Docker 1

Laboratorium
Bartosz Brudek / Tomasz Goluch
2023/24

1. Wstęp

Zadaniem tego laboratorium jest poznanie podstawowych funkcjonalności Dockera takich jak pobieranie obrazów, uruchamianie ich w kontenerach oraz budowanie obrazów Dockera warstwa po warstwie. Kolejnym krokiem będzie przygotowanie plików *Dockerfile* 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 gotowy obraz Dockera przy pomocy plików *Dockerfile*.

2. Przygotowanie środowiska

Aby móc otworzyć i edytować solucję, należy zainstalować:

1. SDK .NET Core w odpowiedniej wersji w zależności od zainstalowanego Visual Studio (na stronie: https://dotnet.microsoft.com/download/dotnet-core/2.2). Najnowsza wersja biblioteki 2.2.**1xx** dla Microsoft Visual 2017 albo 2.2.**2xx** dla Microsoft Visual 2019.
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 wgrać 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.

3. Stworzenie Dockerfile'a

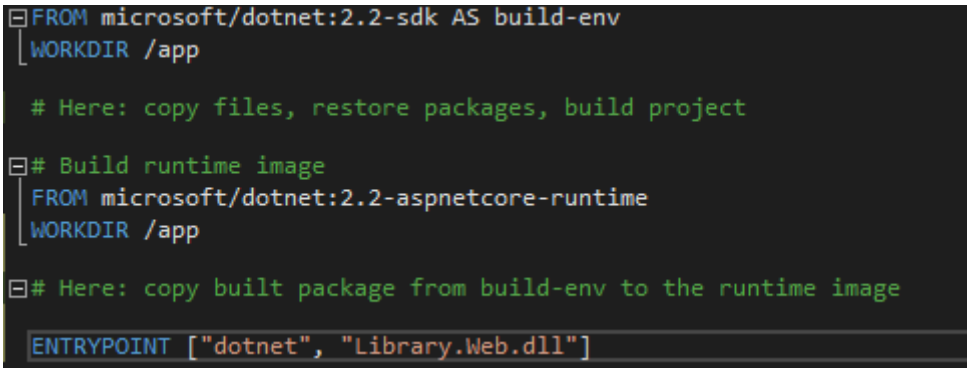
W systemie opartym na wielu usługach powinno się stosować takie podejście do wdrażania, w którym każda z nich może być uruchamiania oddzielnie. Idealnym narzędziem do tego zadania jest Docker. Każda aplikacja lub usługa powinna posiadać swój plik *Dockerfile*. Dobrą praktyką jest podzielenie go na dwa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

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

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

- `microsoft/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 posiadać strukturę przedstawioną na rysunku (rys.1):



Rys. 1. Szablon Dockerfile'a

4. Opis projektów wykorzystanych do laboratorium

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 90 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.

Każdy z projektów wymaga przekazania zmiennych konfiguracyjnych.

- Library.WebApi potrzebuje adresu serwera RabbitMq oraz danych potrzebnych do autoryzacji do komunikacji kolejkami,
- Libaray.NotificationService2 potrzebuje adresu serwera RabbitMq oraz danych potrzebnych do autoryzacji do komunikacji kolejkami,
- Library.Web potrzebuje znać adres serwisu Library.WebApi, żeby wykonywać zapytania do niego.

Aplikacje napisane w .NET Core, które wykorzystują klasę *HostBuilder* (wszystkie w solucji opierają się na niej) pozwalają na załadownie konfiguracji na wiele różnych sposobów. Na potrzeby laboratorium zostaną omówione dwa z nich:

- Zmienne środowiskowe – aplikacja w momencie uruchomienia sczytuje wszystkie zmienne środowiskowe ustawione w środowisku uruchomieniowym i załadowuje je do swojej pamięci
- Plik appsettings.json – aplikacja szuka w katalogu, w którym jest uruchamiana, pliku appsettings.json, a potem załadowuje jego zawartość do pamięci.

Tak załadowane dane konfiguracyjne można zmapować na obiekty klas, dzięki czemu w przejrzysty sposób można korzystać z nich w aplikacji.

UWAGA! – aplikacje wczytują konfiguracje obiema metodami, najpierw wczytując plik appsettings.json, a potem zmienne środowiskowe. Jeśli ustawisz zmienną obiema metodami, zostanie zapisana wartość z zmiennej środowiskowej!

Wszystkie aplikacje znajdujące się w solucji obsługują obie metody ładowania zmiennych konfiguracyjnych. W każdym projekcie można znaleźć plik appsettings.json, w którym wystarczy tylko podmienić potrzebne wartości zmiennych, aby aplikacja zaczęła działać.

5. Zadania laboratoryjne

0. Uruchomienie solucji Library na komputerze lokalnym.

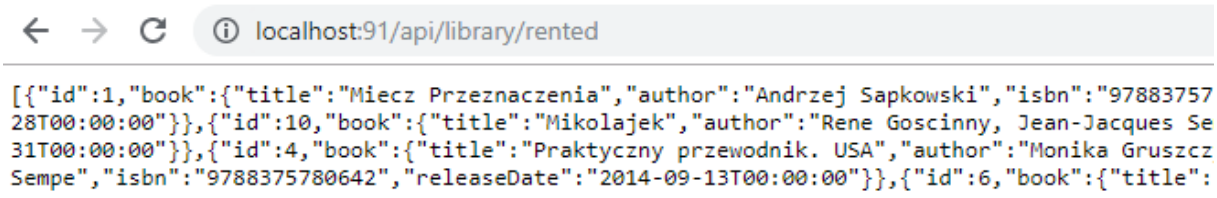
a. Pobrać ze strony prowadzącego pliki do laboratorium z Docker'a cz.1, rozpakować i uruchomić solucję. W przypadku problemów należy sprawdzić czy zainstalowana jest odpowiednia wersja Visual Studio ≥ 15.x.x.x oraz odpowiedni framework .NET. W przypadku problemów z Visual Studio proszę z poziomu katalogu z plikiem solucji (.sln) uruchomić następujące komendy:

```
dotnet publish
dotnet build
```

a następnie z poziomu katalogu z plikiem projektu (*.csproj):

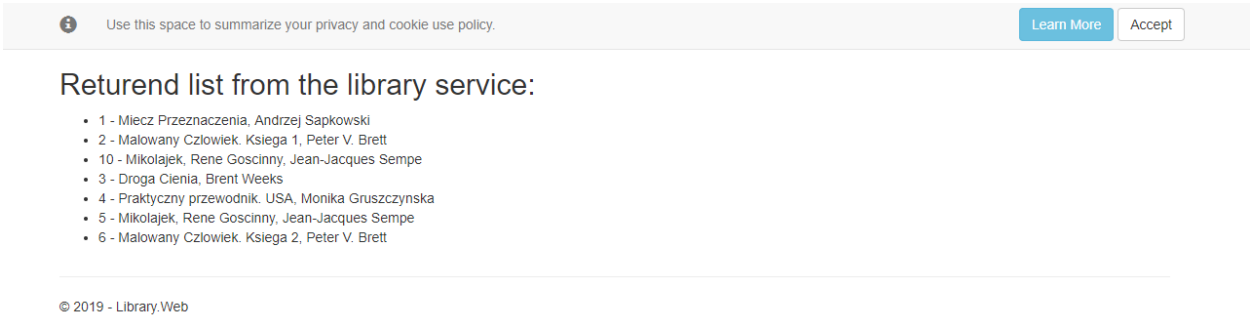
```
dotnet run
```

b. Proszę odpytać endpoint w przeglądarce: http://localhost:<NUMER_PORTU>/api/library/rented. Powinien on zwrócić listę książek (rys 2):



Rys. 2. Wynik zapytania endpointa Library.WebApi w przeglądarce.

W kolejnym kroku proszę odpytać endpoint http://localhost:<NUMER_PORTU> i pokazać wynik (rys. 3).



Rys. 3. Taki wynik powinien zostać wyświetlony w przeglądarce

c. Domyślnie solucja skonfigurowana jest do współpracy z serwerem RabbitMQ dostępnym na locahost'cie (dwa pliki appsettings.json w Library.WebApi i Library.NotificationService2):

```
"RabbitMq": {
  "Username": "guest",
  "Password": "guest",
  "ServerAddress": "rabbitmq://localhost"
},
```

Jeśli RabbitMQ nie jest zainstalowany na locahost proszę wpisać własne dane uwierzytelniające do konta z platformy: https://www.cloudamqp.com/. Jeśli w konsoli pokaże się wiadomość o takiej treści:

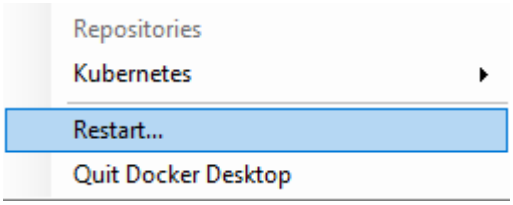
```
RabbitMQ Connect Failed: Broker unreachable: guest@localhost:5672/
RabbitMQ Connect Failed: Broker unreachable: guest@localhost:5672/
```

To znaczy, że prawdopodobnie adres/dane logowania są błędne. Po pomyślnej konfiguracji i prawidłowym zalogowaniu możemy wypożyczyć książkę odpytując odpowiedni endpoint w przeglądarce: http://localhost:<NUMER_PORTU>/api/library/rent/<NUMER_KSIĄŻKI>. W konsoli aplikacji Library.NotificationService2 powinniśmy dostać stosowny komunikat (rys.4):

```
The Miecz Przeznaczenia with id: 1 was rented
The Malowany Czlowiek. Ksiega 1 with id: 2 was rented
```

Rys. 4. Informacja aplikacji Library.NotificationService2 o poprawnym wypożyczeniu książki.

1. **Ściągnięcie i uruchomienie kontenera RabbitMQ.**
   a. Działający lokalnie system będziemy krok po kroku przenosić do kornerów Dockera. W pierwszym kroku uruchomimy obraz serwera RabbitMQ. Proszę pobrać obraz i uruchomić kontener z RabbitMQ.
      i. Obraz nazywa się **rabbitmq:management** i znajduje się na oficjalnym dockerhubie. W przypadku z problemów z komunikacją z serwerem Dockera przydatny może okazać się jego restart. Dla Docker Desktop wybieramy opcję Restart z menu Dockera z zasobnika systemowego:



Dla Docker Toolbox będzie to komenda:

```
docker-machine restart
```

      ii. Kontener powinien mieć ustawione następujące mapowania portów:
          "5672:5672",
          "15672:15672"
   b. Zmodyfikować dane konfiguracyjne (pliki appsettings.json) w aplikacji Library.WebApi oraz Library.NotificationService2 do komunikacji w obrazem:

```
"RabbitMq": {
  "Username": "guest",
  "Password": "guest",
  "ServerAddress": "rabbitmq://localhost"
},
```

Uwaga, jeśli laboratorium odbywa się na Docker Toolbox to zamiast `localhost` proszę użyć adresu ip docker-machine (wyświetlanego przy starcie Docker Tollbox) np.: `"ServerAddress": "rabbitmq://192.168.99.100"`.

   c. Uruchomić całą soolucję lokalnie, tak, aby aplikacje Library.WebApi i Libarary.NotificationService2 połączyły się z RabbitMQ działającym w kontenerze z zadania nr. 3.
   d. Ponownie proszę odpytać odpowiedni endpoint w przeglądarce: http://localhost:<NUMER_PORTU>/api/library/rent/<NUMER_KSIĄŻKI> i pokazać, że

aplikacja Library.NotificationService2 odebrała komunikat o wypożyczeniu książki (rys.4).

2. **Zbudowanie obrazu zawierającego aplikację Libarary.NotificationService2 (bez użycia Dockerfile).**
   a. W kolejnym kroku zbudujemy obraz zawierający aplikację Libarary.NotificationService2. Proszę uruchomić nowe okno wiersza poleceń albo powershella, ewentualnie opuścić kontener RabbitMQ bez zamykania go. Następnie, proszę pobrać obraz **microsoft/dotnet:2.2-sdk** i uruchomić kontener o nazwie *notificationservice* z mapowaniem portów „92:80" (adres_hosta:adres_wewnątrz_kontenera).
   b. Proszę dostosować konfigurację w pliku appsettings.json do komunikacji w obrazem *rabbitmq:management* (szczegóły będą opisane w pkt. 4g):

```
"RabbitMq": {
  "Username": "guest",
  "Password": "guest",
  "ServerAddress": "rabbitmq://rabbit"
},
```

   c. Skopiować do kontenera *notificationservice* zawartość folderu *Libarary.NotificationService2* (zawarty w plikach potrzebnych do laboratorium).
   d. Podłączyć się do uruchomionego kontenera z poziomu wiersza poleceń lub konsoli powershell.
   e. Uruchomić komendę w kontenerze: *dotnet restore.*
   f. Uruchomić komendę w kontenerze: *dotnet publish -c Release -o out*.
   g. Wejść do katalogu out i uruchomić komendę w kontenerze: *dotnet Library.NotificationService2.dll*.

Wynikiem powinien być podobny komunikat w kontenerze:

```
Hosting environment: Production
Content root path: /app
Now listening on: http://[::]:80
Application started. Press Ctrl+C to shut down.
```

   h. Sprawdzić czy serwis jest dostępny pod adresem hosta: http://localhost:92:

```
Notification Service started!
```

   i. Wyjść z kontenera.
   j. Utrwalić aktualny stan kontenera jako obraz *notificationserviceimg*.
   k. Sprawdzić, że znajduje się on na liście dostępnych obrazów.
   l. Uruchomić obraz tak aby uruchomiona została również aplikacja znajdująca się wewnątrz kontenera.
   m. Ponownie sprawdzić czy serwis jest dostępny pod adresem hosta: http://localhost:92.
3. **Połączenie kontenera RabbitMQ z kontenerem Libarary.NotificationService2 przy pomocy docker network.**
   a. Ponieważ dwa kontenery nie widzą siebie nawzajem poprzez maszynę hosta. W takim przypadku trzeba je przypiąć do tej samej sieci. Służą do tego następujące komendy:

b. `docker network create my-network` – utworzenie prostej sieci o nazwie *my-network* pozwalającej na komunikację pomiędzy kontenerami uruchamianymi na jednej maszynie Dockera.
   `docker network connect my-network <ID_KONTENERA_RABBITA> --alias rabbit` – podpięcie kontenera (w tym przypadku *rabbitmq*) do sieci.
   `docker run … --network my-network` – uruchomienie kontenera wraz z podpięciem do sieci.
c. Komendą `docker inspect <NAZWA_SIECI>` proszę sprawdzić które kontenery udało się przypiąć do sieci.
d. Komendą `docker inspect <ID_KONTENERA>` proszę czy kontenery posiadają poprawne aliasy.
e. Uruchomić tylko dwie aplikacje lokalnie (Library.WebApi i Library.Web), bez Libarary.NotificationService2 tak, żeby połączyły się z RabbitMQ działającym w kontenerze z zadania nr. 1.
f. Ponownie proszę odpytać odpowiedni endpoint w przeglądarce: http://localhost:<NUMER_PORTU>/api/library/rent/<NUMER_KSIĄŻKI> i pokazać, że aplikacja Library.NotificationService2 odebrała komunikat o wypożyczeniu książki (rys.4).

Wynikiem powinien być podobny komunikat w kontenerze:

```
RabbitMQ Connect Failed: Broker unreachable: guest@rabbit:5672/
RabbitMQ Connect Failed: Broker unreachable: guest@rabbit:5672/
Hosting environment: Production
Content root path: /app/out
Now listening on: http://[::]:80
Application started. Press Ctrl+C to shut down.
The Malowany Czlowiek. Ksiega 1 with id: 2 was rented
The Mikolajek with id: 5 was rented
```

4. **Zbudowanie obrazu zawierającego aplikację Libarary.WebApi z wykorzystaniem Dockerfile.**
   a. W kolejnym kroku zbudujemy obraz zawierający aplikację Libarary.WebApi ale z wykorzystaniem Dockerfile. Proszę dodać do projektu Library. WebApi plik Dockerfile.
   b. Proszę dostosować konfigurację w pliku appsettings.json do komunikacji w obrazem *rabbitmq:management*. Uwaga, jeśli laboratorium odbywa się na Docker Toolbox to zamiast `localhost` proszę użyć `0.0.0.0` np.: `"Url": "http://0.0.0.0:91"`.
   c. Zbudować i uruchomić kontener z aplikacją Library.WebApi.
   d. Aplikacja powinna być dostępna z przeglądarki na porcie 91 (Powinna się pokazać po wpisaniu adresu http://localhost:91/ api/library/rented/). Należy dodać mapowanie portów przy uruchamianiu kontenera z portu 80 na maszynie hosta na port 90 w kontenerze.
   e. Przypiąć kontener do tej samej sieci pamiętając o ponownym nadaniu aliasu pod którym będzie widoczny kontener *rabbitmq:management*.
   f. Ponownie proszę odpytać odpowiedni endpoint w przeglądarce: http://localhost:<NUMER_PORTU>/api/library/rent/<NUMER_KSIĄŻKI> i pokazać, że aplikacja Library.NotificationService2 odebrała komunikat o wypożyczeniu książki (rys.4).
   g. Uruchomić lokalnie aplikację Library.Web.
   h. Aplikacja powinna mieć zmodyfikowany plik appsettings.json tak, żeby połączyła się z kolejką RabbitMQ z kontenerem z obrazu *rabbitmq:management*.

i. *Wejść na stronę* http://localhost:90 i pokazać wynik (rys. 3).

5. **Zaliczenie laboratorium.**
   a. Proszę przesłać następujące pliki:
      i. print screen'y dokumentujące wykonanie każdego z podpunktów laboratorium
      ii. historię z każdego okna poleceń. W linuksie służy do tego komenda: `history > d:/history.txt`, a w Windows (cmd): `doskey /HISTORY > d:\history.txt`, (powershell): `Get-History | Export-CSV D:\history.csv`
      iii. wszystkie pliki Dockerfile
      iv. skrypty – jeśli były wykorzystywane
      v. pliki appsettings.json wykorzystywane do komunikacji z obrazem *rabbitmq:management*.

6. Błędy

Pakiet Microsoft.AspNetCore.Authentication.Google 2.1.0 nie jest zgodny z elementem netcoreapp2.2 (.NETCoreApp,Version=v2.2). Pakiet Microsoft.AspNetCore.Authentication.Google 2.1.0 obsługuje: netstandard2.0 (.NETStandard,Version=v2.0) – Nieaktualna wersja Visual Studio 2017, proszę uaktualnić w Rozszerzenia i aktualizacje → Aktualizacje → Aktualizacja programu Visual Studio 15.x.x.x. Operacja jest czasochłonna i może to potrwać.