LABORATORIUM 10. TESTOWANIE APLIKACJI WEB API

Cel laboratorium:

Celem zajęć jest przygotowanie aplikacji służącej do cenzurowania słów w taki sposób, aby spełniła założenia określone za pomocą testów jednostkowych, opracowanie dla niej testów integracyjnych oraz wdrożenie z wykorzystaniem docker-compose.

Liczba punktów możliwych do uzyskania: 8 punktów

Zakres tematyczny zajęć:

Testy jednostkowe jako sposób określania wymagań funkcjonalnych,
Opracowywanie testów integracyjnych poprzez symulowany serwer aplikacji,
Publikacja aplikacji za pomocą dotnet publish,
Publikacja aplikacji do postaci kontenera Docker,
Wdrażanie aplikacji z wykorzystaniem docker-compose.

Pytania kontrolne:

1. Czym różnią się testy jednostkowe i integracyjne?
2. W jakim celu stosuje się konteneryzację aplikacji?
3. Jakie znasz mechanizmy zarządzania (orkiestracji) kontenerów?

Zadanie 10.1. Implementacja mechanizmu cenzury

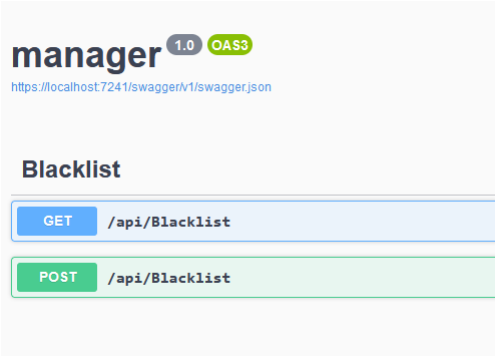
Pobierz z Moodle przykładowy projekt „Lab10”. Składa się on z 3 projektów: „api” to podsystem, który realizuje mechanizm „cenzury” – po wysłaniu tekstu powinien on zwracać ocenzurowany tekst, na podstawie listy słów „zakazanych” – słowa znajdujące się na liście słów zakazanych mają zostać zamienione w taki sposób, aby wszystkie znaki poza pierwszym zostały zamienione na znak „*”, na przykład słowo „bomba” na `b****`.

W projekcie „test” przygotowany jest zestaw testów jednostkowych, które sprawdzają poprawność działania metod klasy `Censor`.

Zaimplementuj klasę `Censor` i jej publiczne metody w taki sposób, aby wszystkie testy jednostkowe zostały spełnione.

Zadanie 10.2. Testy integracyjne menedżera zakazanych słów

Drugi projekt z przykładowego projektu, „manager”, odpowiada za dostarczanie do kontrolera API listy słów, które mają być cenzurowane. Dostarcza dwie metody, jak przedstawiono na rysunku 10.1.



Rys 10.1. Dostępne metody interfejsu API aplikacji „manager”

Wysłanie żądania typu GET powoduje zwrócenie wszystkich słów, które obecnie są w bazie danych. Wysłanie żądania typu POST z atrybutem o nazwie `word` w łańcuchu zapytania (*query string*) powoduje dodanie nowego słowa do bazy.

„manager” wykorzystuje prosty mechanizm persystencji w postaci serializacji danych do pliku tekstowego.

Przygotuj testy integracyjne, które sprawdzą, czy mechanizm działa poprawnie. W tym celu:

Dodaj nową klasę do projektu test, o nazwie `CustomWebApplicationFactory` o następującej zawartości:

```
public class CustomWebApplicationFactory<TStartup> :
WebApplicationFactory<TStartup>
    where TStartup : class
{
    private string temp = Path.GetRandomFileName();

    protected override void ConfigureWebHost(IWebHostBuilder
builder)
    {
        builder.ConfigureServices(services =>
        {
            services.AddSingleton<IBlacklistStorage>(new
BlacklistStorage(temp));
        });
    }

    protected override void Dispose(bool disposing)
    {
        base.Dispose(disposing);
        File.Delete(temp);
```

```
    }
}
```

Klasa ta tworzy nowy, tymczasowy odpowiednik rzeczywistego serwera, do fazy testów, polegając w przypadku `BlacklistStorage` na pliku tymczasowym, który jest usuwany po zakończeniu testów.

A następnie dodaj nową klasę, która będzie zawierać właściwy mechanizm testów integracyjnych, `ManagerIntegrationTests`:

```
public class ManagerIntegrationTests :
IClassFixture<CustomWebApplicationFactory<Program>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<Program>
_factory;

    public
ManagerIntegrationTests(CustomWebApplicationFactory<Program>
factory)
    {
        _factory = factory;
        _client = factory.CreateClient(
            new WebApplicationFactoryClientOptions {
AllowAutoRedirect = false }
        );
    }

    [Fact]
    public async void Get_WhenEmpty_ReturnEmptyList()
    {
        var act = await _client.GetAsync("/api/Blacklist");

        Assert.True(act.IsSuccessStatusCode);
        var json = JsonSerializer.Deserialize<string[]>(await
act.Content.ReadAsStringAsync());

        Assert.Empty(json);
    }
}
```

W tym przykładzie został przygotowany jeden test, który pobiera z systemu manager dane, jeżeli żadne nie zostały wprowadzone, czego wynikiem powinna być pusta kolekcja.

Na wzór tego testu przygotuj test, który sprawdzi, czy po dodaniu elementu do listy jest on widoczny w liście zwracanej przez manager.

Zadanie 10.3. Aplikacja internetowa wykorzystująca mechanizm cenzury

W projekcie „api" dodaj w pliku `Program.cs` wywołanie:

```
app.UseFileServer();
```

powyżej `app.Run()`, a następnie dodaj folder `wwwroot` i utwórz w nim plik `index.html`. Opracuj aplikację internetową w języku JavaScript, która po wciśnięciu przycisku prześle wprowadzony przez użytkownika do pola tekstowego tekst do serwisu api, pod adres `/censor?text=tekst`, odbierze odpowiedź i wyświetli go w wersji „ocenzurowanej" poniżej.

Pamiętaj, że kiedy chcesz testować aplikację „api", uruchomiona musi być również jednocześnie aplikacja „manager" – ponieważ każde odwołanie do *endpointu* `/censor` odwołuje się do aplikacji „manager".

Aby wykonać żądanie JavaScript możesz skorzystać z następującego przykładu:

```
const output = document.getElementById('alert');
const textarea = document.getElementById('text');

function censorText() {
    const uri = `/censor?text=${textarea.value}`;

    fetch(uri, { method: 'POST' })
        .then(response => response.text())
        .then(text => {
            output.innerText = text;
        });
}
```

Zadanie 10.4. Publikacja za pomocą dotnet publish

Aby przygotować wersję aplikacji, która może być uruchomiona na serwerze wykorzystywana jest komenda `dotnet publish`. Generuje ona wyjściową „paczkę" plików, które następnie mogą być skopiowane i uruchomione na innym komputerze. W zależności od wybranego podejścia biblioteki całego .NET mogą być dołączone do projektu lub można korzystać z tych zainstalowanych na komputerze docelowym.

Uruchom narzędzie Terminal, konsolę lub Wiersz Polecenia i przejdź do folderu projektu „api". Następnie spróbuj wydać komendę:

```
dotnet publish
```

W folderze `/bin/Debug/net6.0/publish` znajdą się testowe (Debug) wersje aplikacji przeznaczone do uruchomienia na komputerach, na których znajduje się już zainstalowane środowisko .NET.

W folderze projektu „api" wydaj komendę:

```
dotnet publish -c Release
```

Projekt zostanie skompilowany i opublikowany w konfiguracji Release, z włączonymi optymalizacjami i w wersji pozbawionej części symboli do debugowania. Użyj komendy:

```
dotnet publish -c Release --self-contained -r linux-x64
```

I sprawdź zawartość folderu `/bin/Release/net6.0/linux-x64/publish/`

Znajdzie się tam wersja aplikacji przygotowana do uruchomienia na komputerach z systemem Linux, na procesorach w architekturze x86-64, bez zainstalowanej platformy .NET.

Spróbuj uruchomić tę aplikację, np. w środowisku WSL2.

Zadanie 10.5. Tworzenie kontenerów Dockera

Dodaj do projektu „api" plik `Dockerfile`. Plik `Dockerfile` definiuje w jaki sposób powinna być przygotowywana aplikacja aby móc zbudować obraz kontenera. Możesz skorzystać z następującego przykładu pliku `Dockerfile` dla projektu „api":

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["api/api.csproj", "api/"]
RUN dotnet restore "api/api.csproj"
COPY . .
WORKDIR "/src/api"
RUN dotnet build "api.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "api.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
```

```
ENTRYPOINT ["dotnet", "api.dll"]
```

Plik ten definiuje, że na początku kopiowane są pliki `.csproj` i wykonywana komenda `dotnet restore`, a potem wszystkie pliki projektu i jest wykonywana znana już ci komenda `dotnet publish`. Wynik zapisywany jest do kontenera o nazwie `final`, który opiera się o obraz dostarczany przez firmę Microsoft i system Linux Debian.

Przejdź do folderu głównego projektu `Lab10` i wydaj komendę:

```
docker build -t censorapi:latest -f api/Dockerfile .
```

Spowoduje to utworzenie obrazu kontenera, oznaczonego nazwą `censorapi` w wersji `latest`. Spróbuj go uruchomić, wydając komendę:

```
docker run --rm -it -p 5000:80 censorapi:latest
```

Aplikacja api powinna się uruchomić i być dostępna pod adresem http://localhost:5000/. Niestety, nie ma ona komunikacji z podsystemem „manager", stąd dostęp do jej funkcji jest niemożliwy. Naciśnij Ctrl+C, aby zakończyć jej działanie.

Na podstawie przykładowego pliku `Dockerfile` opracuj plik dla projektu manager i wygeneruj obraz kontenera, oznacz go `censormanager:latest`.

Zadanie 10.6. Wykorzystanie zmiennych konfiguracyjnych

Aplikacja „api" ma ustawione „na stałe" odwołanie do adresu aplikacji manager, co w sytuacji wdrażania przez konteneryzację nie będzie działało poprawnie i nie jest uniwersalne. Musimy zmodyfikować aplikację, aby adres „manager", z którego ma korzystać pobierała z konfiguracji, w szczególności ze zmiennych środowiskowych.

Mechanizm konfiguracji ASP.NET Core sam będzie wyszukiwał konfigurację we wszystkich możliwych źródłach (pliki `appsettings.json`, zmienne środowiskowe, pliki sekretów), więc wystarczy, aby dodać linijkę:

```
var url = app.Configuration["blacklistUrl"] ??
"https://localhost:7241";
```

Powyżej wywołania `app.MapPost()` w pliku `Program.cs`. w projekcie „api".

Następnie, w metodzie lambda wewnątrz `MapPost()` wykorzystaj zmienną `url` zamiast adresu ustawionego na sztywno.

Wygeneruj nowy obraz kontenera „censorapi", zawierający zmienioną wersję.

Zadanie 10.7. Uruchamianie systemu poprzez docker-compose

`docker-compose` pozwala na uruchamianie kontenerów zarządzając także siecią, wzajemnymi zależnościami i widocznością kontenerów. Przygotuj plik `docker-compose.yml` w głównym folderze (Lab10) twojej aplikacji o następującej treści:

```
version: '3.4'


services:
  manager:
    image: censormanager:latest
```

```
    restart: always
  api:
    image: censorapi:latest
    restart: always
    ports:
      - 5000:80
    environment:
      - ASPNETCORE_blacklistUrl=http://manager
    depends_on:
      - manager
```

Plik ten uruchamia dwie usługi: „manager” z obrazu `censormanager:latest` oraz „api” z obrazu `censorapi:latest`, gdzie ta druga ma odwołania do pierwszej – zależy od niej (więcej będzie uruchomiona później) oraz ma adres do usługi „manager” podany w zmiennej środowiskowej. Jak widzisz, zmienne środowiskowe standardowo są poprzedzane przez `ASPNETCORE_`. Aplikacja „api” startuje na porcie 5000 hosta i jest widoczna, natomiast aplikacja „manager” nie będzie widoczna „z zewnątrz”, poza wydzieloną siecią.

Przetestuj działanie swojej aplikacji wydając komendę:

```
docker-compose up
```

Naciśnij Ctrl+C, aby zakończyć działanie wszystkich elementów projektu.

Zadanie 10.8. Wykorzystanie mechanizmu raportów stanu zdrowia (*health checks*)

Docker i inne systemy orkiestracji kontenerów pozwalają na sprawdzenie, czy aplikacja raportuje swoje poprawne działanie i potrafią ewentualnie zrestartować aplikację, jeśli nie jest to prawda. Dla dodania do projektu „api” podstawowego systemu raportowania stanu aplikacji dodaj w pliku `Program.cs`:

```
builder.Services.AddHealthChecks();
```

Powyżej wywołania `builder.Build()` oraz:

```
app.MapHealthChecks("/healthz");
```

powyżej wywołania `app.Run()`.

Następnie, aby zintegrować to rozwiązanie z platformą Docker, możesz zmodyfikować plik `Dockerfile`, dodając:

```
RUN apt-get update && apt-get install -y curl && rm -rf
/var/lib/apt/lists/*
```

Poniżej wywołania `EXPOSE` oraz:

```
HEALTHCHECK CMD curl --fail http://localhost:80/healthz ||
exit
```

Poniżej wywołania `ENTRYPOINT`.

Teraz, aplikacja powinna raportować swój status, widoczny po jej uruchomieniu oraz wydaniu polecenia:

```
docker ps
```

W sposób, który zaprezentowano na rysunku 10.2.

```
E:\_t>docker ps
CONTAINER ID   IMAGE                 COMMAND                  CREATED         STATUS                   PORTS
              NAMES
f34f8dbab932   censorapi:latest      "dotnet api.dll"         3 minutes ago   Up 2 minutes (healthy)   443/tcp, 0.0.0.0:5
000->80/tcp   lab10-api-1
9b6eb74ec58a   censormanager:latest  "dotnet manager.dll"     3 minutes ago   Up 2 minutes             80/tcp, 443/tcp
              lab10-manager-1
```

Rys 10.2. Raportowanie stanu zdrowia aplikacji.