

Docker 2

Laboratorium
Bartosz Brudek
2020

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 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 uruchamiana 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):

```

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. Szablon Dockerfile'a

4. Stworzenie docker-compose'a

Compose to narzędzie służące do zdefiniowania 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 `docker-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).

```

1  version: '3'
2
3  services:
4
5      serwis_1:
6          build: .
7          image: nazwa_obrazu
8          networks:
9              - siec1
10         ports:
11             - port_maszyny_hosta:port_wewnatrz_kontenera
12         environment:
13             - nazwa_zmiennej_srodowiskowej=wartosc
14         depends_on:
15             - inny_serwis
16
17     inny_serwis:
18         build: ./inny_serwis
19         image: nazwa_obrazu2
20         networks:
21             - siec1
22
23     networks:
24         siec1:
25             driver: bridge

```

Rys. 4.1 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`.

5. 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,
- Library.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 czytuje 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 konfigurację 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.

6. Zadania laboratoryjne

1. Uruchomienie solucji Library na komputerze lokalnym.

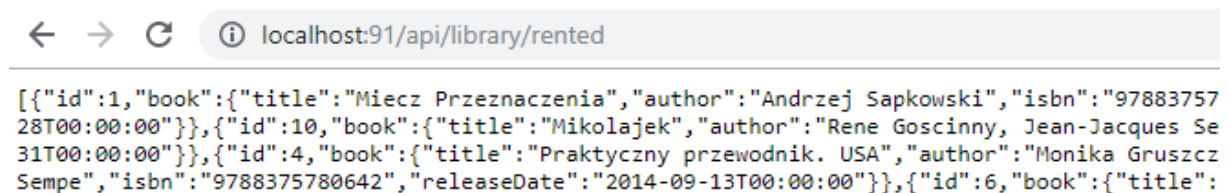
- Pobrać ze strony prowadzącego pliki do laboratorium z Docker'a cz.1, rozpakować i uruchomić solucję. W przypadku problemów należy sprawdzić czy zainstalowana jest odpowiednia wersja Visual Studio $\geq 15.x.x.x$ oraz odpowiedni framework .NET. W przypadku problemów z Visual Studio proszę z poziomu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

```
dotnet publish
dotnet build
```

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

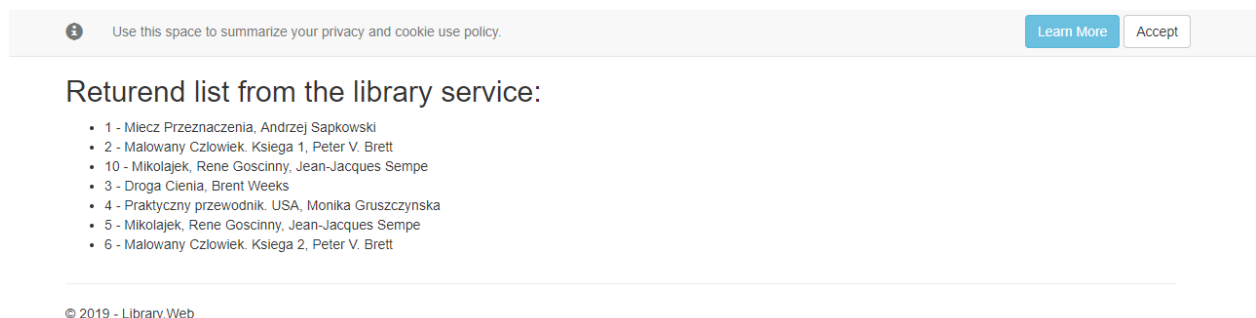
```
dotnet run
```

- 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 localhost'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 localhost 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/
```

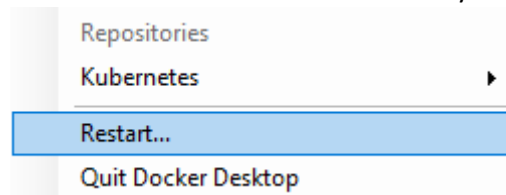
To znaczy, że prawdopodobnie adres/dane logowania są błędne. Po pomyślnej konfiguracji i prawidłowym zalogowaniu możemy wypożyczyć książkę odpytując odpowiedni endpoint w przeglądarce: http://localhost:<NUMER_PORTU>/api/library/rent/<NUMER_KSIAŻKI>. W konsoli aplikacji Library.NotificationService2 powinniśmy 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. Uruchomienie kontenera **rabbitmq:management** oraz 3 kontenerów solucji Library przy użyciu plików Dockerfile.

- a. 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. Dodać do projektów Library.Web, Library.WebApi oraz Library.NotificationService2 pliki Dockerfile.
- c. Zbudować i uruchomić kontener z aplikacją Library.Web.
- d. 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 z portu 90 na maszynie hosta na port 80 w kontenerze. 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>".

- e. Przy uruchamianiu należy ustawić zmienną środowiskową o nazwie `LibraryWebApiServiceHost` na wartość <http://localhost:91>.
- f. Uruchomić lokalnie aplikację `Library.WebApi`.
- g. Aplikacja powinna mieć zmodyfikowany plik `appsettings.json` tak, żeby połączyła się z kolejką `RabbitMQ` na serwerze wskazanym przez prowadzącego.
- h. Wejść na stronę <http://localhost:90> i pokazać wynik.

 Use this space to summarize your privacy and cookie use policy.

[Learn More](#) [Accept](#)

### Returned list from the library service:

- 1 - Miecz Przeznaczenia, Andrzej Sapkowski
- 2 - Malowany Człowiek. Księga 1, Peter V. Brett
- 10 - Mikołajek, Rene Goscinny, Jean-Jacques Sempe
- 3 - Droga Cienia, Brent Weeks
- 4 - Praktyczny przewodnik. USA, Monika Gruszczynska
- 5 - Mikołajek, Rene Goscinny, Jean-Jacques Sempe
- 6 - Malowany Człowiek. Księga 2, Peter V. Brett

© 2019 - Library.Web

*Taki wynik powinien zostać wyświetlony w przeglądarce*

- i. Zbudować i uruchomić kontener z aplikacją `Library.WebApi`
- j. Podpiąć się do kontenera i pokazać, co wyświetla się w konsoli.

```
RabbitMQ Connect Failed: Broker unreachable: admin@rabbit:5672/
RabbitMQ Connect Failed: Broker unreachable: admin@rabbit:5672/
RabbitMQ Connect Failed: Broker unreachable: admin@rabbit:5672/
RabbitMQ Connect Failed: Broker unreachable: admin@rabbit:5672/
```

*Taki komunikat powinien widnieć w konsoli*

## 2. Uruchomienie solucji `Library` przy użyciu `Docker Compose`.

- a. Dodać plik `docker-compose.yml` do katalogu głównego solucji z konfiguracją dla serwisów `Library.NotificationService2`, `Library.Web` oraz `Library.WebApi`.
  - i. Serwis `Library.NotificationService2` powinien mieć:
    - 1. **DOPISAĆ CO MA MIEĆ I ŻE JEDEN Z TYCH TRZECH OBRAZÓW POWINIEN BY BUDOWANY W COMPOSE NA PODSTAWIE DOCKERFILE A RESZTA KORZYSTAĆ Z JUŻ ZBUDOWANYCH OBRAZÓW.**
  - ii. Serwis `Library.Web` powinien mieć:
    - 1. ustawione mapowanie portów z 90 na maszynie hosta na 80 w kontenerze
    - 2. ustawioną ścieżkę środowiskową `LibraryWebApiServiceHost` z wartością `http://library.webapi`
    - 3. tą samą sieć, w której znajdują się inne komponenty (sieć nazywa się *api*)
  - iii. Serwis `Library.WebApi` powinien mieć:
    - 1. ustawione mapowanie portów z 91 na maszynie hosta na 80 w kontenerze
    - 2. tą samą sieć, w której znajdują się inne komponenty (sieć nazywa się *api*)
- b. Uruchomić wszystkie serwisy w zdefiniowane w `docker-compose`.
- c. Pokazać wynik pracy
  - i. Wejść na stronę `localhost:90` w przeglądarce i pokazać działającą stronę `Library.Web`.

- ii. 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.

```
library.notificationService2_1 | The Miecz Przeznaczenia with id: 1 was rented
library.notificationService2_1 | The Malowany Człowiek. Księga 1 with id: 2 was rented
```

*Takie komunikaty powinny się wyświetlić na konsoli podpiętej do serwisu library.notificationService, kiedy zostanie wywołany endpoint `localhost:91/api/library/rent/1` oraz `localhost:91/api/library/rent/2`*

### 3. Zaliczenie laboratorium.

- a. Proszę przesłać następujące pliki:
  - i. print screen'y dokumentujące wykonanie każdego z podpunktów laboratorium
  - ii. wszystkie pliki Dockerfile i docker-compose.yml
  - iii. skrypty – jeśli były wykorzystywane
  - iv. pliki appsettings.json wykorzystywane do komunikacji z obrazem `rabbitmq:management`.

## 7. 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.