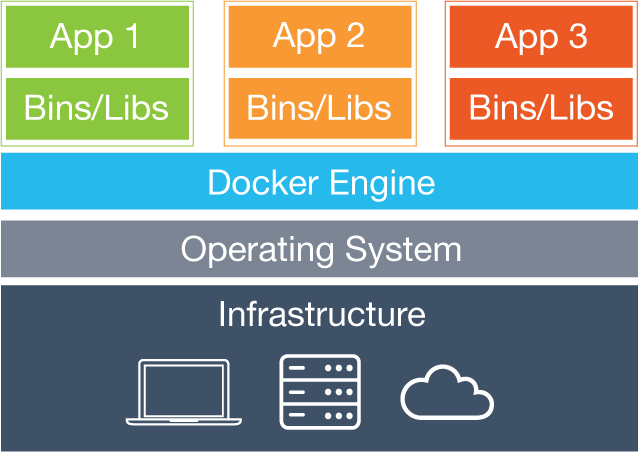
Rozproszone systemy operacyjne

Projekt

Grupa G

**Opis rozwiązania Docker**

Docker to zestaw narzędzi umożliwiających pracę z kontenerami w systemie operacyjnym Linux udostępniany na zasadach Open-Source. Kontenery są dostępne w jądrze Linuxa już od pewnego czasu, ale dopiero za sprawą Dockera, zaczęły być popularne oraz wykorzystywane przez programistów, w szczególności do wygodnego udostępniania aplikacji, wraz z jej wszystkimi zależnościami (wszelkimi bibliotekami i narzędziami z których korzysta). Można za jego pomocą tworzyć lekkie środowiska wirtualne, które nie posiadają części odpowiedzialnej za wirtualizację sprzętu.



Docker składa się z narzędzi:

* Docker Engine – służy do tworzenia, zarządzania i uruchamiania kontenerów.
* Docker Hub – służy do dzielenia się obrazami kontenerów między ich użytkownikami (obrazy mogą być dostępne zarówno publicznie jak i tylko dla ograniczonej grupy odbiorów)

Za pomocą narzędzia Docker Engine możemy przygotować obrazy kontenerów, które następnie będą uruchamiane na konkretnych kontenerach. Obraz przygotowuje się wykorzystując specjalny plik Dockerfile:

FROM python:2.7

ADD . /code

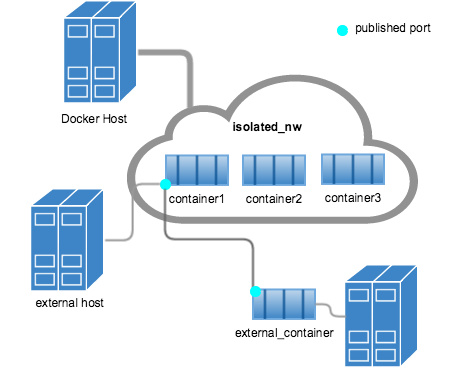
WORKDIR /code

RUN pip install -r requirements.txt

CMD python app.py

W pliku tym określamy na podstawie jakiego obrazu budujemy nasz obraz oraz mamy do dyspozycji różne instrukcje, dodające do naszego obrazu jakieś pliki, bądź wywołujące polecenia. Każda instrukcja stanowi tak zwaną warstwę, po wywołaniu której Docker Engine zapamiętuje jakie zmiany zaszły w obrazie. Dzięki temu po dokonaniu zmiany w naszym obrazie, do naszych współpracowników bądź na platformę DockerHub musimy wysłać tylko paczkę zawierającą tą zmianę, co znacznie przyśpiesza pracę z obrazami kontenerów.

Przygotowane obrazy można następnie uruchamiać na kontenerach. Kontenery działają w odseparowanym środowisku i można się z nimi komunikować na przykład wykorzystując Sockety. Domyślnie wszystkie uruchomione kontenery znajdują się w jednej wirtualnej sieci, ale mamy możliwość zmapowania portu konkretnego kontenera, z portem fizycznej maszyny na której kontenery są uruchamiane:

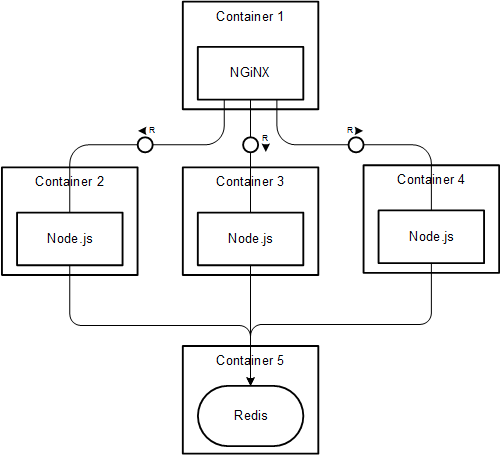


**Narzedzie Docker Compose**

Podczas poznawania narzędzia docker natrafiliśmy na program Docker Compose. Służy on do wygodnego skonfigurowania środowiska składającego się z wielu współpracujących ze sobą kontenerów. Za pomocą jednego pliku konfiguracyjnego docker-compose.yml można zestawić wiele komunikujących się ze sobą kontenerów, ustalić topologię wirtualnych sieci a następnie nadzorować ich wspólną pracę. Daje to niezwykłą wygodę, pozwalając zawrzeć w jednym miejscu konfigurację architektury projektu oraz podzielić się nią ze współpracownikami, wymieniając jeden pliki konfiguracyjny oraz udostępniając odpowiednie obrazy kontenerów.

**Raport z uruchomienia dostępnych w sieci wybranych konfiguracji demonstrujących wykorzystanie Docker**

Poznając narzędzie Docker oraz Docker Compose postanowiliśmy uruchomić i przetestować prostą aplikację o strukturze zbliżonej do aplikacji jaką docelowo chcemy wykonać w ramach projektu. Postanowiliśmy skonfigurować i uruchomić aplikację wykorzystując kilka popularnych konfiguracji demonstracyjnych.



Wszystkie zapytania kierowane są najpierw do Load Balancera (kontener 1), który następnie rozdziela je pomiędzy trzy kontenery odpowiedzialne za ich obsługę. Load Balancer został zrealizowany za pomocą oficjalnego obrazu serwera http nginx. Do serwera został przekazany następujący plik konfiguracyjny:

http {

upstream myapp1 {

server srv1.example.com;

server srv2.example.com;

server srv3.example.com;

}

server {

listen 80;

location / {

proxy\_pass http://myapp1;

}

}

}

Takie ustawienie sprawia że zachowuje się on jak load balancer przekazując zapytania do jednego z trzech dostępnych serwerów. Serwery wybierane są przy wykorzystaniu domyślnego algorytmu round-robin. By zbudować obraz serwera nginx wygenerowaliśmy następujący plik konfiguracyjny Dockerfile:

FROM nginx

MAINTAINER Przemysław Łada

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

W pliku tym zawarta jest informacja o obrazie na jakim będzie bazował nasz obraz (oficjalny obraz serwera nginx z DockerHub) oraz polecenie dodające nasz plik konfiguracyjny nginx.conf

Obraz ten został następnie zbudowany za pomocą następującego polecenia:

docker build -t przlada/nginx .

Zapytania są poprzez Load Balancer przekazywane do kontenerów na których uruchomiony jest serwer WWW z działająca bardzo prostą aplikacją w języku Python wykorzystującej framework Flask. Aplikacja obsługuje zapytanie protokołu http oraz odpowiada na nie wysyłając stronę html z identyfikatorem kontenera, oraz wartością zwiększoną wartością licznika odwiedzin, który przechowywany jest w bazie danych. Poniżej znajduje się najważniejsza część kodu tej aplikacji:

redis = Redis(host='redis', port=6379)

@app.route('/')

def hello():

redis.incr('hits')

return 'Hello World on'+os.getenv('HOSTNAME', "Brak")+'! I have been seen %s times.' % redis.get('hits')

Plik app.py zawierający powyższy program oraz plik requirements.txt wykorzystywane są przez poniższy plik Dockerfile odpowiedzialny za obraz kontenerów obsługujących serwery www.

FROM python:2.7

ADD . /code

WORKDIR /code

RUN pip install -r requirements.txt

CMD python app.py

Do budowy obrazu aplikacji serwerowych wykorzystywany jest oficjalny obraz „python” z portalu DockerHub. Dodawane są do niego nasze pliki, wykonywana jest instalacji wymaganych bibliotek oraz uruchamiany jest serwer www frameworka Flask z naszą aplikacją z pliku app.py

Kontener 5 wykorzystuje oficjalny obraz serwera bazy danych redis i nie są w nim wprowadzane żadne dodatkowe konfiguracje, dlatego nie potrzebne było tworzenie dla niego pliku konfiguracyjnego Dockerfile.

By wygodnie uruchomić jednocześnie wszystkie kontenery wykorzystaliśmy narzędzie DockerCompose. Zaczęliśmy od stworzenia następującego pliku konfiguracyjnego docker-compose.yml:

nginx:

build: ./nginx

ports:

- "80:80"

node1:

build: ./node

volumes:

- .:/code

depends\_on:

- redis

node2:

build: ./node

volumes:

- .:/code

depends\_on:

- redis

node3:

build: ./node

volumes:

- .:/code

depends\_on:

- redis

redis:

image: redis

Konfigurowany jest w nim po kolei każdy kontener. Konfiguracja kontenera zawiera informację gdzie może być znaleziony plik Dockerfile potrzebny do zbudowania potrzebnego obrazu, jakie porty mają być zmapowane z portami fizycznego urządzenia na którym uruchamiane są kontenery. Przy kontenerach odpowiedzialnych za serwery http (node1, node2, node3) wprowadzony jest dodatkowy parametr depends\_on, wpływający na to że zarządca program Docker Compose najpierw uruchomi kontener obsługujący serwer bazy danych redis a dopiero w następnej kolejności zależne od niego kontenery node1, node2, node3.

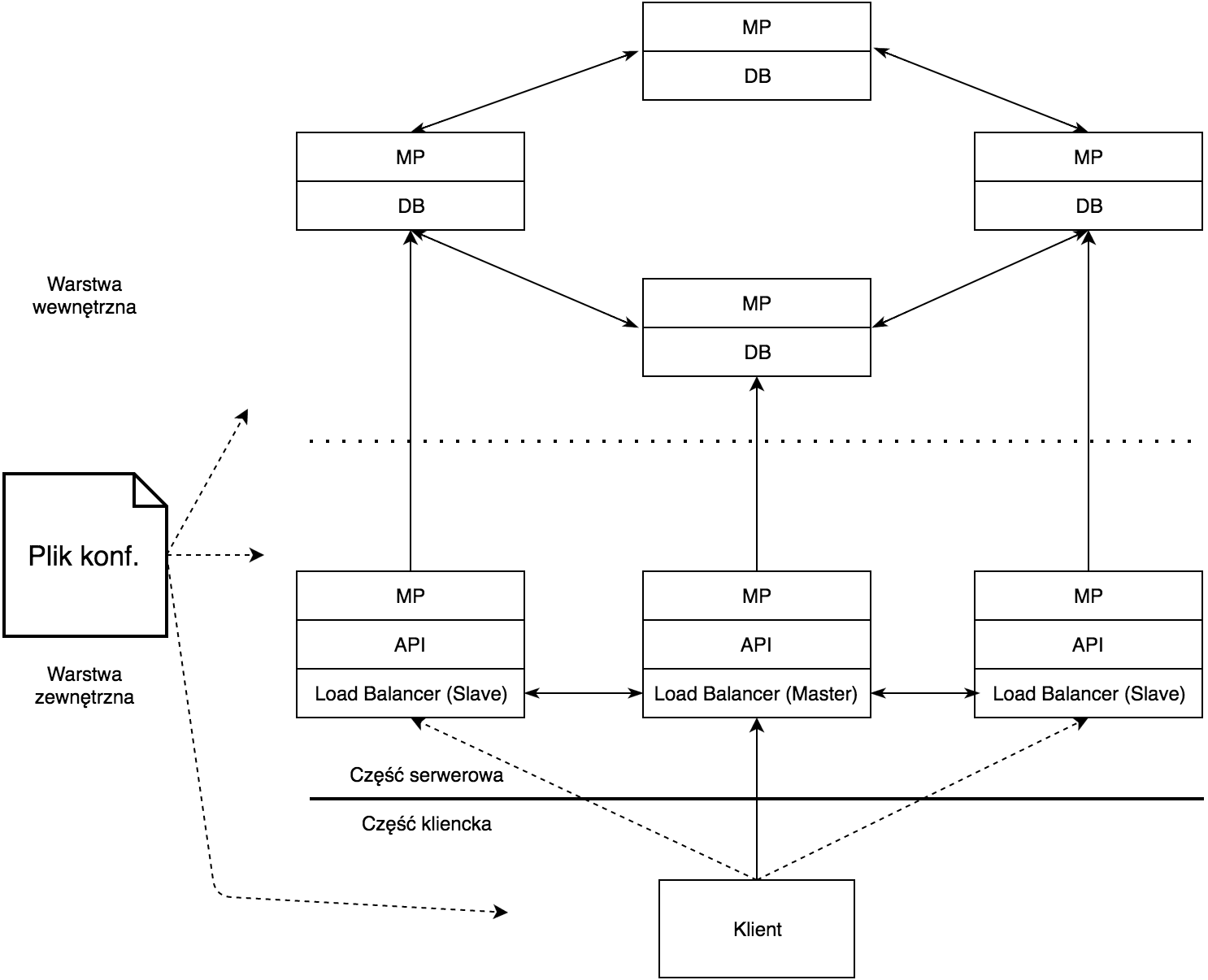
Po przygotowaniu środowiska możemy je uruchomić w trybie interaktywnym, poleceniem:

docker-compose up

Gdy wszystkie kontenery zostaną uruchomione, możemy przetestować aplikację, wysyłając zapytanie http na port 80. W odpowiedzi dostaniem zawsze identyfikator następnego węzła z pośród trzech odpowiedzialnych za obsługę zapytań http oraz kolejny numer licznika odwiedzin. Pokazuje to prawidłowe działanie komunikujących się ze sobą kontenerów oraz równoważenie obciążenia poprzez load balancer. Widać również że kontenery korzystają z jednej tej samej bazy danych, ponieważ licznik odwiedzin jest odpowiednio zwiększany po każdym zapytaniu.

**Koncepcja i architektura rozwiązania**

Nasza grupa zdecydowała się na stworzenie aplikacji pozwalającej jej użytkownikom wymieniać się plikami graficznymi



Rysunek - Architektura systemu

Główne elementy aplikacji:

* Część kliencka:
  + Aplikacja kliencka
* Część serwerowa:
  + Warstwa zewnętrzna:
    - Moduł Load Balancer
    - Moduł obsługi zapytań do interfejsu programistycznego (API)
  + Warstwa wewnętrzna:
    - Moduł rozproszonej bazy danych (DB)
  + Elementy wspólne dla warstwy wewnętrznej i zewnętrznej:
    - Moduł nadzorcy węzłów (MP)
* Elementy wspólne dla części serwerowej i klienckiej:
  + Moduł interpretujący wspólny plik konfiguracyjny

**Wymagania funkcjonalne:**

**Wymagania niefunkcjonalne:**

**Harmonogram realizacji projektu**

**Organizacja środowiska programistycznego**

**Zadania do wykonania:**

**Aplikacja kliencka**

**Obsługa zapytań od klientów**

**-Równoważenie obciążenia (Load Balancing)**

**-Buforowanie i nadzór nad wykonaniem zapytań**

**Moduł obsługi zapytań do interfejsu programistycznego (API)**

**Moduł rozproszonej bazy danych klucz-wartość (DB)**

**-Obsługa interfejsu programistycznego bazy danych**

**-Nadzorca spójności danych**

**Moduł nadzorcy węzłów (MP)**

**Moduł interpretujący wspólny plik konfiguracyjny**

**Mechanizm uwierzytelniania węzłów w części serwerowej**

**Zestaw testów**

**Skrypt prezentujący działanie aplikacji**

**Konfiguracja narzędzi Docker i Docker Compose**