Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”

Politechnika Wrocławska

Wydział Informatyki i Zarządzania

Zaawansowane Technologie Webowe

Laboratorium

Temat:	Projekt grupowy – Konteneryzacja web aplikacji
Opracował:	mgr inż. Piotr Jóźwiak
Data:	lipiec 2020
Liczba godzin:	2 godziny

Table of Contents

Wstęp

Podczas implementacji aplikacji często pojawia się problem z czynnością przekazania projektu innemu programiście. Tym innym programistą może być zarówno kolejny deweloper, tester czy po prostu wdrożeniowiec na produkcję (DevOps). Odkąd powstały narzędzia ułatwiające zarządzanie zależnościami w projekcie takie jak Composer, Maven czy też Ant, sam program przestał być głównym źródłem problemów. W to miejsce jednak weszły jednak inne problemy, jak środowisko uruchomieniowe oraz jego konfiguracja – system operacyjny, odpowiedni interpreter (w odpowiedniej wersji), baza danych czy też inne usługi. Oznacza to, iż przekazując nasz kod innemu programiście, nie zawsze możemy być pewni, iż będzie on działał w sposób identyczny jak na naszym komputerze. Czasem, mimo nawet dokładnej analizy porównawczej środowisk, brak jednej biblioteki może zdecydowanie opóźnić lub też uniemożliwić szybkie rozpoczęcie pracy w nowym projekcie.

Opisany powyżej problem jest szczególnie dotkliwy w sytuacji, gdy okazuje się, że aplikacja poprawnie uruchamia się w środowiskach deweloperskich i testowych, a z jakiś przyczyn nie chce wystartować w produkcji.

Rozwiązanie tego problemu może być konteneryzacja aplikacji webowych. Do tego celu wykorzystać można Docekra. Czym zatem jest Docker? Jest to technologia umożliwiająca zapakowanie aplikacji wraz ze środowiskiem uruchomieniowym w postać kontenera. Zaletą takiego podejścia jest to, że dystrybucja oprogramowania nie polega na przesłaniu jedynie źródeł/binariów aplikacji, ale całości skonfigurowanej wraz z systemem operacyjnym. Od tej pory konfiguracja będzie działać zawsze i to niezależnie od tego, gdzie zostanie uruchomiona. Kontener Dockera zapewni nam dostarczenie odpowiedniej wirtualizacji systemu operacyjnego oraz zapisanej konfiguracji. To podejście jest codziennością i podstawą działania aplikacji w oparciu o mikro serwisy funkcjonujące pod dużym obciążeniem. Scenariusz jest bardzo prosty. Gdy pojawia się zwiększony ruch do aplikacji webowej, to *load balancer* może podjąć decyzję o uruchomieniu dodatkowej instancji aplikacji webowej celem zwiększenia chwilowej przepustowości. Na takim założeniu pracuje dzisiaj środowisko Kubernetes.

Celem dzisiejszych zajęć będzie przyjrzenie się sposobowi przygotowania kontenera z aplikacją webową nad którą pracujecie w grupach.

Oczywiście każdy zespół posiada różny stos technologiczny, ale w przykładzie, który zostanie omówiony, chodzi o zrozumienie pewnej idei oraz sposobu działania tego rozwiązania. Jeśli opanuje się te podstawowe rozumowanie koncepcji konteneryzacji web aplikacji, to wykonanie poniższych kroków dostosowane do wybranych technologii nie będzie stanowiło większego problemu.

Wymagania

Do wykonania poniższych przykładów musimy zainstalować dodatkowe oprogramowanie Docker'a, którego proces instalacji został opisany na tej stronie: https://docs.docker.com/engine/install/. Aby upewnić się, że środowisko jest gotowe należy wykonać poniższe polecenie:

```
λ docker -v
Docker version 19.03.12, build 48a66213fe
```

Jeśli w wyniku powyższego polecenia otrzymaliśmy wersję zainstalowanego Dockera, to oznacza, że środowisko jest gotowe do pracy z kontenerami.

Przykładowa aplikacja webowa

Zanim zaczniemy opisywać proces tworzenia kontenera z aplikacją webową, musimy najpierw przygotować samą aplikację. W naszym przypadku posłużymy się aplikacją wygenerowaną przez generator Express, z tej racji, że nie proces tworzenia aplikacji jest dla nas istotny, lecz sam proces konteneryzacji.

Dlatego całość przykładu będzie opierać się o aplikację webową wygenerowaną w poniższy sposób.

Na początek instalujemy menadżerem pakietów express generatora:

```
npm install express-generator -g
```

Następnie przechodzimy do folderu, w którym chcemy utworzyć szkielet aplikacji webowej i wydajemy polecenie:

```
express mywebappname
```

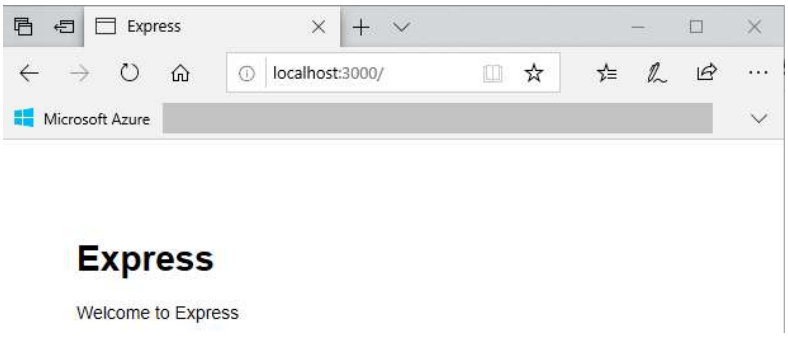
Powyższe polecenie utworzy aplikację webową o nazwie mywebappname w folderze o tej samej nazwie. Aby sprawdzić jej działanie musimy jeszcze zainstalować niezbędne zależności. Zatem wchodzimy do folderu z aplikacją i wydajemy poniższe polecenie:

```
cd mywebappname
npm install
```

Czas uruchomić aplikację, zatem można wykorzystać poniższe polecenie:

```
SET DEBUG=mywebappname:* & npm start
```

Po chwili powinno pojawić się polecenie informujące o tym, że aplikacja jest dostępna pod adresem http://localhost:3000. Sprawdźmy to:



Zatem samo środowisko jest już gotowe.

Przygotowanie do konteneryzacji

Zanim rozpoczniemy proces konteneryzacji aplikacji, musimy zastanowić się nad składowymi naszego systemu. W najbardziej rozbudowanej wersji możemy wyróżnić trzy składowe systemu:

- Front-end
- Back-end
- Baza danych

Oczywiście to, jakie elementy w rzeczywistości wystąpią w danym systemie jest mocno powiązane z tym, co dany system implementuje oraz jakie technologie wykorzystuje. Bo przecież baza danych nie zawsze będzie potrzebna – np. w stronach statycznych jej nie ma. W innej sytuacji może być tak, że nie mamy rozdzielenia Front-endu od Back-endu. W każdym razie, zawsze na początku trzeba się zastanowić nad tym, co ma wejść w skład kontenera.

Dość oczywistym jest fakt, że chcielibyśmy zamknąć w kontener Fornt-end. Powstanie wówczas możliwość uruchomienia danego serwisu na wielu instancjach jednocześnie, a przez to możemy zacząć load balancować ruch.

Pojawia się pytanie, czy chcemy w kontenerze umieścić także bazę danych? To zależy od scenariusza użycia. Na pewno wstawienie bazy danych do kontenera ma swoje zalety, ale posiada także dwa duże ograniczenia. Pierwszym z nich jest problem z synchronizacją danych pomiędzy instancjami bazy danych w przypadku uruchomienia aplikacji w środowisku balancowanym. Kolejnym problemem może być przenaszalność danych w momencie wystąpienia awarii. Zatem jeśli jest to prosta strona z WordPress to można się pokusić o umieszczenie jej w jednym kontenerze z bazą danych oraz zadbanie o odpowiednio częste kopie zapasowe danych. W innych przypadkach bazę danych warto ustawić centralnie poza kontenerem.

Co w takim razie z Back-end'em – jeśli takowy występuje? Tutaj także podejście jest różne. W najprostszym ujęciu można zapakować go wraz z Fron-endem. Będzie to działało, ale może uniemożliwić nam uruchomienie balancera. Wszystko zależeć będzie od tego, jak została zaimplementowana nasza aplikacja. W innych sytuacjach Back-end warto zapakować do oddzielnego kontenera.

W naszym przykładzie posiadamy tylko Front-end, bez bazy danych oraz oddzielonego Back-endu, ale nie to nie zmienia nic w samym sposobie budowania obrazu Dockera.

Ważne jest jedynie to, aby ustalić co będzie niezbędne do uruchomienia naszej aplikacji. Na pewno będzie to system Linux, a do tego niezbędne środowisko uruchomieniowe, czyli serwer aplikacji oraz odpowiednie biblioteki i interpretery. W przypadku Node.Js możemy pójść na skróty i skorzystać z obrazu Dockerowego przygotowanego już z pełnym środowiskiem Node. Jest on dostępny tutaj: https://hub.docker.com/_/node. Należy sprawdzić, czy w naszym przypadku także nie znajdzie się gotowy obraz w repozytorium Dockera.

Definicja kontenera

Jesteśmy gotowi już na przygotowanie kontenera. W tym celu w folderze z naszą aplikacją webową tworzymy plik **`Dockerfile`**:

```
touch Dockerfile
```

`Dockerfile` jest plikiem z przepisem jak utworzyć obraz kontenera. To tutaj zapisujemy poszczególne polecenia jakie ma wykonać Docker, celem utworzenia obrazu.

Definicję rozpoczynamy od wskazania obrazu bazowego, na jakim ma zostać skompilowana nasza aplikacja. W naszym przypadku jest to już wcześniej wspomniany obraz Node z oficjalnego repozytorium dockera. W tym celu dodajemy wpis do pliku **`Dockerfile`**:

```
FROM node:10
```

Następnie tworzymy folder na źródła naszej aplikacji. W to miejsce wgramy wszystkie niezbędne pliki do uruchomienia serwisu:

```
# Create app directory
RUN mkdir -p /opt/app
```

Następnie zmieniamy folder roboczy na nowo utworzony folder:

```
WORKDIR /opt/app
```

W kolejnym kroku tworzymy użytkownika, pod którym będziemy uruchamiać naszą aplikację. Domyślnie wszystko uruchamia się na prawach użytkownika root.

```
# Add user app
RUN adduser --disabled-password app
```

Teraz przyszedł czas na wgranie plików źródłowych aplikacji do obrazu:

```
# Copy source files to WORKDIR
COPY . .
```

Musimy zmienić właściciela plików na użytkownika, który będzie uruchamiał naszą aplikację:

```
RUN chown -R app:app /opt/app
```

Teraz możemy określić, że dalszy kod będzie uruchamiał się z poziomu użytkownika app:

```
# Change execution to user app
USER app
```

Instalujemy wszystkie zależności na obraz poleceniem:

```
# Install all dependecies
RUN npm install
```

Dajemy dostęp do portu 3000 z kontenera:

```
# Allow access to port 3000 on conainer
EXPOSE 3000
```

Na koniec startujemy serwer aplikacji **npm**:

```
# Run command - start npm server
CMD [ "npm", "start" ]
```

Mamy już gotowy plik **Dockerfile**. Zanim skompilujemy obraz musimy zdefiniować jeszcze plik z opisem czego docker ma nie wgrywać do obrazu. W tym celu tworzymy plik o nazwie **.dockerignore** w tym samym miejscu co plik **Dockerfile** oraz wpisujemy do niego:

```
node_modules
npm-debug.log
```

W ten oto sposób nie będziemy wgrywali naszych lokalnych kopii bibliotek, gdyż zainstalują się one same w trakcie uruchamiania kontenera.

Budowanie obrazu kontenera

Jesteśmy już gotowi do zbudowania obrazu. W tym celu wydajemy polecenie będąc w folderze z plikiem Dockerfile:

```
docker build -t express-webapp .
```

Powyższe polecenie zbuduje obraz o nazwie express-webapp korzystając z pliku Dockerfile w aktualnej lokalizacji. Proces tworzenia kontenera wygląda następująco:

```
Sending build context to Docker daemon  46.08kB
Step 1/10 : FROM node:10
 ---> 1cc99e24ab2d
Step 2/10 : RUN mkdir -p /opt/app
 ---> Using cache
 ---> 17d3154b4351
Step 3/10 : WORKDIR /opt/app
 ---> Using cache
 ---> 87ae27888e60
Step 4/10 : RUN adduser --disabled-password app
 ---> Using cache
 ---> 3f01f12bcb67
Step 5/10 : COPY . .
 ---> 04bf0e967713
```

```
Step 6/10 : RUN chown -R app:app /opt/app
 ---> Running in 9bc1c57ab5c5
Removing intermediate container 9bc1c57ab5c5
 ---> 77974c1ac2ad
Step 7/10 : USER app
 ---> Running in c7c16600779a
Removing intermediate container c7c16600779a
 ---> 2b1f191cf6db
Step 8/10 : RUN npm install
 ---> Running in b9beb40315bc
added 100 packages from 139 contributors and audited 101 packages in 3.042s
found 4 vulnerabilities (3 low, 1 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
Removing intermediate container b9beb40315bc
 ---> be1dd0b8ac22
Step 9/10 : EXPOSE 3000
 ---> Running in b776c6659fa6
Removing intermediate container b776c6659fa6
 ---> 7bbba9b6e49d
Step 10/10 : CMD [ "npm", "start" ]
 ---> Running in 550f26a2745e
Removing intermediate container 550f26a2745e
 ---> f54d14db144a
Successfully built f54d14db144a
Successfully tagged express-webapp:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-
Windows Docker host. All files and directories added to build context will have '
-rwxr-xr-
x' permissions. It is recommended to double check and reset permissions for sensi
tive files and directories.
```

Jeśli wszystko się udało, to wydając poniższe polecenie zobaczymy nowo utworzony obraz kontenera:

```
docker images
```

```
C:\Prv\OneDrive\Dokumenty\PWr\Dydaktyka\ZTW\ProjektPower\Lab14\exampleWebApp\helloworld>docker images
REPOSITORY                                  TAG                 IMAGE ID            CREATED             SIZE
express-webapp                              latest              f54d14db144a        15 minutes ago      920MB
```

Uruchomienie kontenera z aplikacją

Przyszedł czas na uruchomienie kontenera. W tym celu wykonujemy poniższe polecenie:

```
docker run -it -p 80:3000 express-webapp
```

Warto tutaj zwrócić uwagę na wartość przekazaną do parametru -p. Wartość 80:3000 oznacza mapowanie portu naszego hosta (w tym wypadku 80) na port kontenera (tutaj 3000). Jak pamiętamy w pliku Docker file, ustawiliśmy dostęp do tego portu odpowiednim poleceniem EXPOSE. Jak widać w trakcie uruchamiania mamy możliwość na dowolne przemapowanie portu wewnątrz kontenera na port pod jakim chcemy, aby aplikacja była widoczna na rzeczywistym hoście. Jak wiadomo port 80 jest domyślnym portem HTTP, toteż po wpisaniu adresu http://localhost powinna uruchomić się aplikacja uruchomiona z kontenera Docker.

Jak debugować

A co jeśli coś nie działa? To najprościej zalogować się do kontenera poprzez uruchomienie shella z wnętrza. Uzyskujemy dostęp do wewnętrznego systemu operacyjnego. Dalej możemy wykorzystać to do sprawdzania czy wszystko poprawnie się zainstalowało, sprawdzić co zawierają logi aplikacji, itp.

Aby uruchomić wewnętrzny shell wystarczy uruchomić polecenie:

```
docker run -it express-webapp /bin/bash
```

Otrzymujemy znak zachęty:

```
app@def3333eb3ad:/opt/app$
```