Grupa G (pt 16:15-18:00):

Ivan Dziemianczyk
Karolina Drabarz
Mikołaj Cybulski
Michał Haponiuk
Przemysław Łada
Amadeusz Sadowski
Michał Chilczuk

Rozproszone systemy operacyjne

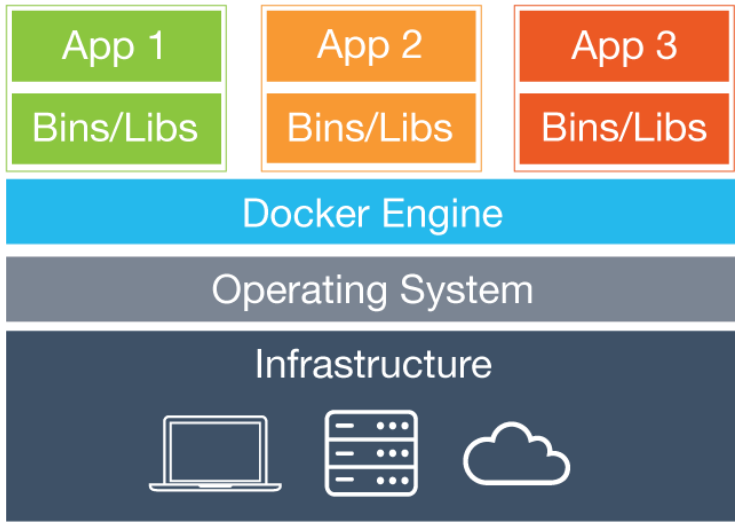
Etap I

Spis treści:

1. Docker

1.1. Opis rozwiązania Docker

Docker (https://www.docker.com/) jest zestawem narzędzi umożliwiających pracę z kontenerami w systemie operacyjnym Linux. Jest 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 zależnościami (bibliotekami i narzędziami, z których korzysta). Można za jego pomocą tworzyć lekkie środowiska wirtualne, które nie posiadają części odpowiedzialnej za wirtualizację sprzętu.



Rys. 1: Przekrój przez warstwy dla aplikacji korzystających z Dockera.

W skład Dockera wchodzą:

- 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 odbiorców).

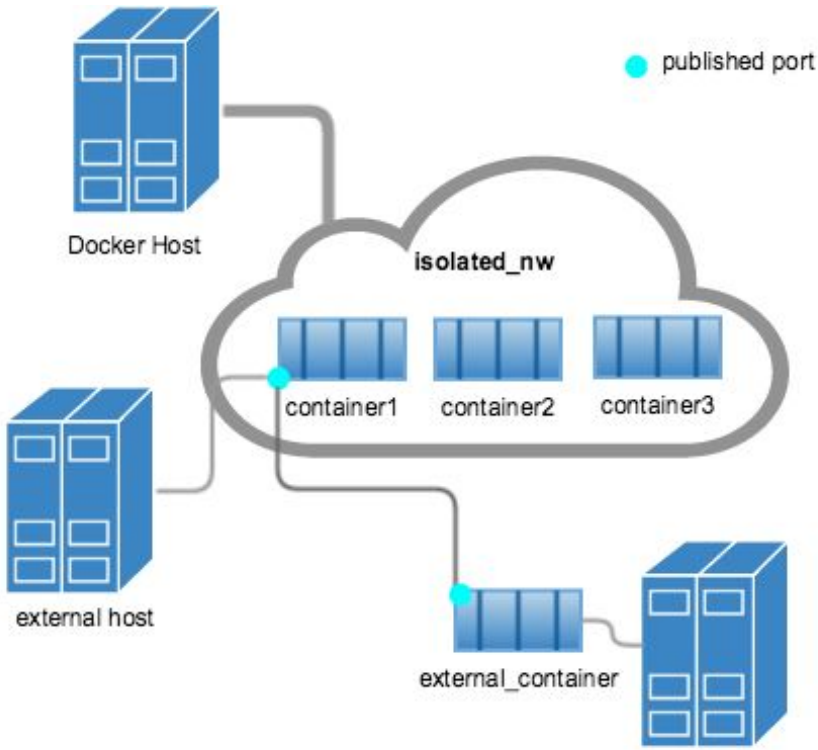
Za pomocą narzędzia Docker Engine możemy przygotować obrazy kontenerów, które następnie będą uruchamiane na konkretnych kontenerach. Obraz przygotowuje się wykorzystując specjalny plik *Dockerfile*. Zawartość takiego przykładowego pliku znajduje się poniżej:

```
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 bazowego obrazu budujemy nasz obraz oraz mamy do dyspozycji instrukcje, dodające wybrane 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 do naszych współpracowników bądź na platformę DockerHub musimy wysłać tylko informację o dokonanych zmianach, 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 gniazda. Domyślnie wszystkie uruchomione kontenery znajdują się w jednej wirtualnej sieci, ale mamy możliwość mapowania portu konkretnego kontenera z portem fizycznej maszyny na której kontenery są uruchamiane:



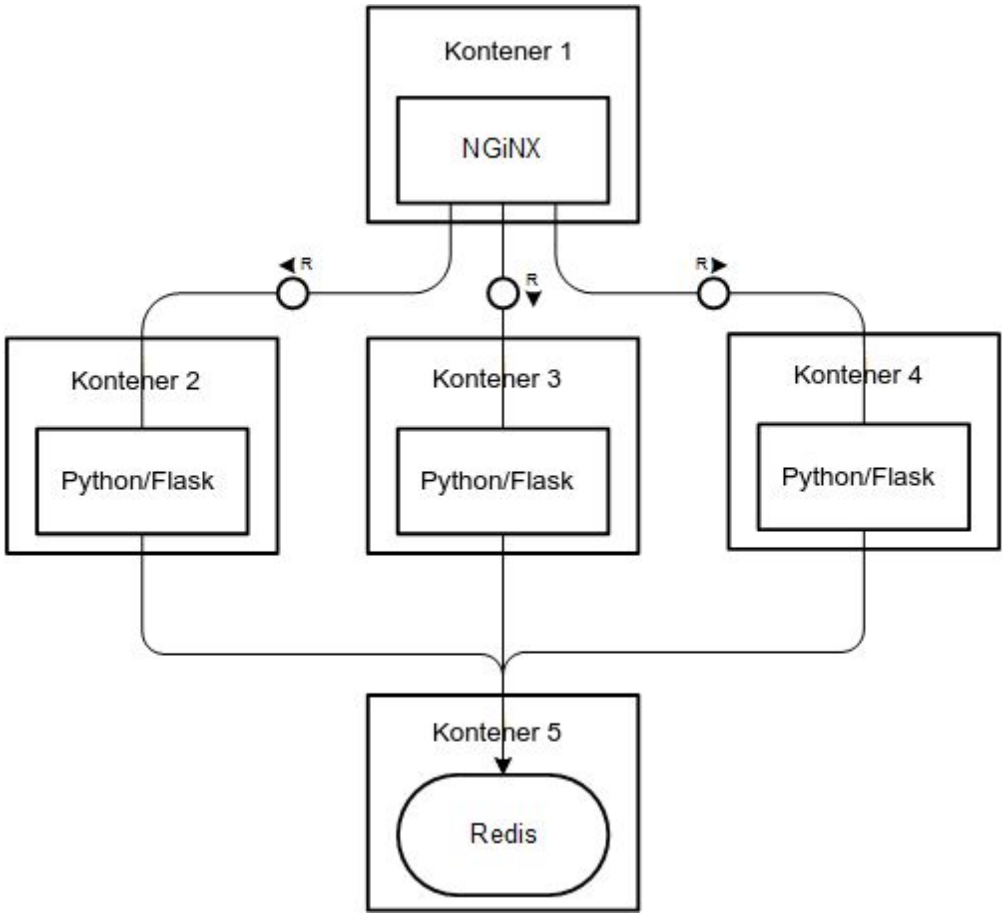
Rys. 2: Środowisko pracy kontenerów.

1.2. Narzędzie Docker Compose

Podczas poznawania narzędzia Docker odkryliśmy 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.

1.3. 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.



Rys. 3: Schemat komunikacji między kontenerami.

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*. Dla serwera został przygotowany następujący plik konfiguracyjny:

```
http {
    upstream myapp1 {
        server node1:5000;
        server node1:5000;
        server node1:5000;
    }

    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 przekazywane poprzez Load Balancer do kontenerów, na których uruchomiony jest serwer http z działającą bardzo prostą aplikacją w języku *Python* wykorzystującą framework *Flask*. Aplikacja obsługuje zapytanie protokołu HTTP oraz odpowiada na nie wysyłając odpowiedź z identyfikatorem kontenera, oraz 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 os.getenv('HOSTNAME', "Brak")+
          ' 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 instalacja wymaganych bibliotek oraz uruchamiany jest serwer HTTP 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 Docker Compose. 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 programu 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:

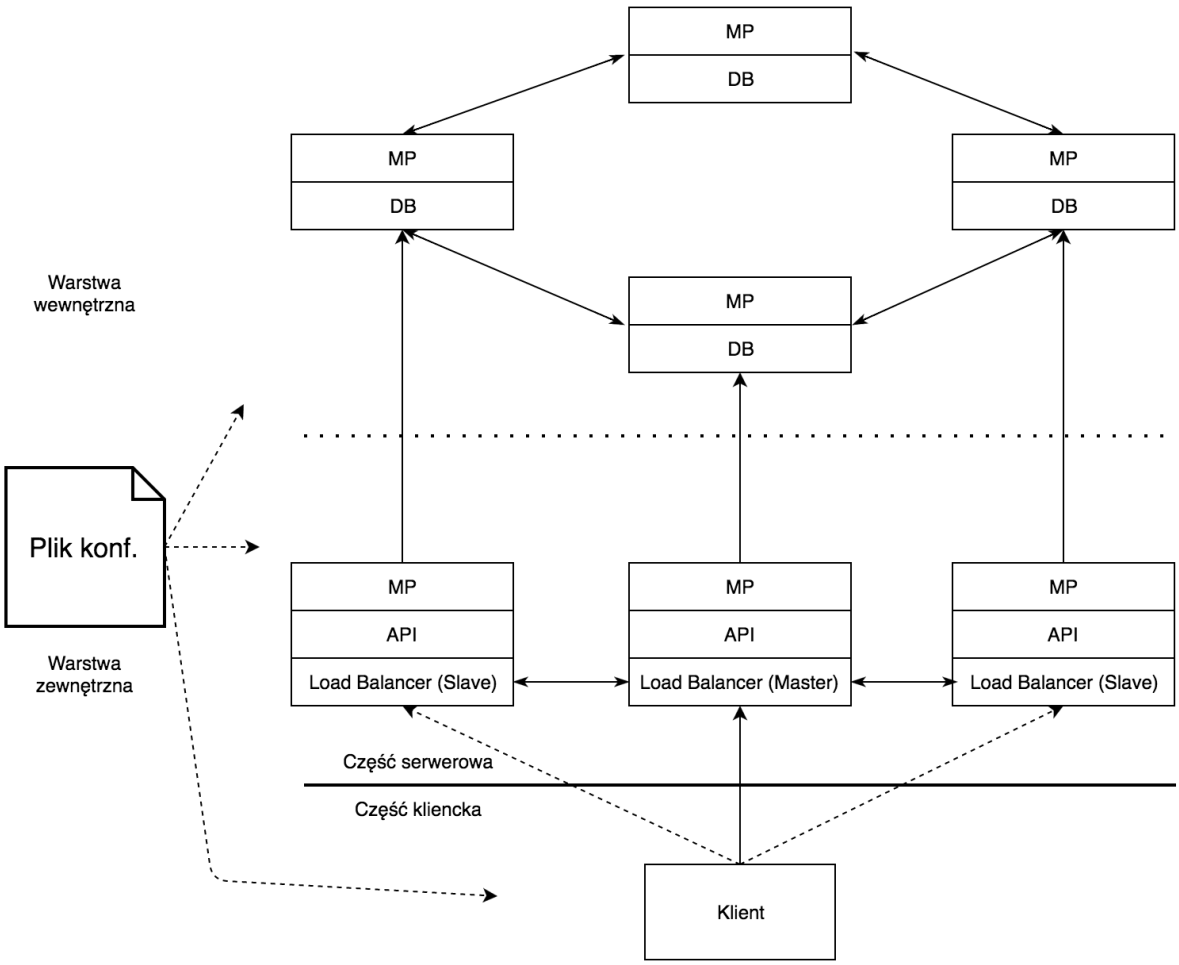
```
docker-compose up
```

Gdy wszystkie kontenery zostaną uruchomione, możemy przetestować aplikację, wysyłając zapytanie HTTP na port 80. W odpowiedzi dostaniemy zawsze identyfikator następnego węzła spośród 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.

2. Koncepcja i architektura rozwiązania

2.1 Ogólny schemat systemu

Nasza grupa zdecydowała się na stworzenie aplikacji pozwalającej użytkownikom wymieniać się plikami graficznymi. Wgrane przez użytkowników pliki są chronione i dostępne tylko upoważnionym do tego użytkownikom.



Rys. 4: 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.

W pierwszym etapie prac postanowiliśmy skupić się na opracowaniu i uruchomieniu warstwy zewnętrznej służącej za komunikację z oprogramowaniem klienckim. Moduły wchodzące w skład tej warstwy będą działać następująco:

- Load balancer - Każdy węzeł warstwy zewnętrznej, może przyjąć na siebie zadanie rozkładu ruchu przychodzącego. Wykorzystując jeden z algorytmów elekcji, węzły wybierają spośród siebie jeden (*Master*) odpowiedzialny za przyjmowanie zapytań od klientów, oraz przekierowanie ich do węzłów odpowiedzialnych za ich odsługę (moduł API). Pozostałe węzły modułu Load Balancera pracują w trybie *Slave*. Wraz z przekierowaniem zapytania do wykonania przez węzeł pracujący w trybie Master jest ono również buforowane na węzłach typu *Slave*. Dzięki temu jeśli wykryta zostanie awaria węzła, odpowiedzialnego za obsługę zapytania (moduł API), możliwe będzie odzyskanie zapytania z bufora i przekazanie go do obsługi innemu węzłowi modułu API. Po wykryciu awarii węzła obsługującego zapytanie za przekazanie zapytania do innego węzła odpowiedzialny jest load balancer pracujący w trybie *Master.* Aplikacja kliencka początkowo nie posiada informacji o tym, który w węzłów został wybrany jako główny Load Balancer (pracujący w trybie *Master*) i do którego powinna kierować swoje zapytania. Na podstawie pliku konfiguracyjnego może wybrać dowolny węzeł obsługujący moduł Load Balancer i skierować do niego swoje zapytanie. Gdy moduł ten okaże się modułem pracującym w trybie *Slave*, prześle on w odpowiedzi przekierowanie do modułu *Master*.
- API - Jest to moduł odpowiedzialny za główną logikę aplikacji. Przyjmuje zapytania oraz jest w stanie je zinterpretować i na nie odpowiedzieć. W trakcie swojej pracy łączy się z warstwą wewnętrzną w celu pobrania bądź zapisania informacji w bazie danych. Po wykonaniu zapytania zwraca odpowiedź do odpowiedniego węzła Load Balancera.
- MP (Membership Protocol) - Jest to moduł działający na wszystkich węzłach aplikacji serwerowej służący do tworzenia oraz zarządzania listą aktywnych węzłów, oraz wykrywania ich awarii. Chcemy stworzyć własną implementację jednego ze znanych rozwiązań opartych na zasadzie virusowego przekazywania informacji (*Gossip protoco*l) jak na przyład protokół *SWIM*.

2.2. Wymagania funkcjonalne

Wymagania funkcjonalne:

- system pozwala użytkownikom magazynować pliki graficzne i udostępniać je innym użytkownikom,
- system umożliwia użytkownikom wybranie, kto ma dostęp do wgranych przez nich plików graficznych,
- użytkownicy mogą usunąć wgrane wcześniej pliki graficzne,
- użytkownicy mogą pobierać pliki graficzne udostępnione im przez innych użytkowników.

2.3. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne:

- każda z warstw powinna być uruchomiona na minimum 2 węzłach, w celu zapewnienia odporności na awarie węzła,
- system zapewnia realizację usługi w trakcie awarii w czasie obsługi,
- Konfiguracja dla części serwerowej i klientów usługi odbywa się przy pomocy jednego pliku,
- system pozwala na uwierzytelnianie węzłów,
- System zapewnienia odporność na utratę danych za pomocą redundancji.

3. Harmonogram realizacji projektu i przydział ról

W ramach spotkań ustalono następujący podział ról w zespole:

Kierownik projektu:	Mikołaj Cybulski
Architekt:	Przemysław Łada
Zarządzanie repozytorium i przechowywanie kopii zapasowych:	Mikołaj Cybulski, Michał Chilczuk
Dokumentalista:	Karolina Drabarz, Amadeusz Sadowski
Tester:	Karolina Drabarz, Ivan Dziemianczyk
Handlowiec:	Michał Haponiuk
Specjalista od narzędzia Docker:	Przemysław Łada

Wszystkie osoby z wyjątkiem kierownika projektu wejdą w skład zespołu programistycznego.

Zadania będą umieszczane w serwisie internetowym służącym do zarządzania projektami o nazwie *MeisterTask*. Najpierw dodane zostaną większe, zbiorcze zadania, które następnie zostaną rozbite na mniejsze, atomowe zadania z łatwiejszym do oszacowania czasem na ich wykonanie. Zarządzanie zadaniami będzie oparte o widok tablicy *Kanban* (Do wykonania, Postęp, Zakończone). Zadania będą rozdzielane przez kierownika projektu w porozumieniu z członkami zespołu programistycznego.

Celem na zakończenie drugiego etapu jest uzyskanie funkcjonalnej warstwy zewnętrznej odpowiedzialnej za przyjęcie i przekazanie do przetworzenia zapytania od klienta. Po wykonaniu tej części łatwiej będzie oszacować nakład na pozostałe zadania i wtedy nastąpi ostateczne określenie dokładniejszej wersji harmonogramu realizacji projektu.

Na dzień dzisiejszy określono następujące zadania do realizacji w ramach całego projektu:

- 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,
- Narzędzie do monitorowania węzłów,
    - Zbieranie i prezentowanie logów,
    - Synchronizacja zegarów systemowych.

Kroki milowe w ramach projektu:

Data	Opis
29.04.2016r.	● Poznanie narzędzia Docker ● Stworzenie środowiska kontenerów Docker ● Ustalenie architektury rozwiązania ● Stworzenie listy zadań do realizacji w ramach projektu ● Stworzenie środowiska do zarządzania przebiegiem projektu
06.05.2016r.	● Implementacja funkcjonalnej warstwy zewnętrznej
09.05.2016r.	● Rozbicie zadań na mniejsze i dokładne ich oszacowanie ● Ustalenie ostatecznego harmonogramu realizacji
03.06.2016r.	● Zakończenie implementacji rozwiązania
06.06.2016r. - 10.06.2016r.	● Testy końcowe i poprawki ● Skompletowanie dokumentacji

Organizacja środowiska programistycznego:

Jako domyślny język programowania wybraliśmy jezyk Python. Zdecydowaliśmy o jego użyciu względu na wygodę pracy oraz możliwość szybkiego prototypowania. Zdajemy sobię sprawę z jego ograniczeń, ale zależy nam bardziej na wydajności pisania kodu programu niż wydajności jego wykonania. Chcemy również wykorzystać różne biblioteki i frameworki tego języka takie jak:

- Flask http://flask.pocoo.org/
- Requests http://docs.python-requests.org/en/master/
- Twisted https://twistedmatrix.com/trac/

Wykorzystujemy program GIT do kontroli wersji, a cały kod naszej aplikacji przechowujemy w repozytorium Github dostępnym pod adresem:

https://github.com/mcybuls1/rso-project

Wszystkie tworzone przez nas za pomocą Docker narzędzia obrazy kontenerów są zarządzane przez narzędzie DockerHub. Utworzona została specjalna organizacja : *“rsogrupag”* gdzie będziemy publikować tworzone przez nas obrazy kontenrów.