między poleceniem docker run a docker start jest znacząca. Pierwsze polecenie odpowiedzialne jest za tworzenie zupełnie nowego kontenera z gotowego obrazu, a drugie za uruchamianie już istniejącego kontenera.

**Praca z uruchomionym kontenerem**

Do tego momentu przeprowadzaliśmy dość “suche” operacje związane jedynie z zarządzaniem obrazami, uruchamianiem, zatrzymywaniem kontenerów. Na pewno chciałbyś w jakiś sposób wykorzystać uruchomiony kontener. Zaczniemy od dość prostej operacji - przeanalizujemy wirtualny system plików uruchomiony w kontenerze. Przypominam, że kontener to logicznie wydzielone środowisko z wydzieloną strukturą plikową, ustawionymi zmiennymi środowiskowymi i konfiguracją etc. Zatem możliwe jest przeanalizowanie takiego systemu plików.

Aby uruchomić terminal znajdujący się w izolowanym systemie plików, wpiszemy:   
docker exec -it <id\_kontenera> /bin/bash



Jak widzisz, powyższym poleceniem udało Nam się uzyskać kontrolę nad wirtualnym systemem plików znajdującym się wewnątrz kontenera. Jest to Unixowy filesystem, o czy możemy się przekonać, wykonując kilka podstawowych poleceń.



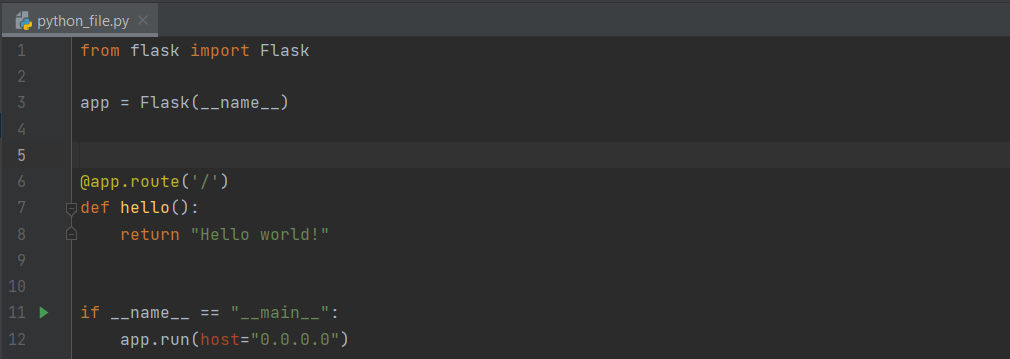
**UWAGA:**

Aby wyłączyć tak uruchomiony terminal wewnątrz wirtualnego systemu pliku, wpisz polecenie exit

**Development aplikacji Python + Flask + Docker**

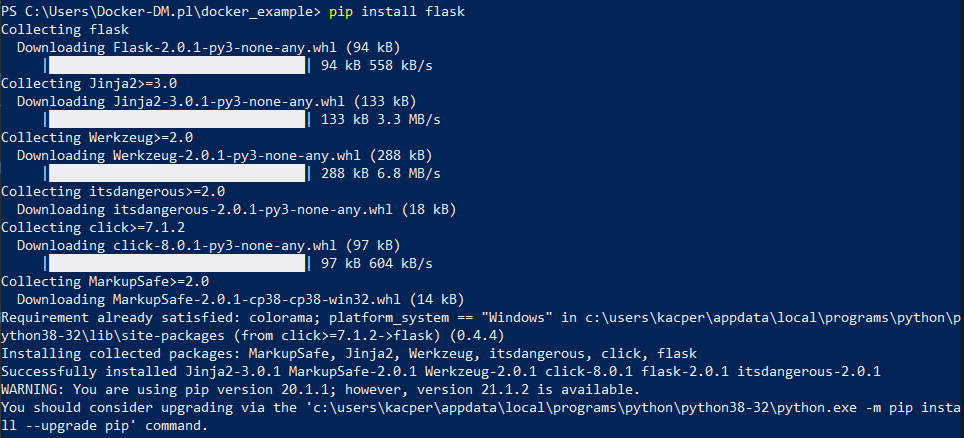
Poznaliśmy już podstawy związane z obsługą Dockera, teraz nastąpiła chwila, w której możemy sprawnie przejść przez development pierwszej aplikacji. Taka apka będzie uruchamiana wewnątrz Dockerowego kontenera z odpowiednimi bibliotekami - wykorzystamy Flask-a, aby umożliwić jej działanie jako web serwis.

Zajmijmy się w pierwszej kolejności zbudowaniem odpowiedniego kodu, który zostanie uruchomiony na localhost na porcie 5000 i wyświetlać będzie prosty komunikat “Hello world!”.



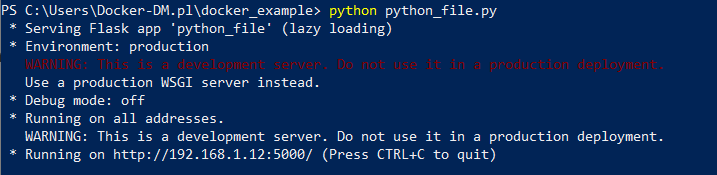
*Plik python\_file.py*

Pamiętaj również, aby zainstalować w swoim wirtualnym środowisku moduł flask (ja dokonałem tego przez polecenie pip):

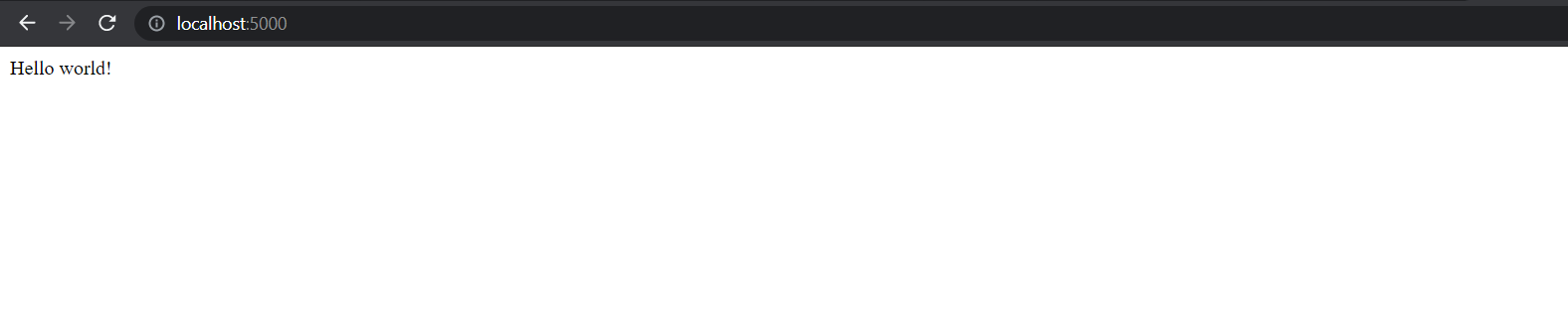


**Uruchomienie lokalne, nie w kontenerze**

Po stworzeniu pliku takiego jak wyżej, możemy uruchomić napisaną aplikację.



Po przejściu na adres localhost:5000 w Naszej przeglądarce, otrzymamy:



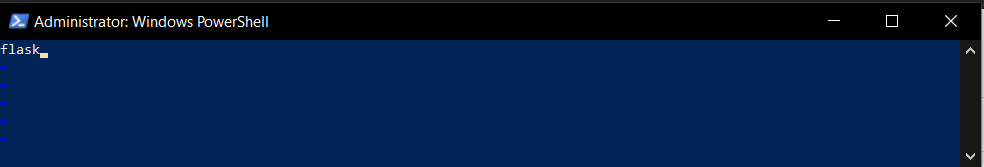
Aplikacja działa, teraz możemy przejść do najważniejszej części tej sekcji. Chcemy taką funkcjonalność uruchomić w Dockerowym kontenerze. Zgodnie ze schematami, które już widziałeś w szkoleniu, będziemy musieli sporządzić odpowiedni Dockerfile. Na podstawie niego, możliwe będzie zbudowanie obrazu i uruchomienie instancji kontenera.

Zanim jednak przystąpimy do tworzenia Dockerfile-a, dodajmy do naszego katalogu jeszcze jeden plik - requirements.txt. Umieścimy w nim nazwy wszystkich bibliotek, z których korzystamy w ramach aplikacji python\_file.py.

Wiedz, że w Dockerfile-u, będziemy musieli sprecyzować, z jakich zewnętrznych bibliotek korzysta Nasza aplikacja. I właśnie do tego celu wykorzystamy requirements.txt - Docker będzie w stanie odczytać z pliku nazwy wszystkich bibliotek i je automatycznie zainstalować, wykorzystując polecenie pip. A to wszystko po to, aby utworzyć prawidłowy system plików i dependencji w ramach kontenera.



*Utworzenie nowego pliku tekstowego*

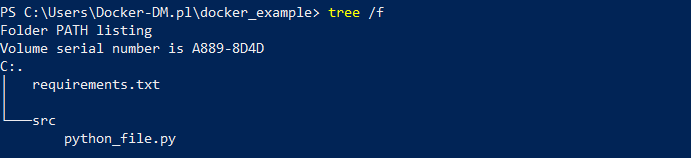


*Wypełnienie pliku odpowiednią zawartością*

**UWAGA:** Abyś mógł korzystać z edytora vim/vi w swoim terminalu, konieczna jest instalacja: https://www.vim.org/download.php

Jak widzisz, w pliku requirements.txt wystarczyło umieścić tylko nazwę biblioteki flask.

Na koniec, w celu efektywniejszego wydzielenia struktury katalogu, w miejscu, w którym znajdują się teraz pliki python\_file.py oraz requirements.txt, utworzyłem folder src i przeniosłem do niego pierwszy z tych dwóch plików. W rezultacie powinniśmy operować na poniższej hierarchii plików:



Przejdźmy teraz do skonstruowania odpowiedniego Dockerfile-a.

**Utworzenie Dockerfile**

Taki plik Dockerowy utworzymy bezpośrednio w katalogu docker\_example, a jego zawartość zakodujemy w następujący sposób:

| # pobranie obrazu Pythona z DockerHub FROM python:latest # wyznaczenie ścieżki roboczej w kontenerze WORKDIR /app # skopiowanie zawartości pliku requirements.txt do working directory COPY requirements.txt . # instalacja dependencji RUN pip install -r requirements.txt # skopiowanie zawartości katalogu src/ do working directory COPY src/ . # komendy do uruchomienia aplikacji CMD ["python", "./python\_file.py"] |
| --- |

**Dobra praktyka**

Myślę, że dodanie komentarzy do wyżej przedstawionego pliku wystarczająco opisuje, za co odpowiada konkretna linia kodu. Chciałbym jedynie zatrzymać się przy poleceniu COPY.

Bądź świadomy tego, iż zamiast pisać dwóch poleceń:

1. **COPY requirements.txt .**
2. **COPY src/ .**

moglibyśmy w zasadzie napisać tak:

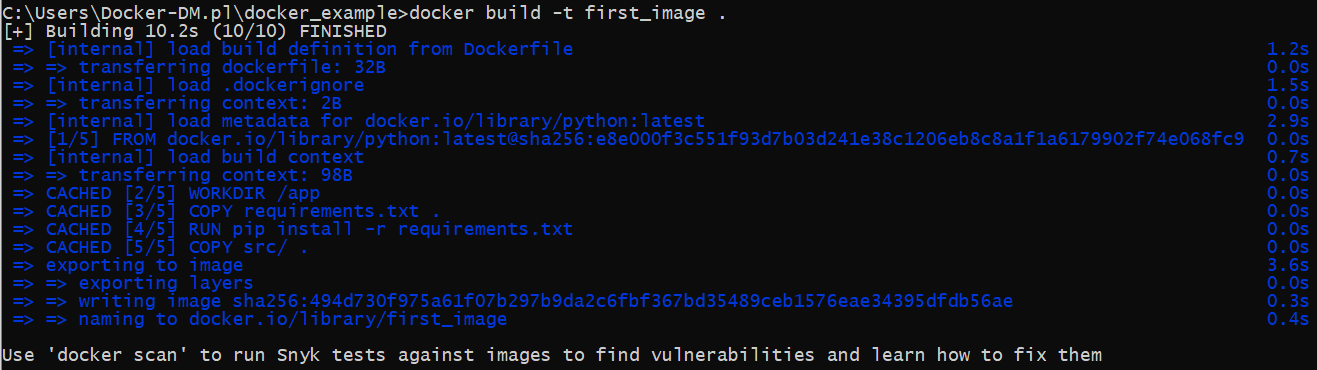
**COPY . .**

Co oznaczałoby przekopiowanie całego katalogu aplikacji do working directory kontenera.

Dlaczego zatem nie ograniczyliśmy tego do tak pojedynczego zapisu? Odpowiedź jest prosta - w grę wchodzi kwestia optymalizacji. Otóż zapis w Dockerfile-u dwóch niezależnych poleceń kopiujących - najpierw pliku z wymaganiami, a następnie katalogu src/, powoduje utworzenie dwóch niezależnych warstw wewnątrz Dockerowego obrazu. Dzięki takiemu rozwarstwieniu, obie części obrazu będą mogły być budowane niezależnie. To znaczy - jeżeli wprowadzimy pojedynczą zmianę wewnątrz requirements.txt, to rebuild będzie dotyczył warstwy, w której się tylko ten plik znajduje. Tak samo w przypadku src/ - pojedyncza modyfikacja naniesie konieczność przebudowy tylko tej jednej konkretnej warstwy (w której katalog się znajduje). Gdybyśmy więc postawili na wykonanie pojedynczej operacji kopiującej, zarówno requirements.txt jak i src/ zostałyby umieszczone w jednej warstwie obrazu, a każda nawet najmniejsza zmiana wewnątrz, np. pliku z wymaganymi dependencjami, wiązałaby się z przebudową całej niemało ważącej warstwy.

**Utworzenie obrazu**

Mając utworzony Dockerfile, możemy zbudować z niego obraz. Do tego celu, użyjemy polecenia docker build -t <nazwa\_obrazu> .

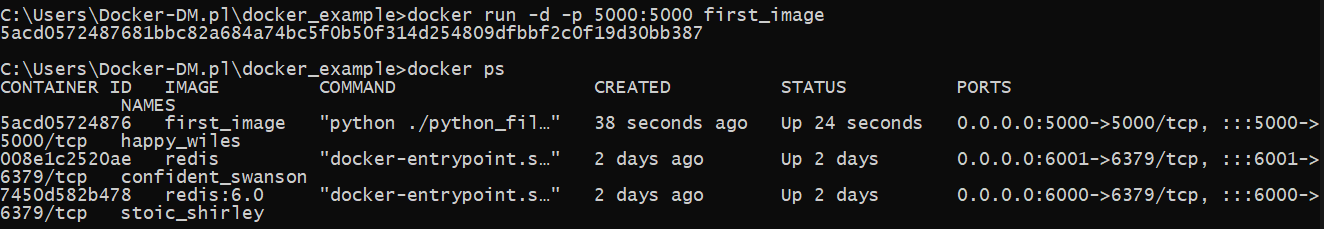


Po sprawdzeniu listy dostępnych obrazów, powinniśmy m.in ujrzeć utworzony obraz first\_image:

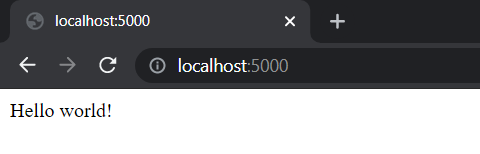


**Uruchomienie kontenera**

Obraz mamy już utworzony, teraz nadeszła pora na uruchomienie z niego kontenera. Użyjemy do tego polecenia docker run z flagami -p oraz -d. O ile samo w sobie polecenie powinno być Ci znajome, to zasady działania parametru -d do tej pory jeszcze nie omówiliśmy. Otóż jest to wprowadzenie, tzw. trybu “detached head”, który najprościej - zwalnia nas z konieczności uruchomienia drugiego terminala po zainstancjonowaniu kontenera (uruchomiona aplikacja nie blokuje terminala, przez który została uruchomiona).

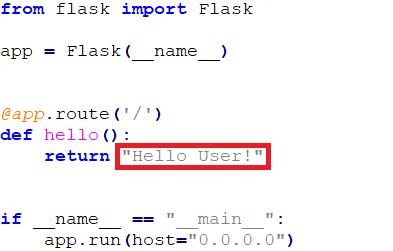


Kontener z aplikacją wyświetlającą “Hello World!” na porcie 5000 został uruchomiony! Możemy się o tym przekonać, przechodząc w przeglądarce na adres URL localhost:5000.



**Wprowadzanie zmian w aplikacji**

Załóżmy, że zmieniliśmy nieco koncept działania Naszej aplikacji i chcielibyśmy, aby zamiast napisu Hello world! wyświetlany był Hello User!. Musimy więc przejść do pliku src/docker\_example i zmodyfikować funkcję hello().



Pytanie teraz, jakie należy sobie postawić, to co zrobić, aby zmiana była również zaaplikowana wewnątrz kontenera, w której działa aplikacja? Otóż rozwiązanie jest jedno - musimy przebudować cały obraz, a możliwe jest to wywołując raz jeszcze polecenie docker build, następnie docker run. Uprzednio oczywiście pamiętając o usunięciu kontenera, w którym znajduje się stara aplikacja oraz obrazu, z którego ten kontener powstał.

1. Zatrzymanie kontenera ze starą wersją aplikacji:



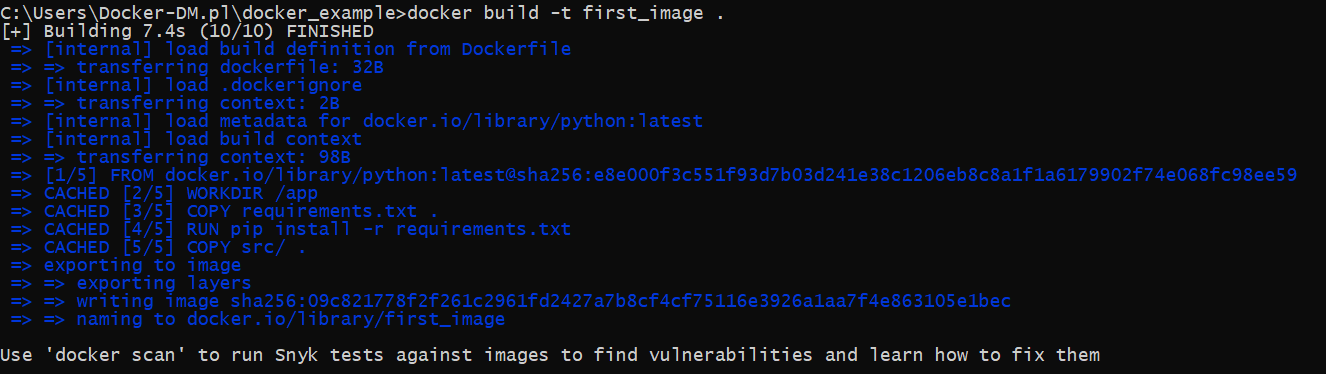
1. Usunięcie zatrzymanego kontenera:



1. Usunięcie starej wersji obrazu:



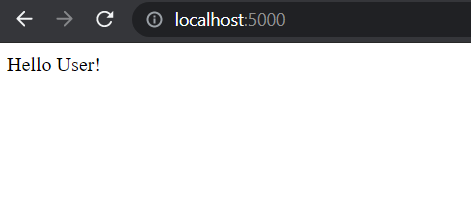
1. Zbudowanie obrazu na podstawie nowej wersji kodu:



1. Uruchomienie nowo powstałego obrazu:



Po wykonaniu wszystkich wyżej operacji, otrzymamy nową instancję kontenera z aktualną wersją programu. Możemy sprawdzić, czy zmiany zostały zaaplikowane, uruchamiając powstałą aplikację w przeglądarce: localhost:5000.



Jak widzisz, aplikacja nie wyświetla już napisu Hello world, tylko Hello User!. To znaczy, że udało nam się poprawnie przebudować obraz i uruchomić najnowszy kontener.

**Czy można łatwiej?**

Przyznaj jednak, że wykonanie tak długiego ciągu powyższych poleceń za każdym razem, gdy chcemy wprowadzić nawet najdrobniejszą zmianę w programie, może być dość uciążliwe. Jest wiele sposobów na automatyzację takiej przebudowy, np. poprzez napisanie skryptu automatycznie wykonującego wszystkie polecenia. Takie rozwiązanie jednak nie pozwala wykorzystać pełnego potencjału Dockera.

Jeżeli chcemy, aby nasz kontener był aktualizowany “on live”, powinniśmy skorzystać z Docker Compose. Jest to kolejne, rodzime dla Dockera, narzędzie umożliwiające właśnie usprawnianie naszej pracy, ale również efektywne tworzenie aplikacji multikontenerowych. Przy rozbudowanych mikroserwisach rzadko kiedy przecież będziemy wykorzystywali tylko jeden kontener aplikacji. W celu łatwego skalowania systemów, często stosuje się rozwiązanie, w którym oddzielne logiki biznesowe aplikacji stawia się w różnych kontenerach. Aby natomiast umożliwić im komunikację i współpracę między sobą, warto wdrożyć wcześniej wspomniane rozwiązanie docker compose. Nad tym tematem pochylimy się jednak w dalszych szkoleniach…