Docker

Laboratorium 2
2019

1. Wstęp

Zadaniem tego laboratorium jest przygotowanie plików *Dockerfile* oraz *docker-compose* dla solucji *Library*. Zawiera ona trzy serwisy, które po odpowiedniej konfiguracji powinny się ze sobą komunikować za pomocą różnych protokołów komunikacyjnych. Celem zajęć jest pokazanie, jak w łatwy sposób można zbudować gotowe środowisko dla wielu usług tworzących razem system.

2. Przygotowanie środowiska

Aby móc otworzyć i edytować solucję, należy zainstalować:

1. SDK .NET Core w wersji 2.2.106 (na stronie: https://dotnet.microsoft.com/download/dotnet-core/2.2);
2. Docker (Oficjalne źródło: https://www.docker.com/);

Do tworzenia i konfigurowania plików *Dockerfile* potrzebna będzie wiedza na temat komendy *dotnet publish*. Służy ona do kompilacji projektu i zapakowania plików wynikowych wraz z zależnościami do folderu. Folder ten można potem wydeployować na maszynie hosta. Dokładny opis i przykłady można znaleźć na poniższej stronie: https://docs.microsoft.com/pl-pl/dotnet/core/tools/dotnet-publish?tabs=netcore21.

1. Stworzenie Dockerfile'a

W systemie opartym na wielu usługach powinno się stosować takie podejście do wdrażania, w którym każda z nich może być uruchamiania oddzielnie. Idealnym narzędziem do tego zadania jest Docker. Każda aplikacja lub usługa powinna posiadać swój plik *Dockerfile*. Dobrą praktyką jest podzielenie go na dwa kroki:

1. pobieranie zależności i budowanie aplikacji na środowisku z narzędziami do budowania;
2. uruchomienie paczki z aplikacją na środowisku uruchomieniowym;

Do wszystkich kontenerów tworzonych na laboratorium najlepiej wykorzystać poniższe bazowe obrazy (wyjątkiem jest obraz dla kolejki RabbitMQ, której konfiguracja kontenera zostanie dostarczona wraz z projektem)

- `crosoft/dotnet:2.2-sdk` jako obraz do budowania;
- `microsoft/dotnet:2.2-aspnetcore-runtime` jako obraz do uruchamiania;

Dockerfile, który należy stworzyć dla projektu *Library.Web*, powinien mieć taką strukturę:

```
FROM microsoft/dotnet:2.2-sdk AS build-env
WORKDIR /app

 # Here: copy files, restore packages, build project

# Build runtime image
FROM microsoft/dotnet:2.2-aspnetcore-runtime
WORKDIR /app

# Here: copy built package from build-env to the runtime image

ENTRYPOINT ["dotnet", "Library.Web.dll"]
```

Rys. 1.1 Szablon Dockerfile'a

3. Stworzenie docker-compose'a

Compose to narzędzie służące do zdefiniowana środowiska opartego na wielu kontenerach dockerowych. Pozwala w jednym miejscu skonfigurować takie rzeczy jak:

- Sieć do komunikacji między kontenerami
- Mapowanie portów
- Wolumeny
- Zależności między kontenerami
- Zmienne środowiskowe

Do użycia Compose'a trzeba wykonać następujące kroki:

1. Zdefiniować pliki Dockerfile dla każdej aplikacji, która ma się uruchomić w Compos'ie
2. Stworzyć plik *docker-compose.yml* i zdefiniować w nim wszystkie aplikacje, które mają być razem uruchomione
3. Wywołać komendę `docker-compose build` w folderze, w którym znajduje się plik *docke-compose.yml*
4. Wywołać komendę `docker-compose up`.

Plik *docker-compose.yml* wykorzystuje język *YAML*. Należy zwrócić szczególną uwagę na formatowanie (wcięcia i białe znaki) podczas tworzenia pliku. Aby upewnić się, że konfiguracja jest dobrze sformatowana, można skorzystać z strony https://codebeautify.org/yaml-validator/. Compose, jak każde inne narzędzie programistyczne, posiada kolejne wersje. Dla każdej z nich istnieje dokumentacja zawierająca przykłady składni akceptowanej w niej. Na potrzeby laboratorium wykorzystany zostanie Compose w wersji 3. Dokumentacja znajduje się pod tym linkiem: https://docs.docker.com/compose/compose-file/(należy wybrać wersję 3 z menu po prawej stronie).

```yaml
version: '3'

services:

  serwis_1:
    build: .
    image: nazwa_obrazu
    networks:
      - siec1
    ports:
      - port_maszyny_hosta:port_wewnątrz_kontenera
    environment:
      - nazwa_zmiennej_srodowiskowej=wartosc
    depends_on:
      - inny_serwis

  inny_serwis:
    build: ./inny_serwis
    image: nazwa_obrazu2
    networks:
      - siec1

networks:
  siec1:
    driver: bridge
```

Rys. 3.1 Przykład pliku docker-compose.yml

Pierwszą rzeczą, jaką należy zdefiniować w pierwszej linijce pliku, jest wersja Compose'a, według której ma być interpretowany plik. Następnie możemy zdefiniować trzy sekcje główne - serwisy (*services*), sieci (*networks*) oraz volumeny (*volumes*). Każdą aplikację, która ma być uruchomiona w oddzielnym kontenerze, definiuje się w grupie *services*.

Każdy serwis musi mieć nadaną nazwę, po której następuje opis jego konfiguracji. Poniżej opisane zostały najważniejsze elementy konfiguracji serwisu. Wszystkie dostępne opcje można znaleźć w dokumentacji Compose'a.

build – służy do zdefiniowania sposobu, w jaki ma zostać zbudowany kontener dla serwisu. Parametr build ustawiony pojedynczą wartością typu string wskazuje na kontekst, w jakim ma zostać zbudowany kontener serwisu. Pod tą ścieżką musi znajdować się plik Dockerfile, inaczej narzędzie zwróci błąd budowania. Parametr build pozwala na bardziej szczegółową konfigurację poprzez dodanie dodatkowych parametrów w kolejnych liniach (przykłady zawierają się w dokumentacji Compose'a).

image – służy do wskazania obrazu, jaki powinien zostać uruchomiony dla tego serwisu. Jeśli parametr build został zdefiniowany, to Compose zbuduje nowy obraz z nazwą podaną w parametrze image lub jeśli znajdzie już istniejący obraz o tej nazwie, to go zaktualizuje.

networks – służy do wskazania listy sieci, w jakich działać ma dany serwis.

ports – służy do zdefiniowania mapowania portów między kontenerem a maszyną hosta. Mapowań może być więcej niż jedno, podawane są po myślnikach. Mapowanie podajemy w cudzysłowach, np. `- "81:80"`

environment – służy do zdefiniowania zmiennych środowiskowych, które będą ustawione w kontenerze. Zmienne mogą być potem odczytywane przez aplikacje uruchomione wewnątrz tego kontenera, np. do pobrania konfiguracji programu.

depends_on – służy do wskazania zależności między kontenerami. Pozwala to ustalić Compose'owi kolejność uruchamiania kontenerów. Np. baza danych powinna zostać uruchomiona przed kontenerem z aplikacją, która z niej korzysta, żeby uniknąć crasha aplikacji.

Compose prócz budowania kontenerów pozwala także zbudować sieć do komunikacji między nimi oraz maszyną hosta. W tym celu należy w sekcji głównej networks podać definicję sieci, z jakich mają korzystać serwisy. Każda definicja sieci musi zaczynać od nazwy (musi być unikalna). Po niej w kolejnych liniach po wcięciu znajdują się ustawienia tej sieci. Parametr **driver** służy do ustawienia typu sieci, jaki zostanie zbudowany. Domyślnie zawsze zostanie użyty typ *bridge*. Więcej o typach sieci można przeczytać na stronie: https://docs.docker.com/compose/compose-file/#network-configuration-reference.

Kontenery widzą siebie nawzajem wewnątrz sieci po aliasach. Aliasy te są nadawane na podstawie nazwy serwisu. Przykładowo serwis o nazwie *moj.serwis* będzie widoczny z aplikacji drugiego serwisu pod adresem *http://moj.serwis*.

4. Zadania laboratoryjne

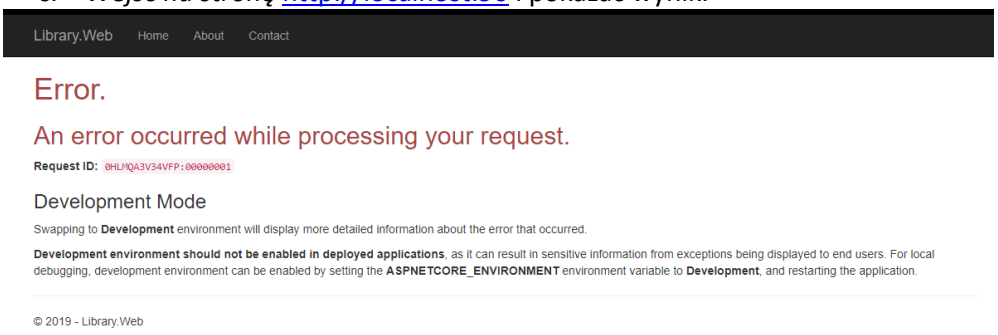
Do zadania wykorzystana zostanie gotowa solucja o nazwie Library, w której zawarte są trzy projekty:

- Library.Web – aplikacja webowa z interfejsem graficznym, która wyświetla dane pobrane z serwisu Library.WebApi. Aplikacja dostępna jest pod adresem *localhost* na porcie 80 wewnątrz kontenera.

- Library.WebApi – serwis, który wystawia dwa endpointy, jeden odpowiedzialny za zwracanie listy książek w bibliotece, drugi który pozwala wypożyczyć książkę o zadanym Id.
- Library.NotificationService2 – aplikacja, która wyświetla w konsoli informacje o wypożyczeniu danego egzemplarza. Aplikacja jest zasubskrybowana do wiadomości o wypożyczeniu, którą nadaje serwis Library.WebApi poprzez kolejkę RabbitMQ.

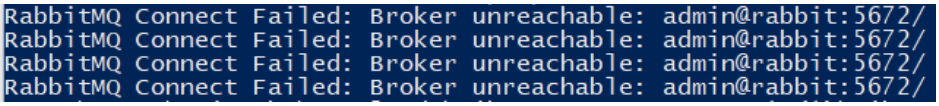
Zadania:

1. Dodać do projektu Library.Web i Library.WebApi dodać pliki Dockerfile.
2. Zbudować i uruchomić kontener z aplikacją Library.Web. (wykorzystując polecenia z poprzedniej laborki).
    a. Aplikacja powinna być dostępna z przeglądarki na porcie 90 (Powinna się pokazać po wpisaniu adresu http://localhost:90). Należy dodać mapowanie portów przy uruchamianiu kontenera.
    b. Przy uruchamianiu należy ustawić ścieżkę środowiskową o nazwie *LibraryWebApiServiceHost* na wartość *http://localhost:81.*
    c. *Wejść na stronę* http://localhost:90 i pokazać wynik.



Taki wynik powinien zostać wyświetlony w przeglądarce

3. Zbudować i uruchomić kontener z aplikacją Library.WebApi
    a. Podpiąć się do kontenera i pokazać, co wyświetla się w konsoli.



Taki komunikat powinien widnieć w konsoli

4. Dodać do pliku docker-compose.yml znajdującym się w katalogu głównym solucji konfiguracje dla serwisów Library.Web oraz Library.WebApi.
    a. Serwis Library.Web powinien mieć:
        i. ustawione mapowanie portów z 90 na maszynie hosta na 80 w kontenerze
        ii. ustawioną ścieżkę środowiskową *LibraryWebApiServiceHost* z wartością *http://library.webapi*
        iii. tą samą sieć, w której znajdują się inne komponenty (sieć nazwya się *api*)
    b. Serwis Library.WebApi powinien mieć:
        i. ustawione mapowanie portów z 91 na maszynie hosta na 80 w kontenerze
        ii. tą samą sieć, w której znajdują się inne komponenty (sieć nazwya się *api*)
5. Uruchomić wszystkie serwisy w zdefiniowane w docker-compose.
6. Pokazać wynik pracy
    a. Wejść na stronę localhost:90 w przeglądarce i pokazać działającą stronę Library.Web.

b. Podpiąć się pod kontener z aplikacją NotificationService2 i pokazać, że po wykonaniu żądania pod adresem http://localhost:91/api/library/rent/1 wyświetla się komunikat o wypożyczeniu książki.