AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Bartłomiej Mucha

kierunek studiów: **informatyka stosowana**

Migracja serwisów internetowych i integracja usług w środowisku kontenerów Docker

Opiekun: **dr inż. Piotr Gronek**

Kraków, styczeń 2020

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....

(czytelny podpis)

Spis treści

Rozdział 1

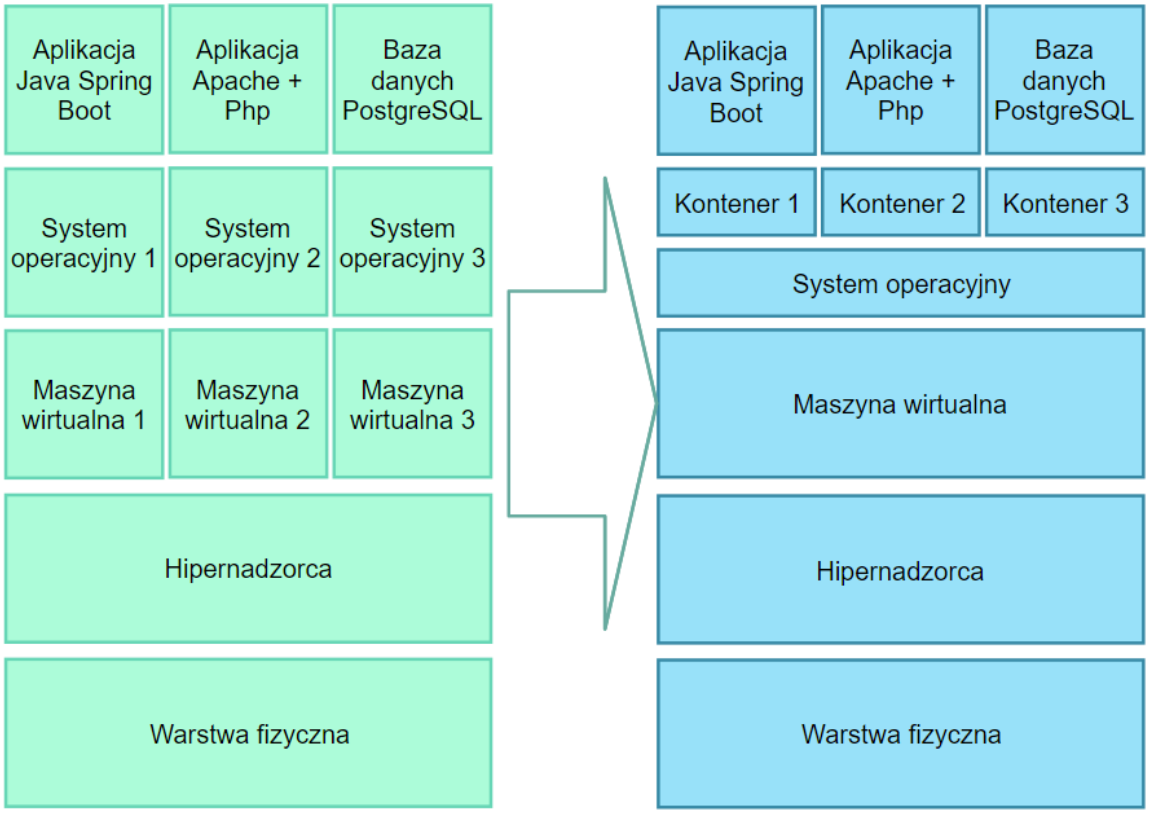
Wstęp

Możliwości, jakie oferują maszyny wirtualne, są nieocenione we współczesnym świecie. Systemy serwerowe, w skład których mogą wchodzić nawet tysiące powiązanych ze sobą lub indywidualnych serwisów, nie mogą występować albo są zasadniczo trudne do wdrożenia na jednej platformie o określonym systemie operacyjnym, konfiguracji i innych elementach wchodzących w skład szeroko rozumianego środowiska. Maszyny wirtualne są rozwiązaniem tego zagadnienia. Każda maszyna wirtualna może zostać stworzona niezależnie od architektury sprzętowej i oferuje dowolne środowisko, które jednocześnie będzie wyizolowane od środowisk gospodarza (*host*) i innych maszyn wirtualnych (*guest*). Korzyścią płynącą z izolacji środowiska jest, chociażby zwiększone bezpieczeństwo systemu, ułatwienie zarządzania serwisami oraz wdrażania nowych wersji aplikacji. Nierzadko używane są stare serwisy, a jednak spełniające swoje zadanie, które wymagają rozwiązań nieaktualnych już wersji elementów środowiskowych, niewspieranych przez nowsze wersje. Co za tym idzie wdrożenie np. dwóch aplikacji działających w ramach tej samej technologii, ale na różnych wersjach będzie tworzyć konflikt. Problemy tego typu zanikają w kontekście wirtualizacji, albowiem nic nie stoi na przeszkodzie osadzenia każdej pojedynczej aplikacji na osobnej maszynie. Wirtualizacja posiada jednak szereg wad. Taką wadą jest na przykład to, że każda maszyna wirtualna musi rezerwować określoną i niezmienną w czasie działania ilość zasobów, takich jak pamięć operacyjna (np. *RAM*) i masowa (nieulotna) oraz liczbę wątków procesora i wiele innych. Stawia to przed administratorem problem, związany z dobraniem parametrów, jakimi taka maszyna ma się cechować. Przydzielenie zbyt małej ilości zasobów sprawi, że serwis będzie niedomagał. Z kolei przydzielanie zbyt dużej ilości zasobów doprowadzi do marnowania się części zasobów, które mogłyby być lepiej wykorzystane. To, jak maszyna wirtualna zużywa zasoby, takie jak pamięć operacyjna, nie jest kontrolowane. Zarezerwowany zasób jest statyczny i nie jest zwalniany w przypadku braku użycia.

Rozwiązaniem tego problemu jest zmiana podejścia do koncepcji wirtualizacji, a mianowicie konteneryzacja serwisów. Platformy takie jakie *Docker*[1] czy *LXC*[2] pozwalają tworzyć niewymagające udziału hipernadzorcy wyizolowane środowiska o dynamicznie przydzielanych zasobach, tak zwane kontenery. Kontenery zarządzane przez platformę Docker współdzielą jądro systemowe z gospodarzem i innymi kontenerami, a w ramach osadzenia aplikacji wystarczy dostarczyć zbudowaną aplikację oraz wymagane do poprawnego jej działania składniki systemowe. W ten sposób można zaoszczędzić na użyciu pamięci twardej względem tej samej aplikacji wdrożonej na maszynie wirtualnej.

1.1 Cel pracy

Celem pracy jest przeniesienie aplikacji z maszyn wirtualnych na kontenery oraz opracowanie tej metodologii. W konsekwencji zostaną stworzone skalowalne środowiska usług internetowych, działające w oparciu o technologię kontenerów Docker na platformie Linux w systemie wirtualizacji *VMware ESX*[3] oraz porównanie i ocena korzyści wynikających z tych działań. Modelem przykładowym jest system aplikacji działających w technologii *Php*[4] i *Java*[5] z frameworkiem *Spring Boot*[6] oraz baza danych *PostgreSQL*[7] na trzech osobnych kontenerach Docker. Kontenery będą osadzone na zainstalowanej w systemie wirtualizacji ESX maszynie wirtualnej z system operacyjnym *Debian 10*[8]. Finalnie wdrożone zostaną realne serwisy w wydziałowej sieci komputerowej.



Rysunek 1.1: Schemat przekształcenia architektury serwerowej.

Plan pracy jest następujący. W części teoretycznej zostaną omówione mechanizmy stojące za działaniem maszyn wirtualnych i kontenerów na platformie Docker oraz ich porównanie. Następnie omówiony zostanie schemat wdrożenia aplikacji. Część praktyczna będzie zawierać szczegółowy opis przygotowania środowiska oraz wdrożenia aplikacji przykładowych.

1.2 Założenia projektu

Celem projektowej części pracy jest opracowanie skalowalnego środowiska usług internetowych, działającego w oparciu o technologię kontenerów Docker na platformie *Linux* w systemie wirtualizacji *VMware ESX*[3]. Usługi realizowane przez system mają obejmować obsługę: - internetowych serwisów aplikacyjnych działających w technologiach odpowiednio: *Php*[4], *Apache Tomcat*[9], *Python*[10] *Django*[11], witryn *CMS WordPress*[12], serwerów relacyjnych baz danych *MySQL*[13], *PostgreSQL*[7]. Elementem pracy będzie także migracja istniejących serwisów internetowych do ich odpowiadających im instancji wyżej wymienionych kontenerów *Docker*[1] oraz ocena uzyskanych korzyści pod kątem konsumowanych zasobów sprzętowych i wydajności działania.

Rozdział 2

Podstawa Teoretyczna

2.1 Wirtualizacja

Szeroko rozumiane urządzenie zwane komputerem składa się z dwóch kategorii komponentów: urządzeń fizycznych i oprogramowania. W skład urządzeń fizycznych wchodzą między innymi płyta główna, centralna jednostka przetwarzająca i urządzenia wejścia/wyjścia. Oprogramowanie składa się z warstw, najniżej, najbliżej sprzętu znajduje się oprogramowanie do zarządzania i komunikacji z urządzaniami, w wyższych warstwach znajdziemy aplikacje użytkowe. Zadaniem systemu operacyjnego jest zarządzanie zarówno zasobami sprzętowymi, jak i oprogramowaniem. Stanowi on interfejs pomiędzy maszyną a użytkownikiem, umożliwiając tym samym stosowanie komputerów do celów osobistych znanych nam z życia codziennego i profesjonalnych na linii człowiek — maszyna, ale i maszyna — maszyna. Ideą systemu operacyjnego jest istnienie środowiska, które dynamicznie będzie tłumaczyć zlecenia użytkownika na język maszynowy, następnie zlecać sprzętowi ich wykonanie i w drugą stronę informować użytkownika o rezultatach pracy maszyny oraz jej zapotrzebowaniu na dane, zachowując jednocześnie optymalne działanie pracy całego systemu i zawartych w nim urządzeń.

Rysunek 2.1: Schemat warstw systemu operacyjnego

Na rysunku 2.1 przedstawiony jest schemat budowy systemu operacyjnego. W skład samego systemu operacyjnego wchodzą sterowniki, jądro i powłoki systemowe. System operacyjny jest również podzielony na przestrzenie jądra i użytkownika. Wirtualizacja zatem jest symulacją warstwy fizycznej komputera, na której można zainstalować i używać dowolnego systemu operacyjnego. Symulować można różnego rodzaju architektury i parametry sprzętowe, jednak zalecane jest, by wirtualna platforma sprzętowa nie przekraczała możliwości rzeczywistego sprzętu. Wirtualizacji można dokonać na kilku poziomach.

2.1.1 Poziom architektury zestawu instrukcji (*ISA*)

Ten poziom jest stosowany w pełnej wirtualizacji. Wszystkie instrukcje procesorowe systemu gościa są interpretowane przez emulator, a następnie mapowane na rzeczywisty sprzęt fizyczny gospodarza. Dalej instrukcje są wykonywane i zwracane do emulatora, który przekazuje je do gościa. Z racji długiej drogi, jaką musi pokonać pojedyncza instrukcja, to rozwiązanie należy do najwolniejszych. Tę metodę stosuje się, gdy architektury sprzętowe gościa i gospodarza są zbyt różne, aby dało się je zmapować lub jest to nieopłacalnie trudne. Rozwiązanie to stosuje się na przykład do emulacji środowisk smartfonów, konsol do gier i mikrokontrolerów.

2.1.2 Poziom warstwy abstrakcji sprzętowej (*HAL*)

Inaczej nazywany poziomem sprzętowego wspomagania. Ta metoda wirtualizacji polega na mapowaniu rzeczywistych zasobów sprzętowych na wirtualne zasoby. Zatem wszystkie instrukcje nieuprzywilejowane są wykonywane bezpośrednio na sprzęcie gospodarza. Instrukcje uprzywilejowane, czyli takie, które może wykonać tylko gospodarz, są przekazywane do hipernadzorcy i ten je wykonuje. Rozwiązanie to jest zasadniczo szybsze od wirtualizacji zestawu instrukcji. Z tej metody korzystają narzędzia do wirtualizacji takie jak *VMware Workstation*[14], *Oracle VirtualBox*[15] i wiele innych.

2.1.3 Poziom systemu operacyjnego

Wirtualizacja na poziomie systemu operacyjnego pozwala na działanie kilku odrębnych odizolowanych od siebie systemów operacyjnych, a dokładnie przestrzeni użytkownika korzystających z jednego jądra systemowego. Zatem nie dochodzi do duplikacji całych systemów operacyjnych, a co za tym idzie nie jest wymagana praca hipernadzorcy. Rozwiązanie takie jest nazywane konteneryzacją i jest stosowane na takich platformach jak Docker, *CoreOS*[16], *LXC*[2] i *Oracle Solaris Containers*[17]. Wadą tej metody jest przymus używania tylko takich systemów operacyjnych, które mogą współdzielić jądro z systemem gospodarza. Nic jednak nie stoi na przeszkodzie, aby umieścić kontenery na maszynie wirtualnej postawionej na poziomie wspierania sprzętowego, jednak spowolni to szybkość tych kontenerów do szybkości samej maszyny wirtualnej.

2.1.4 Poziom bibliotek lub języków programowania

Poziom ten jest stosowany w ramach technologii *JIT* (*just in time*). Niektóre środowiska programistyczne pozwalają na kompilację kodu źródłowego do postaci kodu pośredniego, który nie ma bezpośredniego przełożenia na kod maszynowy. Program jest wykonywany na maszynie wirtualnej, która kompiluje kod pośredni do kodu maszynowego, a może to zrobić na początku wywołania programu lub pierwszego wywołania konkretnej części kodu. Z tej technologii korzysta *Java*[5], *.Net*[18], *Python*[10] i wiele innych języków programowania.

2.1.5 Poziom aplikacji

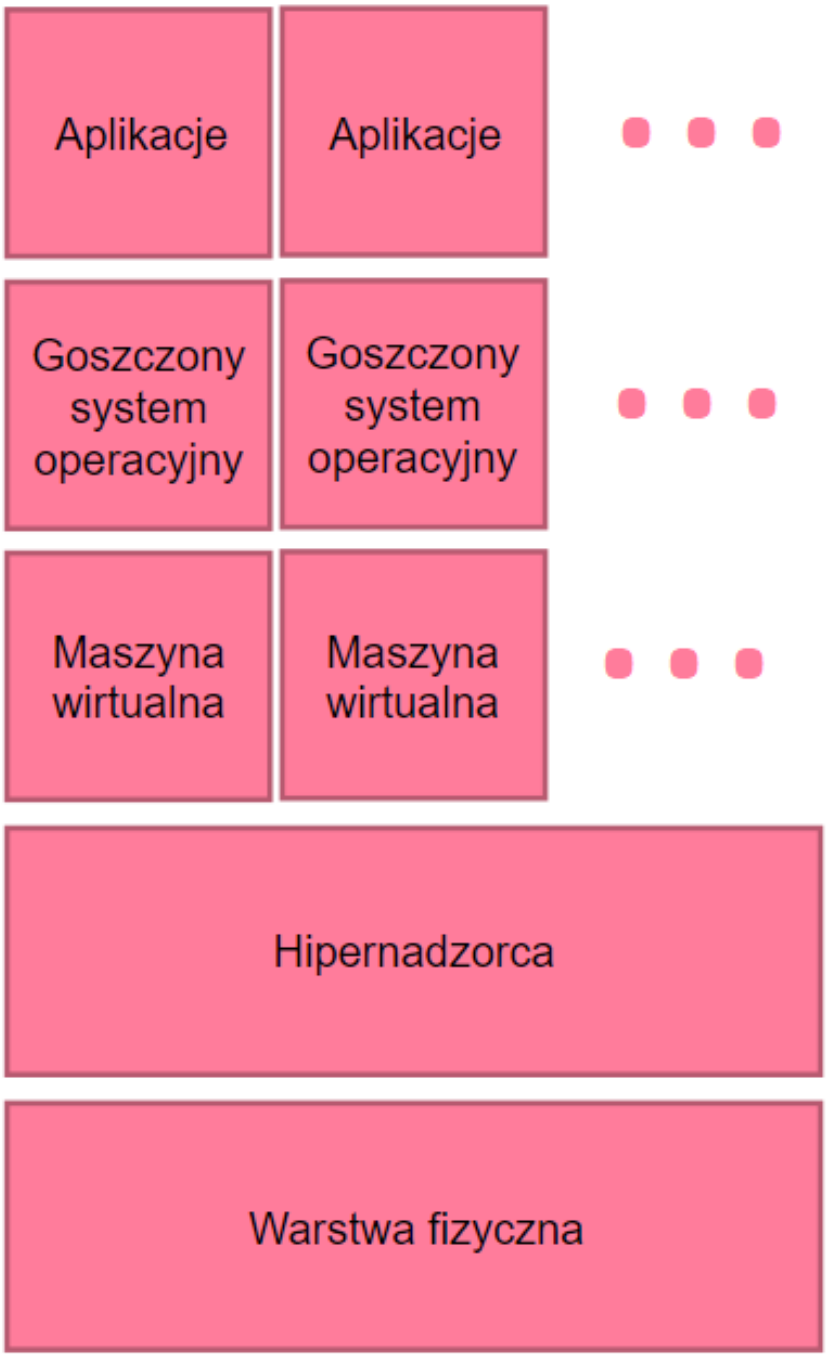
Wirtualizowane są tylko aplikacje, co oznacza, że nie wykorzystują one bezpośrednio dostarczanego przez system operacyjny środowiska wykonawczego, a używają dostarczonego przez inną aplikację takie środowisko, na jakie zostały zbudowane. Przykładem platformy umożliwiającej wirtualizację aplikacji jest *Wine*[19], która umożliwia działanie aplikacji przystosowanych do pracy w systemie *Microsoft Windows*[20] na systemie operacyjnym typu Linux.

2.2 Hipernadzorca

Komponentem systemu komputerowego umożliwiającym dokonanie wirtualizacji na poziomie wsparcia sprzętowego jest hipernadzorca (*hypervisor*). Rozróżniane są dwa typy hipernadzorców:

2.2.1 Hipernadzorca typu pierwszego

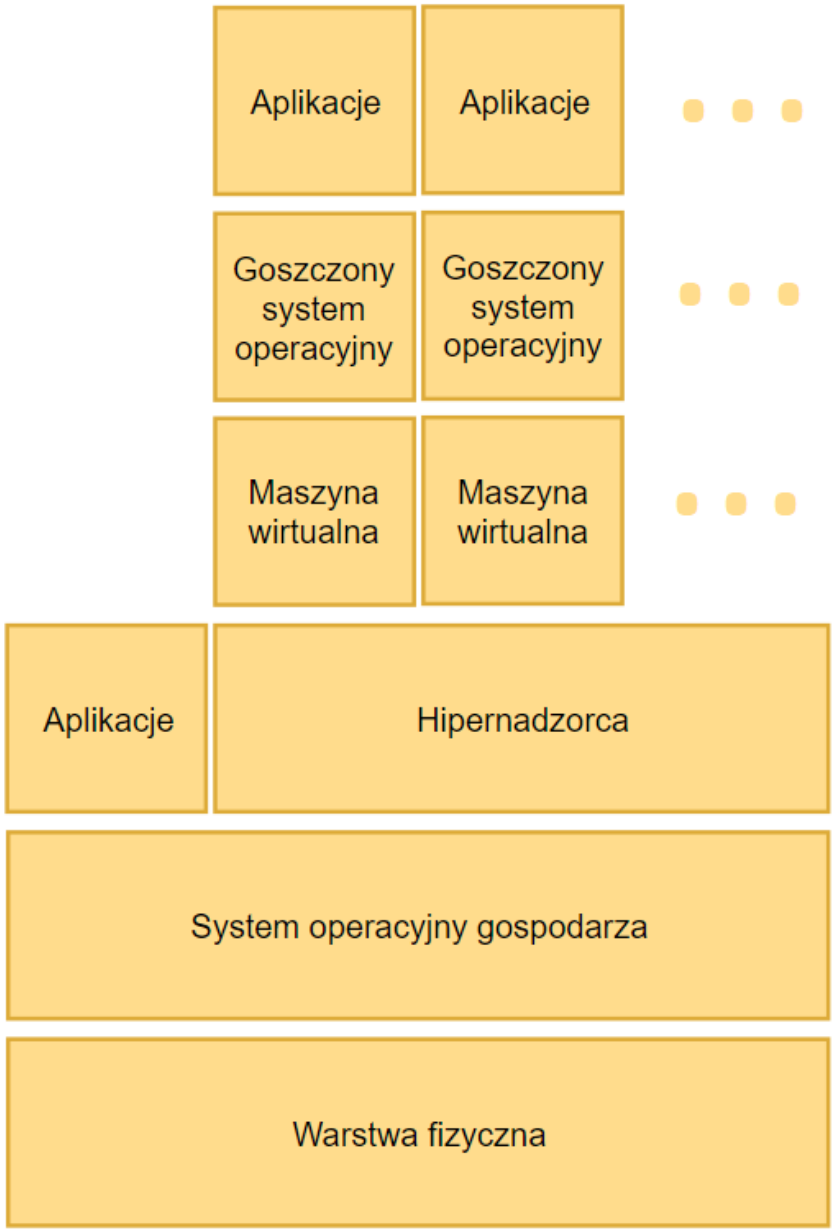
Jest to tak zwany hipernadzorca natywny (*bare metal*) osadzony bezpośrednio na sprzęcie, nie wymaga on zainstalowanego systemu operacyjnego gospodarza. Pozwala on osadzać maszyny wirtualne nad sobą, co znacząco upraszcza schemat abstrakcji architektury całego systemu. Przykładami tego typu hipernadzorców są *Microsoft Hyper-V*[21] i *VMware ESXi*[3].



Rysunek 2.2: Schemat wirtualizacji dla hipernadzorcy typu pierwszego

2.2.2 Hipernadzorca typu drugiego

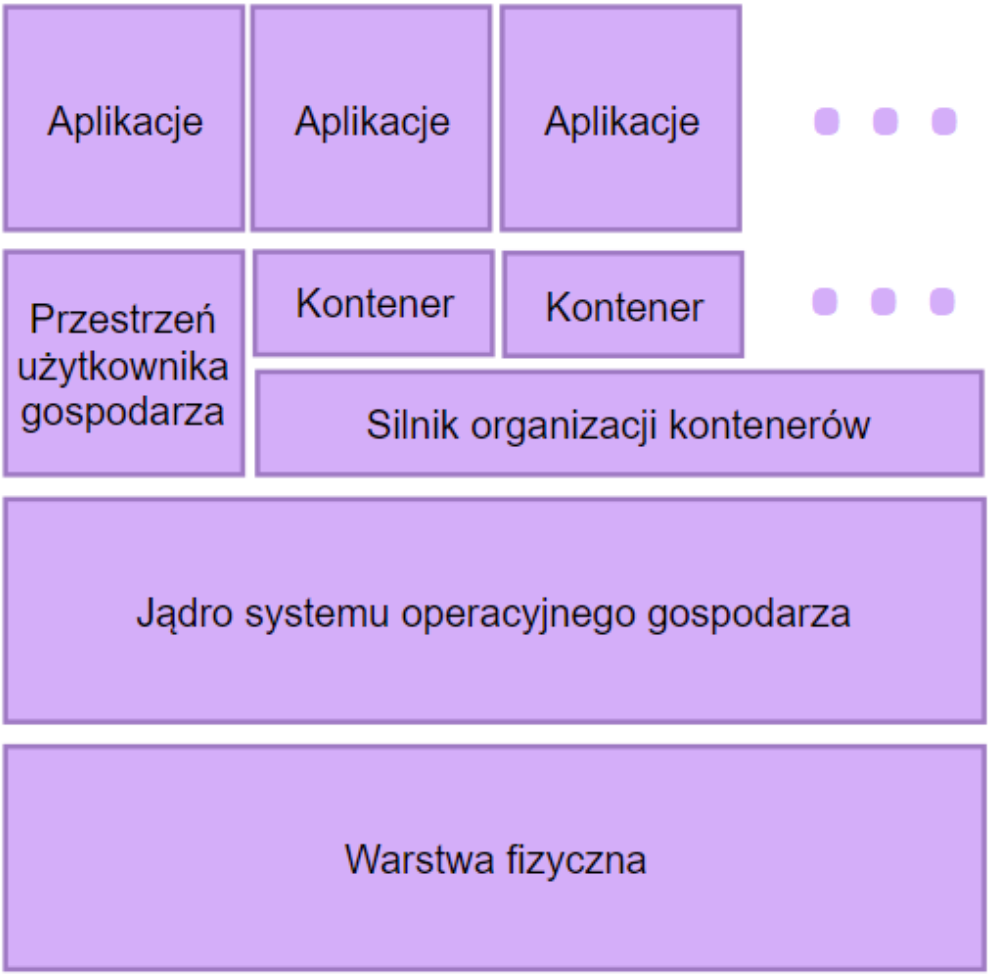
Hipernadzorca hostowany jest osadzony w systemie operacyjnym gospodarza jako aplikacja. Takimi aplikacjami są przykładowo *VMware Workstation*[14], *VMware Player*[22], *Oracle VirtualBox*[15] i *QEMU*[23].



Rysunek 2.3: Schemat wirtualizacji dla hipernadzorcy typu drugiego

2.3 Konteneryzacja

Konteneryzacja jako wirtualizacja na poziomie systemu operacyjnego, z racji współdzielenia przez kontenery jądra systemu gospodarza, powinna cechować się większą wydajnością od wirtualizacji na poziomie *ISA* lub *HAL*. O ile nie zastąpi ona klasycznych maszyn wirtualnych pod względem możliwości uruchamiania różnorodnych systemów operacyjnych, o tyle najmocniejszą stroną konteneryzacji jest dużo niższe zapotrzebowanie na pamięć operacyjną, co za tym idzie przenaszalność oraz szybkość zarówno działania, jak i startowania.



Rysunek 2.4: Schemat konteneryzacji

Współdzielenie jądra systemowego oraz dostępu do fizycznej warstwy komputera niesie ze sobą pewne ryzyko, jakim jest drastyczny spadek wydajności. Wszystkie kontenery konkurują ze sobą o zasoby sprzętowe, ale również o zasoby jądra, które nie jest odizolowane między nimi. Zespół Sysdig napotkał problem degradacji szybkości działania środowiska wewnątrz kontenerów. Z artykułu autorstwa Gianluca Borello[24] wynika, że jeden kontener spowalniał inny, poprzez wielokrotne wyszukiwania plików o zadanych ścieżkach dostępu. Proces rozwikłania ścieżek dostępu jet procesem nietrywialnym i czasochłonnym, jądro *Linuxa* zapisuje w związku z tym dane, jakie uzyskał w tablicy haszowanej, również próby nieudane, by w przyszłości nie powtarzać całego procesu. Ma to na celu szybsze rozwikłania ścieżek. Jeżeli jednak dojdzie do nadmiernego rozrostu tablicy, to efekt staje się odwrotny do zamierzonego i jądro działa wolniej, tym samym spowalniając uruchomione w każdej przestrzeni użytkownika aplikacje. Jądro systemu jest bardzo skomplikowanym oprogramowaniem i jak każde oprogramowanie może w pewnych warunkach nie działać optymalnie, jak w wymienionym przypadku i innych tworząc wąskie gardło pomiędzy warstwami

użytkownika i warstwą fizyczną.

Rozdział 3

Założenia projektowe

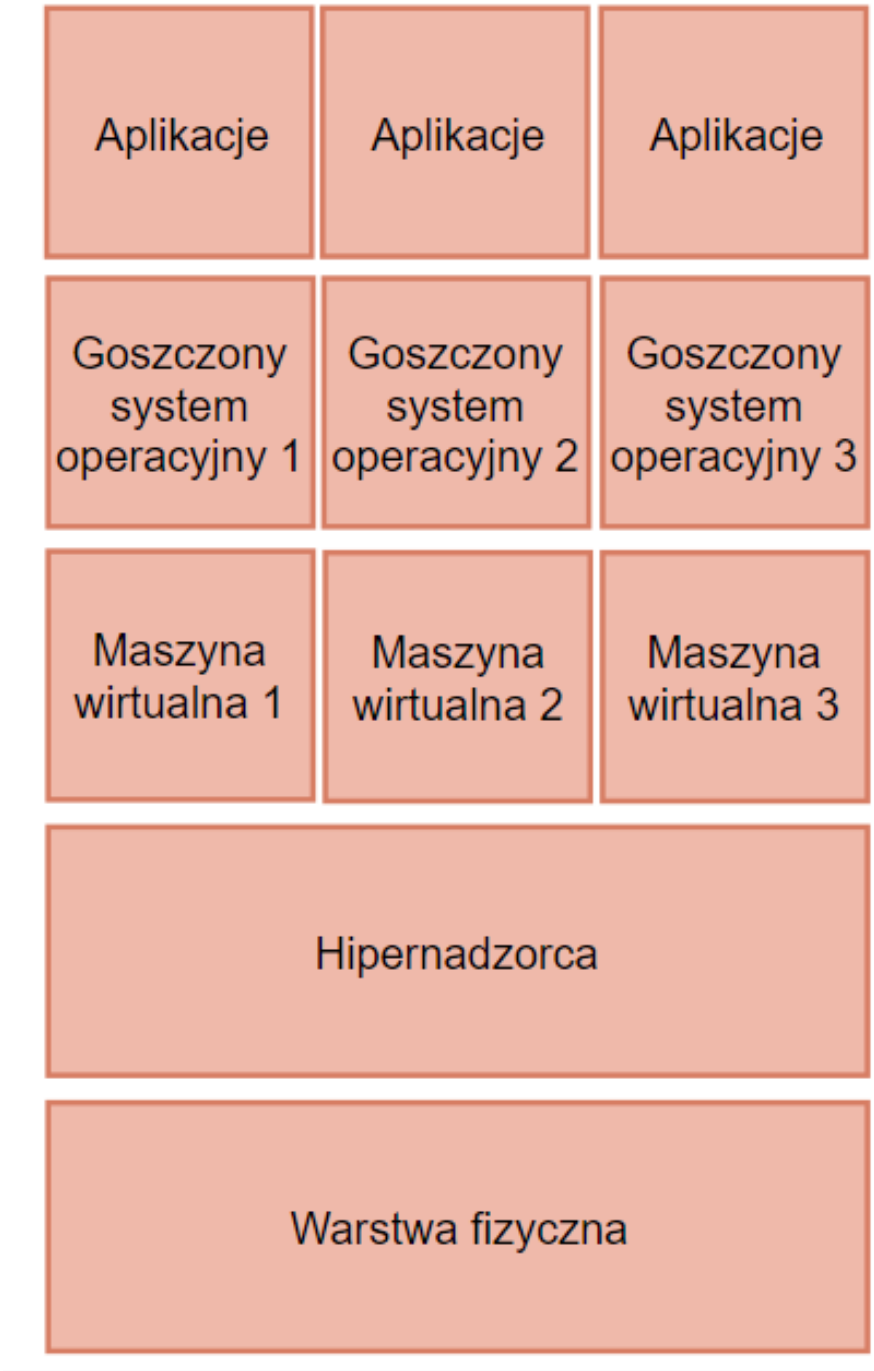
3.1 Architektura wejściowa serwisów

3.1.1 Platforma systemu serwerowego

Na wydziałowym serwerze zainstalowany jest hipernadzorca typu pierwszego, dokładnie *VMware ESX 6.5*[3], osadzony bezpośrednio nad warstwą fizyczną klastra komputerowego. Na serwerze wdrożonych jest szereg aplikacji działających na odrębnych maszynach wirtualnych.

3.1.2 Wizualizacja platformy wejściowej

Na rysunku 3.1 znajduje się schemat wejściowej architektury systemu serwerowego, który w ramach tej pracy ma zostać przekształcony.

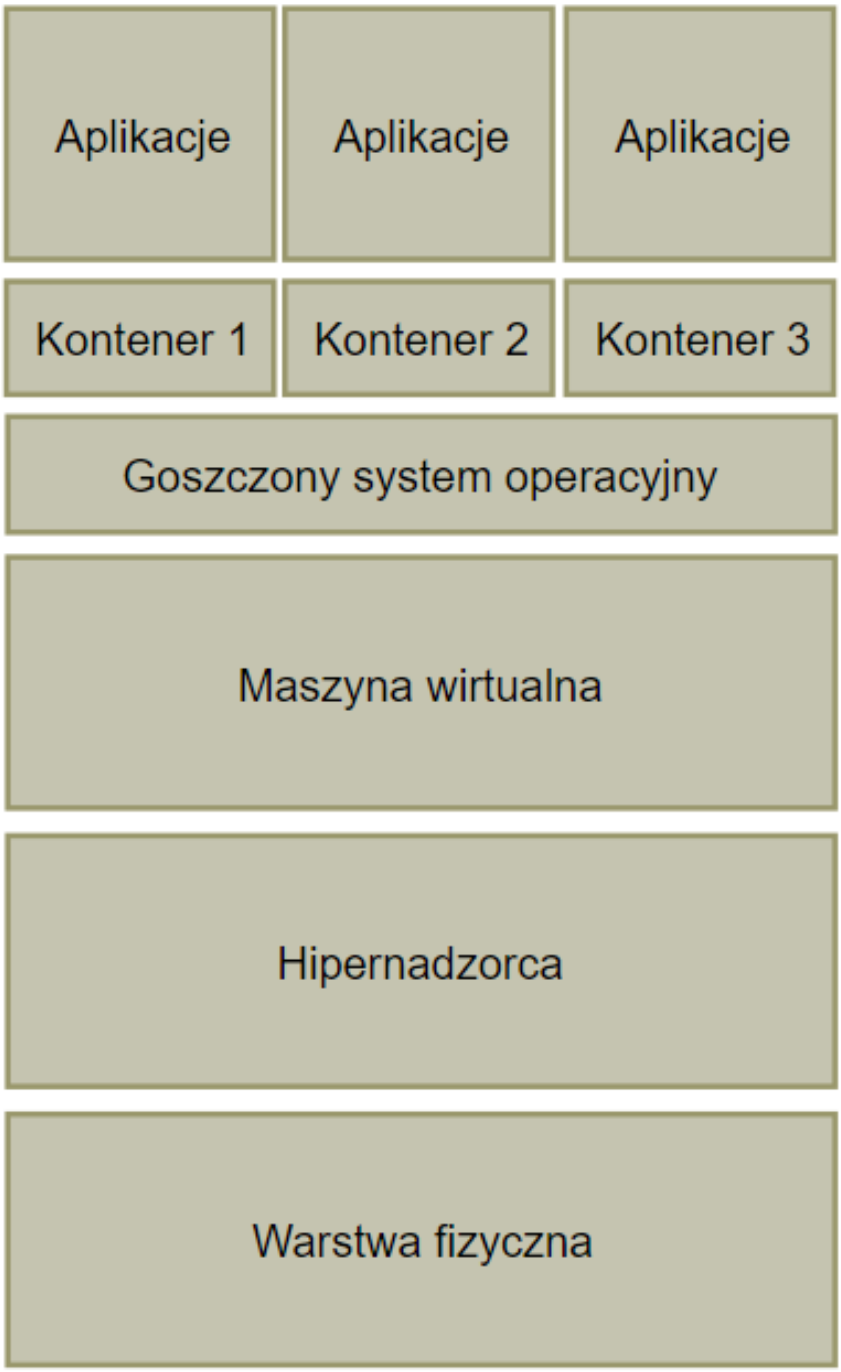


Rysunek 3.1: Schemat architektury wejściowej

3.2 Architektura docelowa serwisów

Koncepcja architektury docelowej polega na zlikwidowaniu pewnej ilości maszyn wirtualnych i przeniesieniu serwisów w nich zawartych do kontenerów znajdujących się w jednej maszynie wirtualnej. Oczywistą korzyścią z tej akcji jest zaoszczędzenie pewnej, być może nawet dużej ilości pamięci sprzętowej. Nie jest jednak oczywiste, do jakiej zmiany dojdzie w kwestii wydajności. O ile pierwotny układ składa się z

jednego poziomu wirtualizacji, tak ten układ będzie się składał z dwóch: wirtualizacji wspomaganej sprzętowo oraz na poziomie systemu operacyjnego.



Rysunek 3.2: Schemat architektury docelowej

Rozdział 4

Opis przykładowych aplikacji

4.1 Opis

Przykładowe aplikacje to najprostsze aplikacje o identycznych wymaganiach jak aplikacje pochodzące z wydziałowego serwera. Aplikacja *Java Spring Boot* jest serwisem *Http* prezentującym zawartość w przeglądarce internetowej. Serwis *Apache-Php* również jest serwisem *Http*, który udostępnia przez przeglądarkę interfejs *CRUD* do trzeciego serwisu, który dostarcza bazę danych PostgreSQL.

4.2 Środowisko projektowe i wymagania aplikacji

Środowisko projektowe:

- Narzędzie do wirtualizcji *VMware Workstation Player 15.5*[22]
- System operacyjny *Debian 10.2*[8]
- platforma *Docker 19.03.5*[1]
- narzędzie do zarządzania kontenerami *Docker-compose 1.25.0*[25]

Wymagania Aplikacji:

- Aplikacja *Java Spring Boot*
  - ⋆ *Java 8*[5] lub nowsza
- Aplikacja *Php*
  - ⋆ *Php 5.4*[4] lub nowsze z dodatkiem umożliwiającym połączenie z bazą danych PostgreSQL
  - ⋆ *Apache 2*[26] lub nowszy
  - ⋆ Dostęp do istniejącej bazy danych *PostgreSQL*[7]

4.3 Działanie i powiązania zbioru kontenerów

Docker

Platforma Docker do utworzenia kontenera wymaga dwóch komponentów: zbudowanej aplikacji i jej zasobów w postaci na przykład plików *Html*, skryptów itd. oraz pliku konfiguracyjnego *dockerfile*, na podstawie którego zostanie utworzona przestrzeń użytkownika w kontenerze. Wykorzystanymi komendami *dockerfile* w projekcie są:

- *RUN* wykonuje polecenie powłoki systemu operacyjnego w kontenerze podczas jego budowy
- *CMD* nadpisuje domyślną instrukcję *command*, która domyślnie uruchamia aplikacje wewnątrz kontenera. Dotyczy to przypadku, w którym kontener pracuje w trybie odłączonym od powłoki systemu gospodarza. Instrukcja *CMD* może zostać wykorzystana raz, każda kolejna nadpisze poprzednią, w konsekwencji wykonana zostanie tylko ostatnia.
- *COPY* kopiuje pod podane miejsce w systemie plików gościa wskazane pliki i katalogi z systemu plików gospodarza
- *ADD* jest to rozszerzona instrukcja *COPY*, pozwala również na dodanie plików z pod podanego adresu *Url* oraz automatycznie rozpakować foldery skompresowane w trakcie kopiowania
- *ENTRYPOINT* wykonuje tę samą czynność co *CMD* z tą różnicą, że może być użyta wielokrotnie i każde użycie zostanie wykonane. Instrukcje *ENTRYPOINT* i *CMD* nie kolidują ze sobą
- *VOLUME* tworzy obszar w pamięci masowej gospodarza, w którym będzie przechowywana kopia danych wygenerowanych przez kontener, np. logi, bazy danych itd. Jeżeli zostanie podana lokacja od strony gospodarza dane znajdujące się tam dane trafią do kontenera.
- *FROM* instrukcja, której obecność jest zawsze wymagana do zbudowania obrazu. Musi znajdować się w pierwszej linii pliku dockerfile. Jej argumentem jest tag obrazu, który ma zostać zbudowny. W trakcie budowy obraz jest pobierany z repozytorium.
- *ENV* tworzy zmienną środowiskową w goszczonym systemie operacyjnym
- *EXPOSE* pokazuje port widoczny dla maszyn zewnętrznych

Z racji tego, iż wybranym system operacyjnym gospodarza dla kontenerów jest *Debian 10*[8], *dockerfile* musi zawierać komendy, które zainstalują wymagane do działania aplikacji paczki i biblioteki oraz koniecznie musi korzystać z obrazów systemów z

rodziny *Linux*.

Docker-compose

W przypadku, kiedy w systemie pracuje wiele kontenerów, pojawia się problem zarządzania nimi. Uruchamianie każdego kontenera z wiersza poleceń jest żmudnym i podatnym na pomyłki procesem. Zastosowanie skryptu np. w powłoce *bash* cechowałoby się małą przejrzystością. Programem służącym do tego celu jest *Docker-compose*. Pozwala on budowanie i uruchamianie wielu kontenerów docker jednocześnie. Tak jak *Docker*[1], *Docker-compose*[25] wymaga pliku konfiguracyjnego, tym razem w notacji *YAML* i o rozszerzeniu *docker-compose.yml*. W tym pliku są zawarte lokacje plików dockerfile i parametry służące do budowy i uruchomienia obrazu oraz konfiguracja sterowników np. sieciowych.

Aplikacja Java Spring Boot

Realizuje proste zadanie, tworzy serwis internetowy i wyświetla przykładowy napis w oknie przeglądarki.

Aplikacja Php

Zadaniem tej aplikacji jest zainicjalizowanie bazy danych i udostępnienie za pośrednictwem strony *www* interfejsu *CRUD* do tej bazy.

Baza danych PostgreSQL

Baza danych musi istnieć i być dostępna dla aplikacji *PHP*.

4.4 Zrealizowane pliki dockerfile

4.4.1 Aplikacja Java Spring Boot

```
1 FROM openjdk:8-jdk-alpine
2 VOLUME /tmp
3 ARG JAR_FILE=target/*.jar
4 COPY ${JAR_FILE} app.jar
5 ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

Rysunek 4.1: Plik dockerfile dla aplikacji Java Spring Boot

Na rysunku 4.1 widać, że kontener jest budowany na podstawie obrazu zawierającego środowisko deweloperskie *Java 8*[5]. W linii drugiej tworzony jest, wolumin, który

domyślnie znajduje się w tym samym katalogu co *dockerfile* i ten katalog będzie zawierać bieżącą zawartość folderu `/tmp` w kontenerze.

4.4.2 Aplikacja Php

```
1 FROM php:7.0-apache
2 MAINTAINER Bartlomiej Mucha <5mucha1@fis.agh.edu.pl>
3
4 # Install apache, PHP, and supplimentary programs. openssh-server, curl, and lynx-cur are for debugging the container.
5 RUN apt-get update \
6   && apt-get install -y libpq-dev wget gnupg\
7   && docker-php-ext-install pgsql \
8   && apt-get clean \
9   && rm -rf /var/lib/apt/lists/*
10
11 RUN apt-get update && apt-get -y install \
12   lsb-release \
13 && rm -rf /var/lib/apt/lists/*
14
15 RUN wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key add -
16 RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ $(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list
17
18 RUN apt-get update && apt-get -y install \
19   postgresql-client-9.6 \
20 && rm -rf /var/lib/apt/lists/*
21 # Manually set up the apache environment variables
22 ENV APACHE_RUN_USER www-data
23 ENV APACHE_RUN_GROUP www-data
24 ENV APACHE_LOG_DIR /var/log/apache2
25 ENV APACHE_LOCK_DIR /var/lock/apache2
26 ENV APACHE_PID_FILE /var/run/apache2.pid
27
28 # Expose apache.
29 EXPOSE 80
30
31 # Copy this repo into place.
32 ADD ProjektBD_KD /var/www/site
33
34 # Update the default apache site with the config we created.
35 ADD apache-config.conf /etc/apache2/sites-enabled/000-default.conf
36 # RUN a2enmod rewrite
37
38 CMD /var/www/site/run.sh
```

Rysunek 4.2: Plik dockerfile dla aplikacji Php

W pliku widniejącym na rysunku 4.2 między liniami 5. i 20. pobierane są potrzebne do funkcjonowania biblioteki. W kolejnej części ustawiane są zmienne globalne dla środowiska *Apache*[26] oraz w 35. linii kopiowany plik konfiguracyjny. Aplikacja *Php* przed uruchomieniem wymaga dostępu do bazy danych *PostgreSQL*[7] znajdującej się w innym kontenerze. Stąd na końcu nadpisana zostaje domyślna komenda. Skrypt `run.sh` ma za zadanie zawiesić proces uruchamiania aplikacji do czasu, aż nawiąże kontakt z bazą danych. Bardziej eleganckie rozwiązanie na ten problem dostarcza *Docker-compose*[25]. Na rysunku 4.3 można zaobserwować dyrektywę *depends_on* w serwisie *apache* (nazywany w pracy Serwisem Php). Dyrektywa ta wymusza kolejność startowania kontenerów, to znaczy, kiedy kontener jest zależny od innego, wystartuje dopiero wtedy, kiedy kontener od którego zleży zgłosi gotowość. Wadą tego rozwiązania jest fakt, że mechanizm dyrektywy *depends_on* sprawdza tylko, czy główny wątek kontenera jest już uruchomiony, a nie czy aplikacja, w tym przypadku

baza danych, jest gotowa do działania.

4.4.3 Docker-compose

```yaml
version: '3'
services:
    postgresdb:
        image: postgres
        container_name: postgresdb
        ports:
            - "5432:5432"
        restart: always
        environment:
            - POSTGRES_PASSWORD=password
            - POSTGRES_USER=user
            - POSTGRES_DB=kddb
    apache:
        build: ./php/app
        container_name: php-apache
        volumes:
            - ./php/app/ProjektBD_KD:/var/www/site
        ports:
            - "8081:80"
        environment:
            - POSTGRES_HOST=postgresdb
        depends_on:
            - postgresdb
    springboot:
        container_name: springboot
        build: ./springboot/gs-spring-boot-docker/complete
        ports:
            - "8080:8080"
networks:
  default:
    driver: bridge
```

Rysunek 4.3: Plik docker-compose.yml

Rozdział 5

Wdrożenie przykładowych oraz rzeczywistych serwisów wydziałowego systemu informatycznego

5.1 Wdrożenie przykładowych aplikacji

Wdrożenie utworzonego systemu w ramach tej pracy polega na imporcie maszyny wirtualnej zawierającej gotowe środowisko i aplikacje do środowiska VMware ESX na wydziałowym systemie serwerowym. Po rozwiązaniu kliku pomniejszych problemów z konfiguracją dostarczonej maszyny wirtualnej udało się ostatecznie ją zainstalować i uruchomić. Zbudowane ówcześnie kontenery zostały wzbudzone komendą: `docker-compose up`. Operacja zakończyła się pełnym sukcesem, serwisy zadziałały poprawnie i były widoczne dla wszystkich urządzeń sieci wydziałowej.

5.2 Wdrożenie rzeczywistych serwisów wydziałowego systemu informatycznego

Wdrożenie serwisów pochodzących z wydziałowego systemu informatycznego polega na zamienieniu zbudowanych plików wykonywalnych oraz plików interpretowanych w odpowiednich miejscach drzewa katalogowego z naniesieniem ewentualnych zmian w plikach budujących obrazy kontenerów i konfiguracji *docker-compose.yml*.

Rozdział 6

Wnioski

Zastępowanie pojedynczej wirtualizacji na poziomie wspomagania sprzętowego na wirtualizację zagnieżdżoną, bo z dodatkową wirtualizacją na poziomie systemu operacyjnego, mogłoby wzbudzić kontrowersje co do słuszności takiego pomysłu. Należy sobie jednak zdać sprawę z tego, że obrazy systemów operacyjnych zajmują pewną ilość miejsca na dysku, zatem pierwszą zaletą zaprezentowanego rozwiązania jest oszczędność na pamięci masowej. Kolejną zaletą jest ograniczenie liczby maszyn wirtualnych, które musi obsłużyć hipernadzorca. Tym samym jest on odciążony, co daje szanse na usprawnienie pracy pozostałych maszyn wirtualnych oraz serwisów w nich zawartych. Z punktu widzenia administratora serwera redukcja maszyn wirtualnych, np. mało znaczących bądź zawierających małe serwisy o dużo niższym zapotrzebowaniu na zasoby sprzętowe niż system operacyjny, na którym funkcjonują, jest z pewnością oszczędnością zasobów sprzętowych oraz ułatwieniem w zarządzeniu infrastrukturą serwera, w szczególności, jeżeli liczba osadzonych w systemie maszyn wirtualnych jest duża.

Rozdział 7

Kod źródłowy

7.1 Repozytorium

Repozytorium zawierające infrastrukturę projektu wraz z kodami źródłowymi znajduje się pod adresem:
`https://github.com/Shadoba/PracaInz`
Maszyna wirtualna wraz z zawartością powyższego odnośnika jest dostępna pod adresem:
`https://drive.google.com/open?id=1oln7lEP7vkTKzK4uAyGMQs2OY97nPn_4`

7.2 Dołączona płyta CD

Dołączona płyta CD zawiera:

- Katalog *Praca inzynierska* w którym znajduje się niniejsza praca dyplomowa w formacie *Pdf* wraz z plikami źródłowymi potrzebnymi do jej wygenerowania w środowisku LaTeX.
- Katalog *Projekt* zawiera kopie repozytorium.

7.3 Kod osób trzecich

Aplikacja *Java Spring Boot* została zbudowana z kodu pochodzącego z oficjalnego samouczka[6] na licencji ASLv2

Bibliografia

[1] Docker, Inc., Docker. https://www.docker.com/. Accessed: 2020-01-14.

[2] Lxc. https://linuxcontainers.org/. Accessed: 2020-01-14.

[3] VMware, Inc., VMware ESX. https://www.vmware.com/products/esxi-and-esx.html. Accessed: 2020-01-14.

[4] PHP Group, PHP. https://www.php.net/. Accessed: 2020-01-14.

[5] Oracle, Java. https://www.java.com/. Accessed: 2020-01-14.

[6] Pivotal Software, Inc., Spring Boot framework. https://spring.io/. Accessed: 2020-01-14.

[7] The PostgreSQL Global Development Group, PostgreSQL. https://www.postgresql.org/. Accessed: 2020-01-14.

[8] Software in the Public Interest, Inc., Debian 10. https://www.debian.org/index.pl.html. Accessed: 2020-01-14.

[9] The Apache Software Foundation, Apache Tomcat. http://http://tomcat.apache.org/. Accessed: 2020-01-14.

[10] Python Software Foundation, Python. https://www.python.org/. Accessed: 2020-01-14.

[11] Django Software Foundation, Django. https://www.djangoproject.com/. Accessed: 2020-01-14.

[12] Wordpress. https://wordpress.org/. Accessed: 2020-01-14.

[13] Oracle, MySQL. https://www.mysql.com/. Accessed: 2020-01-14.

[14] VMware, Inc., VMware Workstation Pro. https://www.vmware.com/products/workstation-pro.html. Accessed: 2020-01-14.

[15] Oracle, VirtualBox. `https://www.virtualbox.org/`. Accessed: 2020-01-14.

[16] Red Hat, Inc., CoreOS. `http://http://coreos.com/`. Accessed: 2020-01-14.

[17] Oracle, Oracle Solaris Containers. `https://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html`. Accessed: 2020-01-14.

[18] Microsoft, .Net. `https://docs.microsoft.com/pl-pl/dotnet/`. Accessed: 2020-01-14.

[19] Open-source wine project. `https://www.winehq.org/`. Accessed: 2020-01-14.

[20] Microsoft, Windows. `https://www.microsoft.com/pl-pl/windows`. Accessed: 2020-01-14.

[21] Microsoft, hyper-v. `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/`. Accessed: 2020-01-14.

[22] VMware, Inc., VMware Workstation Player. `https://www.vmware.com/products/workstation-player.html`. Accessed: 2020-01-14.

[23] QEMU. `https://www.qemu.org/`. Accessed: 2020-01-14.

[24] Gianluca Borello. *Container isolation gone wrong*. Sysdig, Inc., 2017. `https://sysdig.com/blog/container-isolation-gone-wrong/`. Accessed: 2020-01-14.

[25] Docker, Inc., Docker-compose. `https://docs.docker.com/compose/`. Accessed: 2020-01-14.

[26] The Apache Software Foundation, Apache HTTP Server Project. `https://httpd.apache.org/`. Accessed: 2020-01-14.