Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych

Rok akademicki 2016/2017

Tomasz Rydzewski
Magdalena Malenda
Joanna Ohradka
Włodzimierz Szewczyk
Piotr Kuciński
Dominik Giżyński
Michał Herman

Projekt RSO

1. Projekt

1.1. Harmonogram

Poniżej przedstawiony został harmonogram projektu. Diagram Gantta projektu został dołączony jako dodatek B.

Tablica 1.1. Harmonogram projektu

Nazwa zadania	**Data rozpoczęcia**	**Data zakończenia**
Organizacja struktury projektu	16-03-21	16-03-29
Pierwsze spotkanie projektowe	16-03-31	16-03-31
Wystartowanie repozytorium Git	16-04-01	16-04-04
Ustalenia funkcjonalne i niefunkcjonalne	16-04-01	16-04-05
Utworzenie koncepcji architektury systemu	16-04-01	16-04-20
Utworzenie koncepcji wykorzystania narzędzia docker	16-04-01	16-04-20
Utworzenie dokumentu dobrych praktyk	16-04-15	16-04-21
Utworzenie tutoriala docker	16-04-18	16-04-21
Dodanie rozwiązania docker do repozytorium	16-04-19	16-04-20
Doprecyzowanie wymagań przedmiotu	16-04-19	16-04-20
Wybranie algorytmów	16-04-19	16-04-20
Ustalenie prezentacji na drugi krok milowy	16-04-21	16-04-21
Modułowy podział projektu	16-04-21	16-04-22
Szczegłowa koncepcja rozwiązania	16-04-25	16-04-28
Ustawienie środowiska programistycznego	16-04-20	16-04-20
Utworzenie pierwszych scenariuszy testowych (do prezentacji pierwszego kroku milowego)	16-04-19	16-04-22
Ustalenie wykorzystywanej bazy danych	16-04-21	16-04-25
Postawienie bazy danych	16-04-26	16-04-27
Część programistyczna	16-04-25	16-06-03
Zebranie dokumentacji na pierwszy krok	16-04-22	16-04-22
I krok milowy	16-04-29	16-04-29
Utworzenie wszystkich scenariuszy testowych	16-04-25	16-05-03
Odłożenie bruncha na II krok milowy	16-04-29	16-04-29
II krok milowy	16-05-06	16-05-06
Ustalenie prezentacji na trzeci krok milowy	16-05-09	16-05-11
Testy bezpieczeństwa	16-05-30	16-06-03
Testy fukcjonalne	16-05-30	16-06-03
Testy awaryjności	16-05-30	16-06-03
III krok milowy	16-06-10	16-06-10

2. Wymagania

2.1. Wymagania funkcjonalne

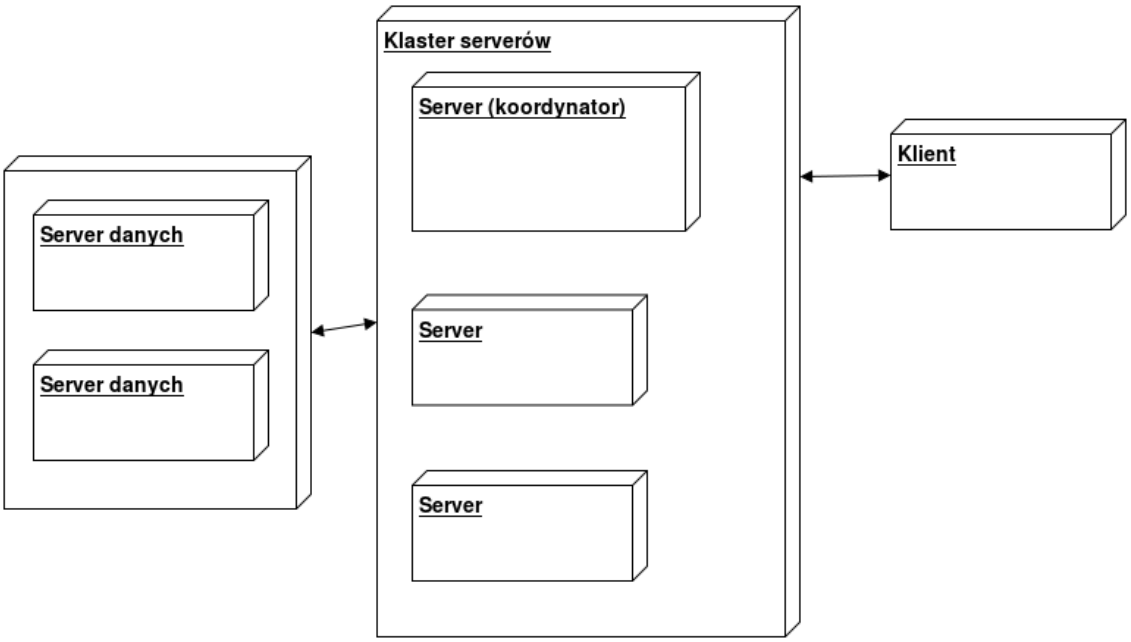
— Przechowywanie wrażliwych danych medycznych
— Udostępnianie danych odpowiednim osobom w zależności od uprawnień
— Prosta, konsolowa aplikacja kliencka wysyłająca zapytania do serwera i wyświetlająca wyniki

2.2. Wymagania niefunkcjonalne

— Wspólny testowy plik konfiguracyjny dla części serwerowej i klienckiej
— Współbieżne oprogramowanie realizujące część serwerową
— Odporność na uszkodzenie węzła
— Realizacja usługi w trakcie awarii w czasie obsługi
— Możliwość ponownego wpięcia węzła, z którym utracono łączność
— Odporność na próbę wpięcia wrogiego, nieuprawnionego węzła
— Zarządzanie zasobami transparentne dla oprogramowania klienckiego
— Większy niż 1 poziom redundancji danych
— Maksymalna ilość przechowywanych danych : 1Gb
— Uruchamianie i zamykanie części serwerowej jednokrotnym wywolaniem skryptu na każdym węźle
— Środowisko Linux Ubuntu 14.04 LTS

3. Architektura

3.1. Opis ogólny architektury systemu



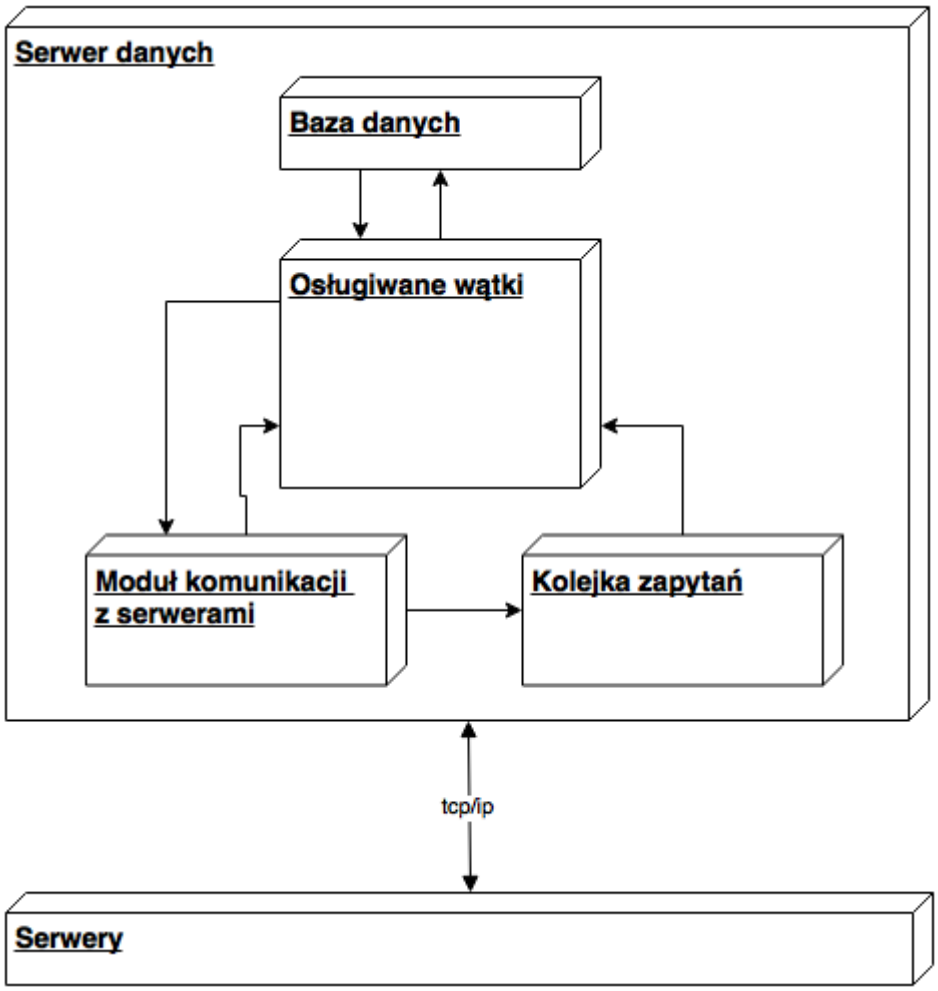
Rysunek 3.1. *Schemat architektury systemu*

System po stronie serwerowej składa się z dwóch warstw. Wewnętrznej, która przechowuje wrażliwe informacje medyczne i zewnętrznej udostępniającej informację oprogramowaniu klienckiemu. Część wewnętrzną stanowi system rozproszonej bazy danych, a część zewnętrzną klaster serwerów.

3.2. Baza danych

Zastosowanie prostej replikacji całego serwera danych. Poziom redundancji danych jest równy ilości serwerów danych. Zrealizowany zostanie model spójności sekwencyjnej. Każda zmiana w dowolnej bazie bazie powoduje rozpropagowanie jej na pozostałe serwery danych. W projekcie przewidziane są dwa serwery danych, co daje redundancję równą 2.

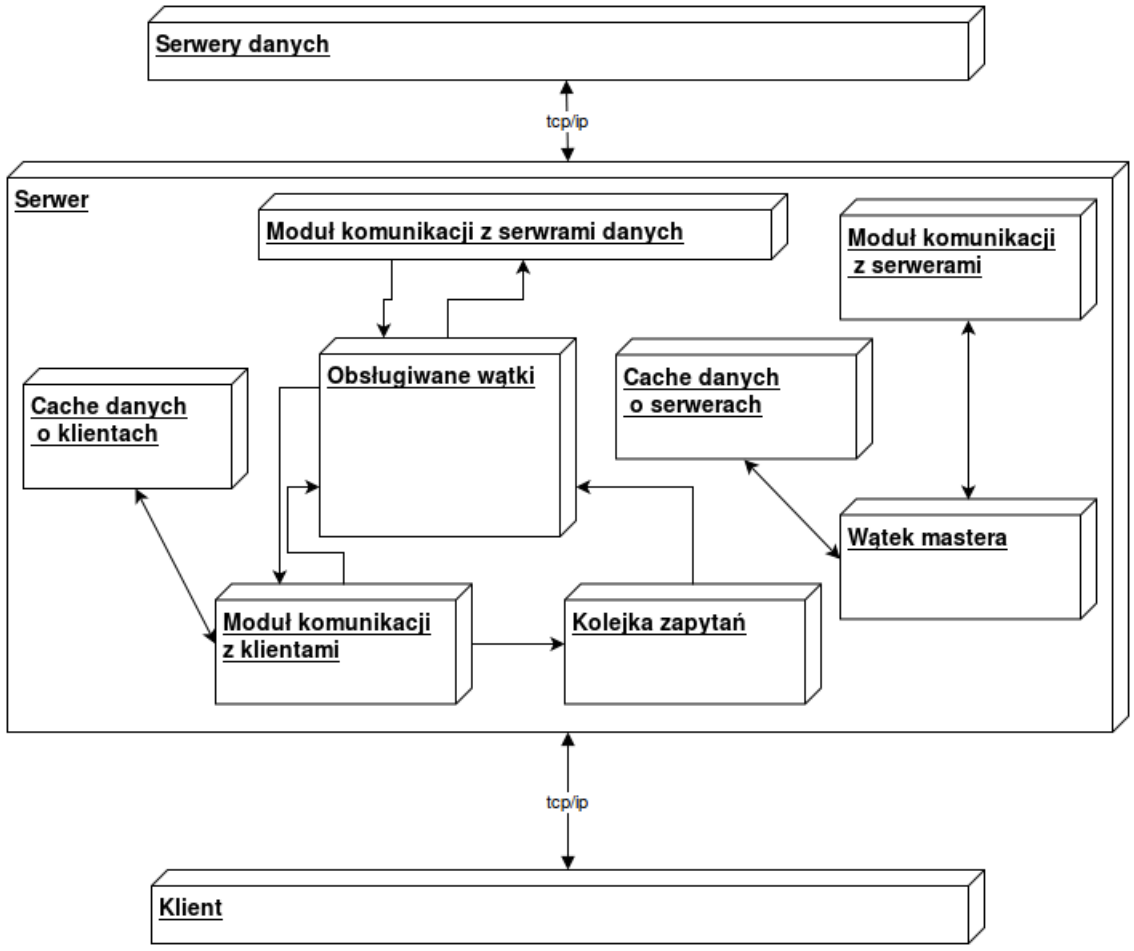
Komunikacja z serwerami odbywa się przez tcp/ip. W komunikacji pośredniczy moduł komunikacyjny, który oczekuje na zapytania. Każde zapytanie jest obsługiwane w oddzielnym wątku. Zapytania są kolejkowanie lub odrzucane, gdy zapytań będzie za dużo. Komunikacja jest szyfrowana przy pomocy RSA.



Rysunek 3.2. *Schemat serwera bazy danych*

3.3. Serwer

Serwer odbiera zapytania od klienta i pobiera dane z bazy. W tym celu nasłuchuje w oczekiwaniu na połączenie od klienta. Komunikacja klient-serwer odbywa się przez tcp/ip. Po odebraniu żądania , tworzony jest nowy wątek, który realizuje zapytanie, komunikuje się z bazą danych oraz formułuje odpowiedź. Serwer może obsłuzyć ograniczoną liczbę klientów. Nadamiarowe zapytania może przekazać innym serwerom, a jeżeli wszystkie są zajęte to wstawia do kolejki oczekujących zapytań lub odsyła odpowiedni komunikat, jeżeli w kolejce nie ma miejsc. Zadaniem serwera jest też autoryzacja łączącego się klienta, sprawdzenie jego uprawnień. Autoryzacja i uwierzytelnienie odbywa się przez podanie loginu i hasła. Gdy serwer dostanie zapytanie od niezalogowanego klienta odsyła prośbę o login i hasło. Po autoryzacji klient jest zapisywany w tablicy aktywnych klientów w cachu, następnie serwer generuje, zapisuje i wysyła token, który będzie służył do uwierzytelniania przy kolejnych żądaniach aż do momentu wylogowania lub gdy użytkownik przez okreslony czas będzie nieaktywny. Dane o aktywnych klientach nie są przechowywane na wszystkich węzłach, więc jeżeli jeden z nich ulegnie awarii konieczna



Rysunek 3.3. *Schemat serwera aplikacyjnego*

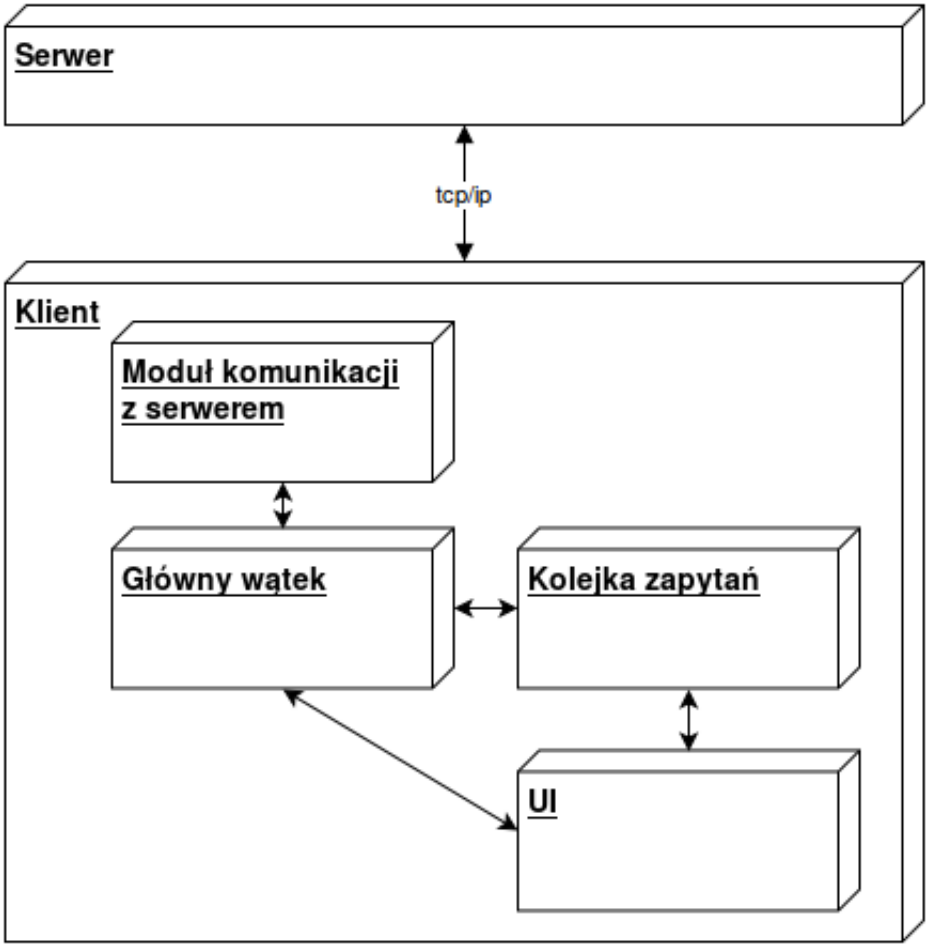
będzie ponowna autoryzacja i uwierzytelnieie na innym węźle. Jeden z serwerów jest koordynatorem. Koordynator kontroluje pracę pozostałych serwerów, sprawdza, czy wszystkie są dostępne, autoryzuje nowy węzeł, który chce się połączyć. Tworzy i rozsyła do wszystkich węzłów tablicę aktywności serwerów oraz serwerów danych. Sprawdzanie obecności serwerów odbywa się poprzez cykliczne odpytywanie. Gdy master ulegnie awarii jego rolę przejmuje inny z serwerów. Wszystkie serwery znają liczbę i adresy pozostałych. Każdy z serwerów ma przyporządkowany numer. Masterem staje się ten o najniższym numerze. Pierwszy serwer, który zauważy, że nie ma mastera wysyła komunikat do wszystkich węzłów o niższym numerze, jeżeli nikt nie odpowie to serwer staje się nowym koordynatorem i wysyła do wszystkich pozostałych węzłów informujący komunikat. Jeżeli któryś z węzłów o niższym numerze odpowie to on przejmuje kontrolę. Brak koordynatora będzie zauwazony, gdy węzeł nie zostanie odpytany w odpowiednim czasie.

3.3.1. Autoryzacja węzłów

Każdy węzeł posiada nazwę oraz numer. Aby dołączyć się do klastra serwerów musi wysłać masterowi numer oraz swoją nazwę zaszyfrowaną kluczem publicznym mastera. W odpowiedzi dostaje potwierdzenie i aktualny stan systemu, aby

zaktualizować listę aktywnych serwerów. Klucze publiczne są zapisywane w plikach konfiguracyjnych serwera i klienta.

3.4. Aplikacja kliencka



Rysunek 3.4. *Schemat aplikacji klienckiej*

Zadaniem aplikacji klienckiej jest nawiązanie połączenia z jednym z serwerów. Adresy serwerów są określone w pliku konfiguracyjnym. Jeżeli połączenie z jednym z węzłów zakończy się niepowodzeniem, następuje próba połączenia się z kolejnymi. Aplikacja umożliwia zalogowanie się i dostęp do funkcji oferowanych przez API serwera. Jeżeli jakieś żądanie się nie powiedzie pomimo dostępności serwera, następuje ponowna próba, wysłania żądania ale do innego serwera. Klient musi aktualizować listę serwerów poprzez cykliczne odpytywanie.

3.5. Plik konfiguracyjny

W pliku konfiguracyjnym klienta znajdują się adresy, porty i nazwy serwerów oraz ich klucze publiczne. W pliku konfiguracyjnym serwerów znajdują się adresy,

porty i nazwy serwerów, adresy serwerów baz danych, maksymalna liczba jednocześnie obsłużonych wątków, okres odpytywania węzłów przez serwer główny, czas trwania sesji z klientem, klucze publiczne serwerów.

4. Docker

4.1. Opis rozwiązania Docker

Docker jest narzędziem ułatwiającym proces tworzenia, dystrybucji i wdrażania oprogramowania. Pozwala on na umieszczenie aplikacji wraz z jej bezpośrednimi zależnościami w kontenerze i uruchomienie jej na dowolnej maszynie z systemem Linux. Aplikacje działające za pomocą Dockera są odizolowane od infrastruktury, dzięki czemu możliwe jest uruchomienie kilku niezależnych kontenerów z aplikacjami symulujących pracę w środowisku rozproszonym. Jednocześnie narzędzie to zapewnia niewielkie zużycie pamięci dzięki współdzieleniu warstw UFS obrazów (ang. Union File System) pomiędzy kontenerami. Współdzielenie jądra systemu pomiędzy kontenerami, a systemem gospodarza zapewnia natomiast krótkie czasy uruchomienia. Ze względu na te cechy korzystanie z aplikacji uruchomionej za pomocą Dockera jest niemal tak wydajne, jak działającej na systemie gospodarza.

Podstawowymi komponentami Dockera są:

— *Docker Engine* - platforma do tworzenia kontenerów na uruchamiane przez użytkownika aplikacje,

— *Docker Hub* – oficjalne repozytorium obrazów przechowujące obrazy udostępniane przez użytkowników

Docker działa w architekturze klient – serwer. Do komunikacji klienta z demonem wykorzystywany jest protokół HTTP i odbywa się ona poprzez gniazda (ang. sockets) lub RESTful API. Klient Dockera jest podstawowym interfejsem komunikacyjnym z Dockerem – przyjmuje komendy z określonego zestawu poleceń wpisywane przez użytkownika i przekazuje je do demona. Demon odpowiada za budowanie obrazów, uruchamianie i dystrybucję kontenerów. Klient i demon mogą działać na tym samym systemie lub ich funkcje mogą zosać rozdzielone pomiędzy różne hosty.

Podstawowymi pojęciami Dockera są:

— *obrazy* – komponent odpowiadający budowaniu,

— *rejestry* – komponent odpowiadający dystrybucji,

— *kontenery* – komponent odpowiadający uruchamianiu.

Obraz jest to szablon służący tylko do odczytu, stanowiący podstawę do utworzenia kontenera. Obraz może zawierać np. system operacyjny Ubuntu z serwerem Apache oraz zainstalowaną aplikacją webową, którą chcemy uruchomić. Za pomocą Dockera mamy możliwość budowania nowych obrazów, aktualizowania istniejących, pobierania obrazów stworzonych przez innych i udostępniania własnych. Każdy obraz składa się z warstw tworzących ujednolicony system plików (*ang. UFS*). Ta technologia powoduje, że obrazy Dockera są lekkie – wprowadzenie zmian w aplikacji nie wiąże się z przebudową całego obrazu, a jedynie z aktualizacją lub dodaniem danej warstwy.

Każdy obraz ma obraz bazowy (np. obraz systemu Ubuntu lub obraz utworzony przez użytkownika), na podstawie którego jest budowany za pomocą zestawu instrukcji. Każda instrukcja powoduje dodanie warstwy do naszego obrazu (taką instrukcją może być np. wywołanie komendy, dodanie pliku lub folderu, instalacja pakietu, utworzenie zmiennej środowiskowej). Polecenia tworzące obraz Dockera przechowywane są w pliku Dockerfile. Podczas budowania obrazu Docker odczytuje kod źródłowy zawarty w pliku Dockerfile, wykonuje zapisane instrukcje i zwraca końcowy obraz.

Przykładowe operacje na obrazie:

— Pobieranie obrazu:

```
1 docker pull {nazwa obrazu}
```

— Wyświetlanie lokalnie dostępnych obrazów:

```
1 docker images
```

— Wyświetlenie warstw składających się na obraz:

```
1 docker history {id lub nazwa obrazu}
```

— Usunięcie obrazu

```
1 docker  rmi {id lub nazwa obrazu}
```

Rejestry są miejscem gdzie można udostępniać i skąd można pobierać obrazy. Mogą one być zarówno publiczne, jak i prywatne. Publiczny rejestr Dockera stanowi Docker Hub zawierający bazę obrazów stworzonych przez użytkowników Dockera. Istnieje również lokalne repozytorium na maszynie użytkownika. Za pomocą klienta Dockera możliwe jest przeszukiwanie opublikowanych obrazów oraz pobieranie ich w celu utworzenia kontenera.

Kontener tworzony jest na podstawie obrazu, który zawiera informacje o tym, co przechowuje kontener, jaki proces ma zostać uruchomiony po jego utworzeniu oraz inne dane konfiguracyjne. Kontener składa się z zestawu ujednoliconych warstw tylko do odczytu, pochodzących z obrazu kontenera oraz z pojedynczej warstwy do odczytu i zapisu umożliwiającej działanie procesów uruchamianych w kontenerze. Na kontenerach można wykonywać podstawowe operacje: uruchomić, zatrzymać, przenieść i usunąć.

Przykładowe operacje na kontenerze

— Tworzenie i uruchamianie kontenera:

```
1 docker runl {nazwa lub id obrazu}
```

Uruchomienie tego polecenia z parametrem -d powoduje uruchomienie kontenera działającego w tle, natomiast parametr –rm=true powoduje, że kontener zostanie usunięty natychmiast po wykonaniu zadania.

— Wyświetlanie wszystkich kontenerów:

```
1 docker ps –a
```

— Zatrzymanie kontenera

```
1 docker stop {id lub nazwa kontenera}
```

— Usunięcie kontenera

```
docker rm {id lub nazwa kontenera}
```

Docker udostępnia również możliwość automatycznego budowania obrazu w synchronizacji z systemem kontroli wersji (GitHub lub Bitbucket). Dzięki umieszczeniu w repozytorium pliku *Dockerfile* oraz powiązaniu konta DockerHub I kontem w wybranym systemie kontroli wersji, po każdorazowej aktualizacji kodu w repozytorium budowany, na jego podstawie budowany jest aktualny obraz Dockera i zapisywany w repozytorium DockerHub.

4.2. Raport z przykładowych uruchomień

4.2.1. Uruchomienie aplikacji *'Hello World'*

Aplikacja uruchamiana jest poleceniem

```
$ docker run ubuntu /bin/echo 'Hello world'
```

gdzie:
— *docker run* – uruchamia kontener
— *ubuntu* – obraz, na podstawie którego tworzony jest kontener (obraz systemu ubuntu)
— */bin/echo 'Hello world'* – polecenie, które ma zostać wywołane wewnątrz utworzonego kontenera

Odpowiedź Dockera na zadaną komendę:

```
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
759d6771041e: Pull complete
8836b825667b: Pull complete
c2f5e51744e6: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:b4dbab2d8029edddfe494f42183de20b7e2e871a424ff16ffe7b15a31f102536
Status: Downloaded newer image for ubuntu:latest
Hello world
```

Docker przeszukuje dostępne lokalnie obrazy. Jeżeli nie znajduje danego obrazu lokalnie, wyszukuje i pobiera go z repozytorium Docker Hub. Następnie uruchamia kontener, wykonuje polecenie wyświetlenia napisu "Hello world" I zatrzymuje kontener.

Aplikację można uruchomić również w trybie interaktywnym, stosując flagi -i -t za pomocą polecenia

```
$ docker run -t -i ubuntu /bin/bash
```

gdzie flaga - t powoduje uruchomienie terminala w kontenerze, natomiast flaga - i pozwala wczytywać w kontenerze znaki wpisywane na klawiaturze. Odpowiedź Dockera:

```
root@879d9c1665ca:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var boot  etc  lib   media
```

```
opt  root  sbin  sys  usr
root@879d9c1665ca:/# exit
exit
```

Innym sposobem uruchomienia aplikacji jest uruchomienie jej w trybie demona dzięki zastosowaniu flagi -d. Poniższe polecenie spowoduje wyświetlenie napisu 'Hello world' co 1 sekundę:

```
$ docker run -d ubuntu /bin/sh -c "while true; do echo hello world;
sleep 1; done"
```

Sprawdzenie, czy kontener został uruchomiony:

```
$ docker ps
```

Odpowiedź Dockera:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
c99a2503f886        ubuntu              ''/bin/sh -c 'while tr''   2 minutes ago
Up 2 minutes                            evil_morse
```

Sprawdzenie wyjścia kontenera:

```
$ docker logs evil_morse
```

Odpowiedź Dockera:

```
hello world
hello world
hello world ....
```

Zatrzymanie kontenera:

```
$ docker stop evil_morse
```

4.2.2. Uruchomienie aplikacji webowej

Aplikacja uruchamiana jest poleceniem:

```
$ docker run -d -P training/webapp python app.py
```

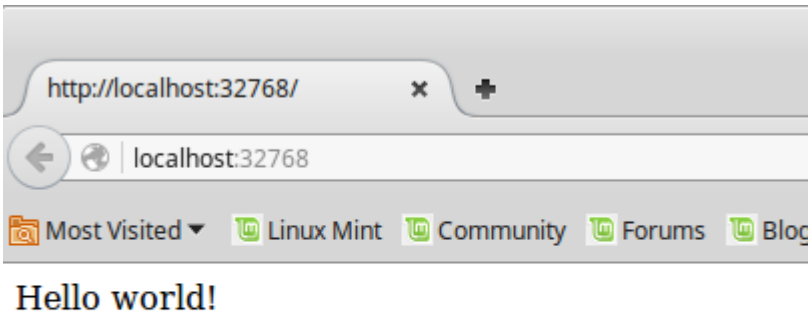
gdzie
— *docker run* – uruchamia kontener
— *-d* - flaga powodująca działanie w tle kontenera
— *-P* – flaga nakazująca Dockerowi odwzorowanie portów kontenera na porty na maszynie gospodarza
— *training/webapp* – obraz, na podstawie którego tworzony jest kontener
— *python app.py* – polecenie, które ma zostać wywołane wewnątrz utworzonego kontenera (uruchomienie aplikacji webowej)
Odpowiedź Dockera:

```
Unable to find image 'training/webapp:latest' locally
latest: Pulling from training/webapp
e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
10bbbc0fc0ff: Pull complete
fca59b508e9f: Pull complete
e7ae2541b15b: Pull complete
9dd97ef58ce9: Pull complete
a4c1b0cb7af7: Pull complete
Digest: sha256:06e9c1983bd6d5db5fba376ccd63bfa529e8d02f23d5079b8f74a616308fb11d
Status: Downloaded newer image for training/webapp:latest
426c053faa028c411e8ca32f21ec9907da7b442dd83a321f4e47697a8e7cfb7f
```

Obraz nie został znaleziony w lokalnym rejestrze, dlatego pobrano go z rejestru Docker Hub. Za pomocą polecenia *docker ps -l* można sprawdzić szczegółowe informacje dotyczące ostatnio uruchomionego kontnera.

```
 $ docker ps -l
CONTAINER ID   IMAGE              COMMAND              CREATED
STATUS              PORTS                    NAMES
426c053faa02   training/webapp    "python app.py"      13 minutes ago
Up 12 minutes       0.0.0.0:32768->5000/tcp   berserk_euler
```

Kolumna *ports* mówi nam o tym, że port 5000 w kontenerze jest odwzorowany na port 32768 na maszynie lokalnej. Działanie aplikacji możemy sprawdzić wyszukując w przeglądarce port 32768.



Rysunek 4.1. *Działająca aplikacja webowa*

4.2.3. Tworzenie własnego obrazu

Własny obraz można utworzyć na dwa sposoby:

— Poprzez aktualizację kontenera stworzonego na podstawie obrazu oraz zapisanie wprowadzonych zmian do obrazu
— Za pomocą pliku *Dockerfile*

W pierwszym przypadku mamy utworzony kontener na podstawie obrazu i wprowadziliśmy do niego nowe dane, np. zainstalowaliśmy program. Nowy obraz tworzymy za pomocą polecenia:

```
$ docker commit -m "Added program" -a "Author" bd625e9176a7 author/ubuntu:v2
```

gdzie
— *docker commit* – zapisuje stn kontenera w postaci obrazu
— *-m* – flaga pozwalająca zapisać co zostało zmienione w obrazie
— *-a* – flaga informująca o autorze wprowadzonych zmian
— *bd625e9176a7* – numer ID kontenera, na bazie którego tworzony jest nowy obraz
— *author/ubuntu:v2* – nazwa repozytorium oraz znacznik wersji utworzonego obrazu
Po wywołaniu tego polecenia moemy sprawdzić listę obrazów w lokalnym repozytorium:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
author/ubuntu       v2                  adbc62b82a11        19 seconds ago
725.1 MB
ubuntu              latest              b72889fa879c        11 days ago
188 MB
training/webapp     latest              6fae60ef3446        11 months ago
348.8 MB
```

Tworzenie obrazu z pliku *Dockerfile* przebiega następująco. Najpierw tworzymy katalog na nasz plik, a w nim plik *Dockerfile*:

```
$ mkdir dockerbuild
$ cd dockerbuild/
$ touch Dockerfile
```

Zawartość pliku może wyglądać następująco:

```
FROM ubuntu:14.04
MAINTAINER Author <author@example.com>
RUN apt-get update
RUN apt-get install build-essential
RUN apt-get install qt5-default
```

Plik *Dockerfile* składa się z instrukcji, których nzwy pisane są wielkimi literami poprzedzających polecenia: *INSTRUKCJA polecenie*. Wyjaśnienie instrukcji wykorzystanych w powyższym pliku:
— *FROM ubuntu:14.04* – ta komenda mówi o tym, że obraz bazuje na obrazie systemu ubuntu wersji 14.04
— *MAINTAINER Author <author@example.com>* - wskazanie autora obrazu
— *RUN ...* - instrukcja RUN nakazuje wykonać w kontenerze dane polecenia
Następnie budujemy obraz na podstawie pliku *Dockerfile* (przy wywołaniu tej komendy musimy znajdować się w katalogu, w którym jest nasz plik *Dockerfile*):

```
$ docker build -t author/ubuntu:v2 .
```

4.2.4. Korzystanie z Docker Hub

Utworzony obraz może zostac udostępniony w repozytorium Docker Hub, a następnie pobrany przez członków naszego zespołu. W tym celu należy się zalogować na swoje konto Docker Hub za pomoc klienta dockera:

```
1 $ docker login
```

Po podaniu danych logowania możemy udostępnić obraz:

```
1 $ docker push author/ubuntu:v2
```

Następnie może on zostać pobrany przez innch użytkowników:

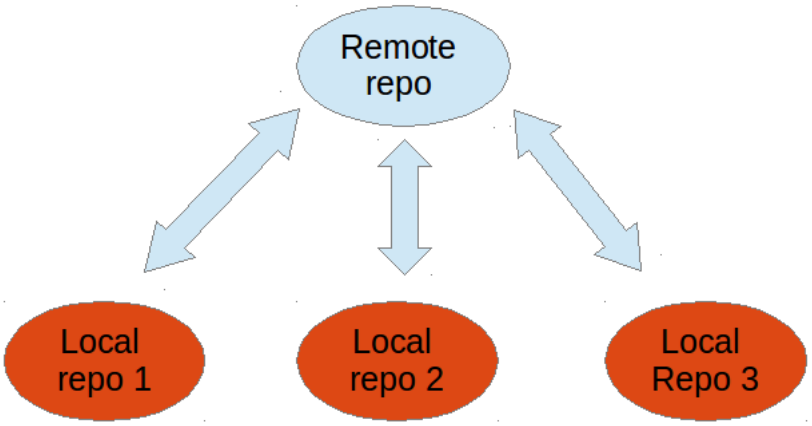
```
1 $ docker pull author/ubuntu:v2
```

Dodatek A. Wprowadzenie i instrukcja użytkowania systemu kontroli wersji Git

A.1. Wprowadzenie

Do wspomagania równoległego, rozgałęzionego procesu rozwoju projektu przez wielu programistów zdecydowano się wykorzystać rozproszony system kontroli wersji Git.

Git jest obecnie najbardziej popularną implementacją rozproszonego systemu kontroli wersji. W przeciwieństwie do innych systemów kontroli wersji, Git nie zapamiętuje zmian między kolejnymi rewizjami, lecz kompletne obrazy. Każdy z użytkowników posiada lokalną kopię repozytorium na swoim własnym komputerze po uprzednim sklonowaniu repozytorium zewnętrznego (zdalnego). Pozwala to na pracę w trybie off-line i wprowadzanie zmian w wersji lokalnej projektu i efektywną pracę nad dużymi projektami.



Rysunek A.1. *Schemat połączeń między repozytoriami Git*

Następnie zmiany mogą być wymieniane między lokalnymi repozytoriami. Służy do tego repozytorium zewnętrzne (remote repository) działające na serwerze. Serwisem przechowującym rozwijany projekt jest GitHub (`https://github.com/`), który udostępnia darmowy hosting open source.

Repozytorium projektu jest dostępne pod adresem: `https://github.com/Gonz8/RSO-16L`

A.2. Przygotowanie do pracy i pierwsze pobranie zawartości repozytorium

Na samym początku wymagane jest posiadanie klienta Git zainstalowanego w swoim systemie. Wszelkie istotne informacje dotyczące korzystania z Git możemy uzyskać wpisując w terminalu:

```
$ git help
```

Następnie musimy określić własną nazwę oraz adres e-mail w systemie Git:

```
$ git config --global user.name ''Your Name''
$ git config --global user.email ''address@example.com''
```

Aby zaimportować repozytorium ze wspomnianego serwera należy wykonać polecenie:

```
$ git clone https://github.com/Gonz8/RSO-16L
```

Wszystkie pliki zostaną sklonowane do nowo utworzonego katalogu, z poziomu którego należy utworzyć lokalne repozytorium:

```
$ git init
```

Utworzone w ten sposób repozytorium jest już powiązane ze zdalną wersją (origin). Możemy to sprawdzić przy użyciu polecenia:

```
$ git remote -v
```

A.3. Użytkowanie

Po zaimportowaniu projektu oraz utworzeniu lokalnej kopii repozytorium można rozpocząć pracę z danymi. Aby sprawdzić status dokonanych zmian należy użyć w katalogu z kopią roboczą następującego polecenia:

```
$ git status
```

Dodanie nowego pliku, kilku plików lub katalogu do kopii roboczej wykonywane jest przy użyciu komendy:

```
$ git add <filename>
$ git add *
```

Aby zatwierdzić wszelkie dokonane zmiany w lokalnym repozytorium należy użyć polecenia:

```
$ git commit -a
```

a następnie podać treść/opis poczynionych zmian. W celu przechwycenia najnowszych zmian z serwera wykonujemy polecenie:

```
$ git fetch origin
```

natomiast, aby przechwycić zmiany z serwera i dodatkowo dołączyć je do własnego katalogu roboczego wykonujemy polecenie:

```
$ git pull
```

Wysłanie zmian poczynionych w wersji lokalnej do zdalnego repozytorium realizowane jest dzięki komendzie:

```
$ git push origin master
$ git push origin <branchname>
```

Git pozwala również na tworzenie, usuwanie i przełączanie się między gałęziami projektu, do wykonania tych operacji służą następujące polecenia:

```
$ git checkout –b <branchname>
$ git branch –d <branchname>
$ git checkout <branchname>
```

Wyświetlenie listy wszystkich gałęzi dostępnych w repozytorium możliwe jest poprzez komendę:

```
$ git branch
```

Natomiast w celu dołączenia innej gałęzi do obecnie aktywnej należy wykonać polecenie:

```
$ git merge <branchname>
```

Ostatnim również istotnym poleceniem jest wyświetlenie historii logów/commit'ów:

```
$ git log
```

Dodatkowo po zainstalowaniu pakietu ***gitk*** można wyświetlać graficzną prezentację historii zmian projektu.

A.4. Dodatkowe narzędzia

Mimo faktu, że Git jest rozproszonym systemem kontroli wersji zdecydowano się na wykonywanie dodatkowej i okresowej kopii zapasowej projektu (tzw. backup). Kopia bezpieczeństwa jest jednym z elementów utrzymania repozytorium i zabezpiecza przed utratą danych (np. awarii może ulec komputer głównego programisty, przez co istnieje ryzyko utraty lokalnej kopii repozytorium). Rozwiązaniem tego zagadnienia będzie okresowe wywoływanie napisanego skryptu, który aktualizuje zawartość sklonowanego repozytorium znajdującego się również w folderze powiązanym z naszym kontem Dropbox. Pozwala to w prosty sposób przechowywać kopię zawartości repozytorium na innym serwerze zewnętrznym (poza GitHub) i zabezpieczyć przed utratą danych.

Serwis GitHub świadczy szereg dodatkowych usług wspomagających rozwój projektu programistycznego. Zdecydowano się wykorzystać usługę Wiki, gdzie w łatwy sposób można wprowadzać i modyfikować istotne treści w kontekście projektu. Jest to miejsce, gdzie można w szybki i wygodny sposób odnaleźć uporządkowane informacje. Jednym z odnośników w tej usłudze jest zakładka "Dobre praktyki

kodowania", gdzie znajdują się wszystkie wspólnie ustalone przez programistów zasady pisania kodu pozwalające utrzymać przejrzystość i spójność kodu.

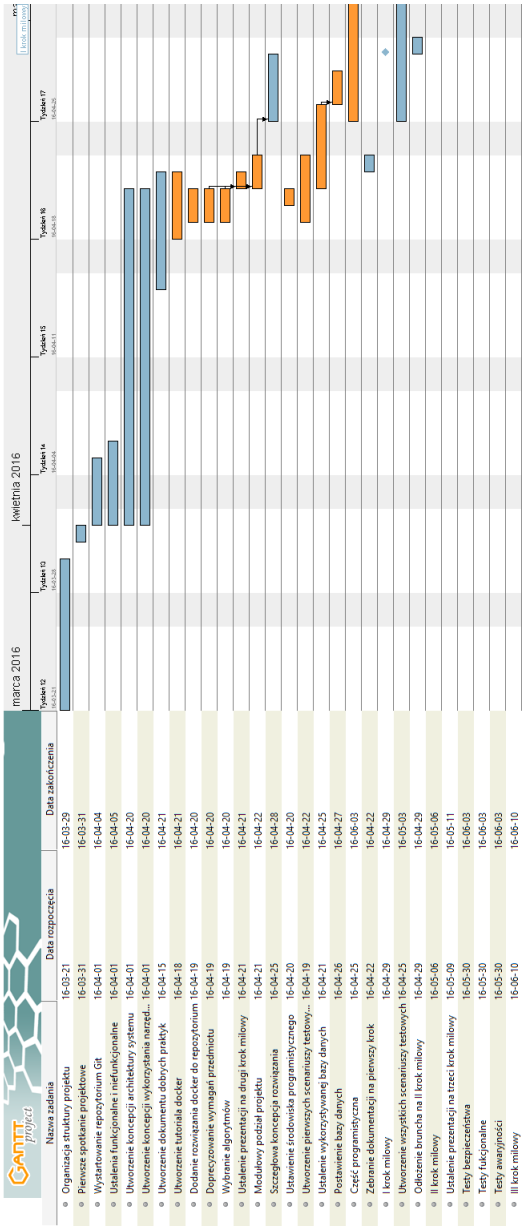
Ponadto zdecydowano się uruchomić usługę śledzenia zgłoszeń (tzw. *issue tracking*) na potrzeby wspierania testowania oprogramowania oraz zgłaszania napotkanych błędów i tworzenia dla nich poprawek.

A.5. Przydatne informacje

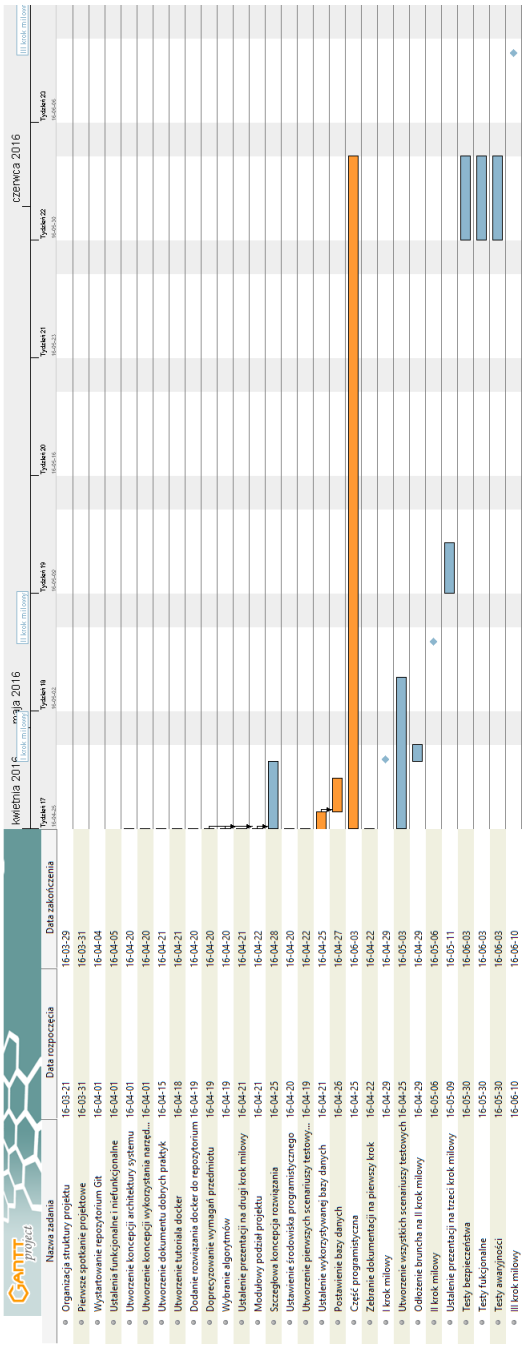
Instrukcja została opracowana na postawie materiałów znalezionych w sieci. Więcej informacji dotyczących użytkowania systemu kontroli wersji Git można znaleźć na stronach:

— `https://git-scm.com/docs/gittutorial`
— `http://www.vogella.com/tutorials/Git/article.html#git`

Dodatek B. Diagram Gantta



Rysunek B.1. *Diagram Gantta projektu, cz. 1*



Rysunek B.2. *Diagram Gantta projektu, cz. 2*

Dodatek C. Narzędzia

Poniżej znajduje się lista narzędzi i technologii użytych do realizacji projektu.

— C++
— Qt
— PostgreSQL
— GanttProject
— LaTeX
— Git