



**POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI**

**KIERUNEK STUDIÓW
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ
LABORATORYJNYCH***

Architektura i programowanie w .NET

Autor/Autorzy:
dr inż. Marcin Badurowicz

Lublin 2020



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



INFORMACJA O PRZEDMIOCIE

Cele przedmiotu:

1. Wprowadzenie studenta do języka C#, środowiska aplikacyjnego ASP.NET i ekosystemu platformy .NET,
2. Wykorzystanie środowiska ASP.NET do tworzenia wieloplatformowych aplikacji internetowych,
3. Wykorzystanie ekosystemu platformy .NET do tworzenia aplikacji współpracujących z bazami danych i innymi systemami informatycznymi.

Efekty kształcenia w zakresie umiejętności:

- Potrafi zaprojektować prosty system informatyczny korzystający z platformy ASP.NET,
- Potrafi zaprojektować, zaprogramować i wdrożyć aplikację wykorzystującą mechanizm ORM do komunikacji z systemem bazodanowym,
- Potrafi poprawnie zaprojektować architekturę aplikacji przy uwzględnieniu problematyki bezpieczeństwa oraz współpracy z innymi systemami informatycznymi.

Literatura do zajęć:

Literatura podstawowa

1. Smith, S., Architect Modern Web Applications with ASP.NET Core and Azure, Microsoft Developer Division, 2019
2. Chiaretta, S., Lattanzi, U., ASP.NET Core 2 Succinctly, Syncfusion 2019
3. Pańczyk, B., Badurowicz, M., Programowanie obiektowe: język C#, Politechnika Lubelska 2013
4. <https://dotnet.microsoft.com/learn/aspnet>

Literatura uzupełniająca

1. Price, M., C# 7.1 i .NET Core 2.0 dla programistów aplikacji wieloplatformowych, Helion 2018
2. Freeman A., ASP.NET Core MVC 2: Zaawansowane programowanie, Helion 2018
3. <https://channel9.msdn.com/Series/ASP.NET-Core---Beginner>.

Metody i kryteria oceny:

Oceny cząstkowe:

- Indywidualne rozwiązywanie zadań przygotowanych w ramach laboratorium, oceniane w skali punktowej dla każdego laboratorium.

Ocena końcowa - zaliczenie przedmiotu:

- Pozytywne oceny cząstkowe – łączna liczba punktów przewyższająca 36 punktów.
- Ewentualne dodatkowe wymagania prowadzącego zajęcia.

Każde laboratorium będzie miało przypisaną ogólną liczbę punktów możliwych do uzyskania. W celu zaliczenia laboratorium należy wystać na



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



platformę Moodle projekt stanowiący rozwiązanie wszystkich zadań laboratoryjnych z konkretnego laboratorium w czasie ustalonym przez prowadzącego zajęcia. Każdy dzień opóźnienia liczony wg. systemu Moodle będzie skutkował karą wynoszącą 0,1 punkta, aż do zmniejszenia liczby możliwych punktów do uzyskania z konkretnych zajęć do 0 (przykładowo: jeżeli laboratorium jest warte 5 punktów, to po 50 dniach opóźnienia student uzyska za nie maksymalnie 0 punktów).



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Plan zajęć laboratoryjnych:

Lab1.	Tworzenie aplikacji konsolowej w języku C# w paradygmacie programowania obiektowego.
Lab2.	Wykorzystanie mechanizmu LINQ do przetwarzania danych w aplikacji.
Lab3.	Tworzenie aplikacji internetowej w architekturze MVC.
Lab4.	Tworzenie aplikacji internetowej z wykorzystaniem podejścia Razor Pages.
Lab5.	Tworzenie aplikacji internetowej korzystającej z bazy danych oraz mechanizmu ORM.
Lab6.	Walidacja danych użytkownika po stronie aplikacji.
Lab7.	Tworzenie aplikacji internetowej przy wykorzystaniu podejścia „database first”.
Lab8.	Tworzenie aplikacji internetowej dostarczającej Web API
Lab9.	Wykorzystanie uwierzytelnienia w aplikacji typu API.
Lab10.	Testowanie i wdrażanie aplikacji Web API.



WYMAGANIA DO ŚRODOWISKA PRACY

Laboratorium do przedmiotu „Architektura i programowanie w .NET” zostało przygotowane z myślą o pracy w dwóch środowiskach programistycznych, do wyboru przez studenta – Microsoft Visual Studio 2022 (dostępne tylko na platformie Windows) oraz Microsoft Visual Studio Code na platformie Windows, Linux lub macOS, gdzie to podejście będzie wymagało wykorzystywania narzędzi konsolowych. Każde polecenie w którym będzie to wymagane, zostanie przygotowane w taki sposób, aby przedstawić wykorzystanie obydwu tych podejść.

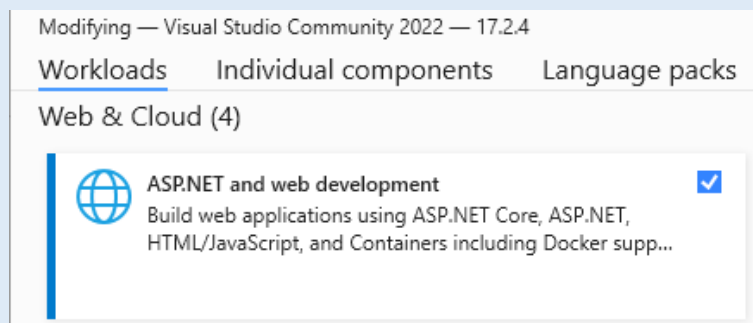
Student może używać innych narzędzi wybranych przez siebie (np. vim, JetBrains Rider, Sublime Text, Notepad), nie zostanie jednak opisane ich użycie.

Aby móc wykonywać zadania laboratoryjne na własnym komputerze, niezbędne jest zainstalowanie odpowiedniego oprogramowania, które zostało już odpowiednio przygotowane w salach laboratoryjnych.

- Polecenia, które należy wykonywać, jeżeli używa się Visual Studio 2022 zostaną oznaczone błękitnym kolorem tła.
- Polecenia, które należy wykonać, jeżeli używa się narzędzi konsolowych oraz np. Visual Studio Code, zostaną oznaczone jasnoszarym kolorem tła.
- Polecenia wspólne będą posiadały tło w kolorze białym.

Wariant dla Visual Studio 2022

Niezbędne jest zainstalowanie Visual Studio 2022 (edycja Community lub wyższa – <https://visualstudio.microsoft.com/pl/>) oraz wybranie w instalatorze zestawu narzędzi „ASP.NET and web development”.



Rys 1. Opcja niezbędna do zaznaczenia w instalatorze Visual Studio 2022

Przy pracy z instalatorem w trybie wiersza polecenia, należy wydać polecenie:

```
vs_community.exe --add Microsoft.VisualStudio.Workload.NetWeb  
-includeRecommended
```

Po zainstalowaniu narzędzi należy również doinstalować narzędzie konsolowe dotnet-ef i dotnet-aspnet-codegenerator wydając polecenia w Terminalu Windows lub Wierszu Poleceń (może być niezbędne podniesienie uprawnień):

```
dotnet tool install --global dotnet-ef  
dotnet tool install --global dotnet-aspnet-codegenerator
```

Wariant dla Visual Studio Code oraz narzędzi konsolowych



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Niezbędne jest zainstalowanie edytora (<https://code.visualstudio.com/>), pakietu .NET 6.0 SDK dla wybranej platformy (<https://dotnet.microsoft.com/en-us/download>) oraz rozszerzenia „C# for Visual Studio Code” w edytorze VSCode (<https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp>)¹ lub alternatywnie C# DevKit (<https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csdevkit>)².

Po zainstalowaniu pakietu .NET SDK należy doinstalować także globalne narzędzia `dotnet-ef` oraz `dotnet-aspnet-codegenerator` wydając komendy:

```
dotnet tool install --global dotnet-ef
dotnet tool install --global dotnet-aspnet-codegenerator
```

Przy wykorzystaniu systemu Windows 10 lub Windows 11 można wykorzystać narzędzie `winget` do zainstalowania .NET SDK³ oraz edytora Visual Studio Code:

```
winget install Microsoft.DotNet.SDK.6
winget install Microsoft.DotNet.AspNetCore.6
winget install Microsoft.VisualStudioCode
```

Oprócz narzędzi programistycznych niezbędne będzie zainstalowanie również (niezależnie od używanego systemu operacyjnego):

- Przeglądarki internetowej, np. Mozilla Firefox, Google Chrome, Microsoft Edge,
- Narzędzia do testowania REST API, np. Postman (<https://www.postman.com/>), Insomnia (<https://insomnia.rest/>), Nightingale (<https://nightingale.rest/>),
- Środowiska Docker lub kompatybilnego (dla potrzeb laboratorium 10) (<https://www.docker.com/>).

¹ Jeśli będziesz używać innego edytora niż VSCode, możesz zignorować posiadanie wtyczki, gdyż głównie będziemy skupiać się na komendach konsoli.

² W momencie przygotowywania tych materiałów DevKit jest w wersji zapoznawczej – aby używać „starego” OmniSharp należy ustawić w VSCode opcję `dotnet.server.useOmniSharp` na `true`,

³ Komendy podane poniżej instalują SDK w wersji 6.0, zadania wymagają wykorzystania konkretnie tej wersji w kilku miejscach.



LABORATORIUM 1. TWORZENIE APLIKACJI KONSOLOWEJ W JĘZYKU C# W PARADYGMACIE PROGRAMOWANIA OBIEKTOWEGO

Cel laboratorium:

Celem laboratorium będzie zapoznanie się z podstawami języka C# poprzez przygotowanie prostej aplikacji konsolowej.

Liczba punktów możliwych do uzyskania: 5 punktów

Zakres tematyczny zajęć:

Tworzenie nowego projektu aplikacji konsolowej w .NET 6,
Wykorzystanie prostych instrukcji języka,
Wykorzystanie wbudowanych klas do odczytu i zapisu informacji do konsoli,
Tworzenie własnych klas,
Wykorzystanie klasy System.Text.Json do serializacji i deserializacji formatu JSON.

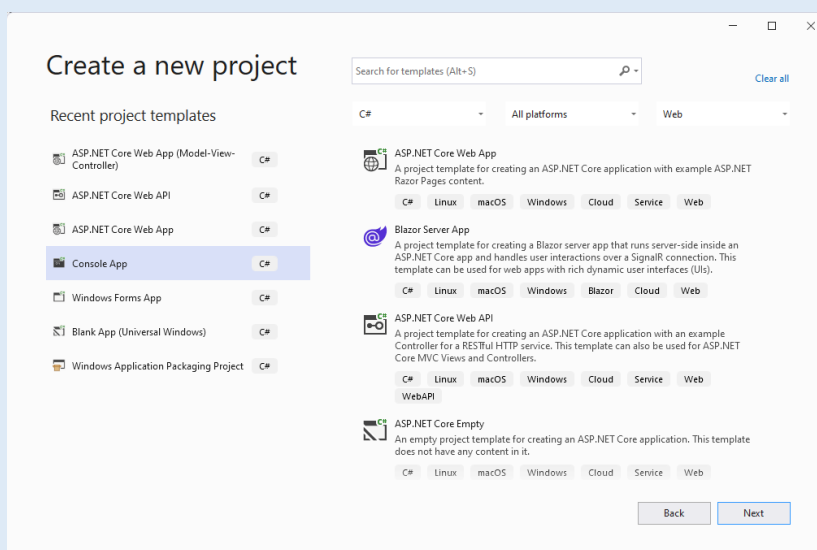
Pytania kontrolne:

1. W jaki sposób działa *inferencja typów* w języku C#?
2. Czym różnią się *właściwości* od *pól* w klasie?
3. Czy C# jest językiem *silnie typowanym*?

Zadanie 1.1. Tworzenie nowego projektu

Wariant dla Visual Studio 2022

Uruchom narzędzie Visual Studio 2022. Skorzystaj z opcji „Create a new project” na ekranie powitalnym i wybierz szablon „Console App”. Na kolejnym ekranie wybierz .NET w wersji 6.0 i nie zaznaczaj opcji nieużywania „top level statements”. Zapisz w wybranym przez siebie folderze pod wybraną przez siebie nazwą, np. „Lab1”.



Rys 1.1. Ekran wyboru szablonu projektu w Visual Studio 2022



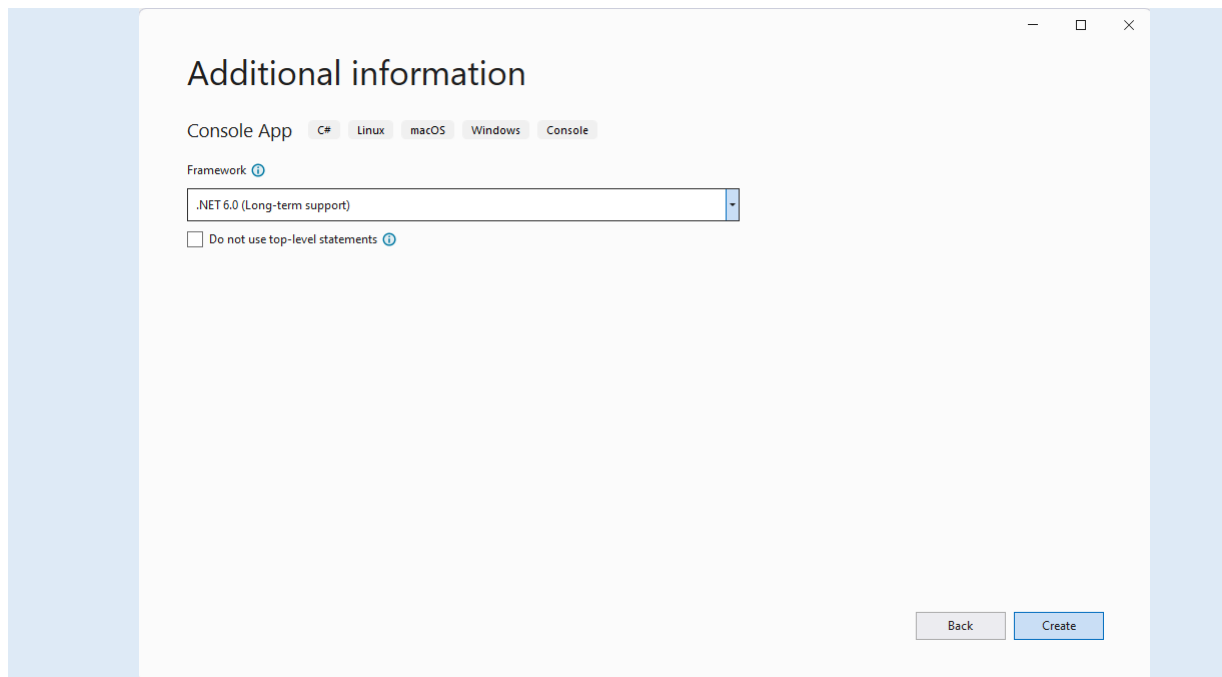
Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny





Rys 1.2. Wybór wersji platformy .NET

Wariant dla Visual Studio Code i narzędzi konsolowych

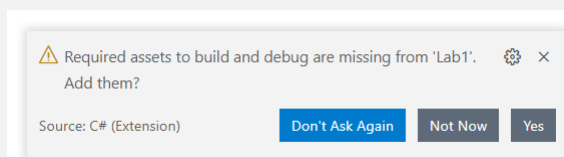
Uruchom narzędzie Terminal. Utwórz nowy folder, który będzie nazwą projektu, np. „Lab1” i przejdź do niego komendą `cd`, a następnie wydaj polecenie:

```
dotnet new console -f net6.0
```

Projekt zostanie utworzony w bieżącym folderze i będzie zgodny z nazwą folderu bieżącego. Otwórz projekt w edytorze kodu, korzystając z opcji „Open folder” lub wydając, w folderze projektu, komendę⁴:

```
code .
```

Zostaną wyświetlone komunikaty o konieczności doinstalowania brakujących elementów zależności projektu, wybierz „Yes” – spowoduje to dodanie do VSCode konfiguracji pozwalającej na uruchomienie projektu.



Rys 1.3. Komunikat o wymaganych elementach zależności projektu.

Zadanie 1.2. Uruchamianie projektu

Wygenerowana aplikacja składa się z jednego pliku zawierającego kod źródłowy – `Program.cs`. Począwszy od .NET 6.0 (oraz C# 10) wykorzystywany jest w niej mechanizm

⁴ W przypadku komputerów w laboratoriach użyj skrótu „Visual Studio Code (.NET)” na pulpicie i opcji „Open folder” zamiast tej komendy aby uruchomić wersję z dodanymi dodatkami dla .NET.

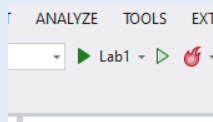
„top level statements”, który powoduje, że statyczna klasa **Program** oraz statyczna metoda **Main**, będąca punktem wejścia programu, nie są generowane jawnie, a zostaną wygenerowane automatycznie w momencie kompilacji programu.

Po utworzeniu szablonu program będzie zawierał tylko jedną instrukcję – wyświetlenie na ekranie konsoli komunikatu „Hello world”.

Spróbuj uruchomić nowy projekt i zaobserwować, czy w konsoli pojawi się odpowiedni komunikat.

Wariant dla Visual Studio 2022

Skorzystaj z opcji „Start Debugging” dostępnej w menu „Debug” oraz pod klawiszem F5, lub skorzystaj z przycisku na głównej belce programu.



Rys 1.3. Przycisk uruchamiania w trybie debugowania

Wariant dla Visual Studio Code i narzędzi konsolowych

W terminalu, w folderze głównym swojego projektu wydaj komendę:

```
dotnet run
```

Z kolei, aby uruchomić aplikację w trybie debugowania w edytorze VSCode możesz skorzystać z automatycznie wygenerowanego zadania – przechodząc do ekranu Run (Ctrl+Shift+D) i wybierając „Start Debugging” z menu Debug (F5), jej ekran wyjścia wtedy będzie znajdował się w oknie „output” edytora.

Zadanie 1.3. Program „FizzBuzz”

Korzystając z instrukcji takich jak pętle (**while**, **for**), instrukcji warunkowych (**if**, **else if**, **else**) oraz metod statycznych **Console.Write()** i **Console.WriteLine()** spróbuj przygotować program, który zadziała w następujący sposób:

Program ma wyświetlić po kolei liczby od 1 do 100 (włącznie), ale zamiast liczb podzielnych przez 3 ma wyświetlić słowo „Fizz”, zamiast podzielnych przez 5 słowo „Buzz”, a zamiast podzielnych przez 3 i 5 jednocześnie – słowo „FizzBuzz”. Ostateczny wynik programu powinien wyglądać tak, jak przedstawiono fragmentarycznie na rysunku 1.4.

Do sprawdzania podzielności możesz wykorzystać operator modulo, %.

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

Rys 1.4. Wynik działania programu FizzBuzz (fragment)

Zadanie 1.4. Gra „zgadnij liczbę”

Spróbuj przygotować teraz program, który wygeneruje liczbę, a następnie zadaniem człowieka będzie tę liczbę zgadnąć na podstawie wskazówek, które powiedzą że próba jest większa lub mniejsza od liczby, która została wygenerowana.

Do wygenerowania liczby losowej możesz skorzystać z klasy Random.

```
var rand = new Random();
var value = rand.Next(1, 101);
```

Możesz zauważyć, że w przykładowym kodzie wykorzystywane jest słowo kluczowe `var`, które tworzy zmienną typu domniemanego (następuje *inferencja typów*), działające podobnie do słowa kluczowego `auto` w języku C++.

Aby móc odczytać wartość wprowadzoną od użytkownika do ekranu konsoli oraz skonwertować do wartości liczbowej z łańcucha znaków możesz skorzystać z następującego kodu:

```
int guess = Convert.ToInt32(Console.ReadLine());
```

Przygotuj grę w taki sposób, aby użytkownik mógł wprowadzić swoją próbę i aby uzyskał informację czy jego lub jej próba jest większa czy mniejsza niż wylosowana wartość. W sytuacji „zgadnięcia” wartości losowej, wyświetl użytkownikowi komunikat.

Zadanie 1.5. Dodawanie liczenia liczby prób

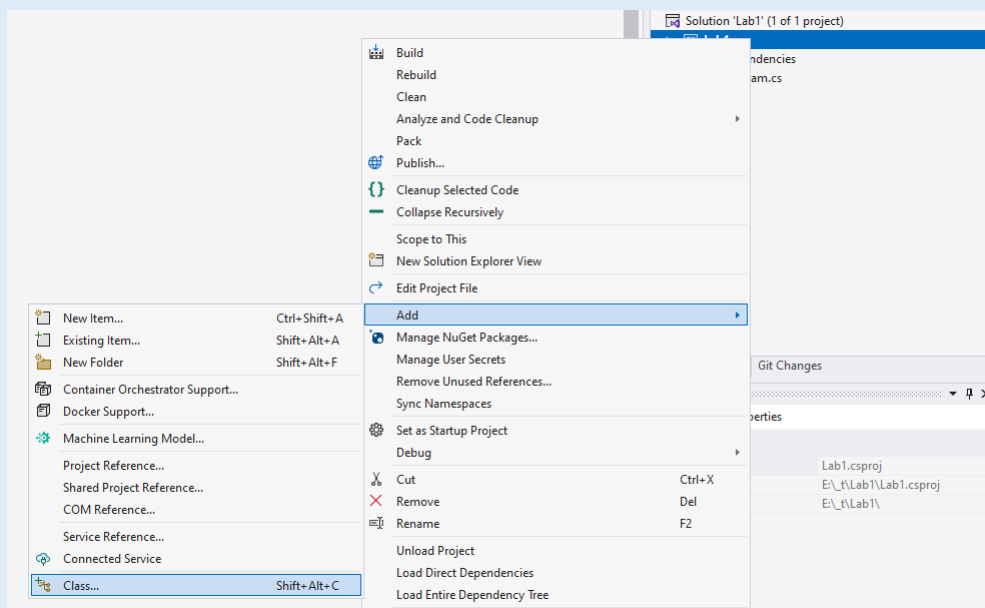
Spróbuj dodać do swojej gry mechanizm, który pozwoli na wyświetlenie ile prób potrzebne było użytkownikowi, aby wygrać grę.

Zadanie 1.6. Dodawanie mechanizmu „High Score”

Dodaj do swojego projektu nową klasę, która będzie modelem danych, służącym do przechowywania informacji o najlepszych wynikach. Zostanie tutaj wykorzystany mechanizm właściwości, które są automatycznie generowanymi metodami dla operacji pobierania i ustawiania danych dla pól, które mogą realizować dodatkowe operacje niż tylko prosty dostęp do pola, ale nie będzie to tutaj wykorzystywane.

Wariant dla Visual Studio 2022

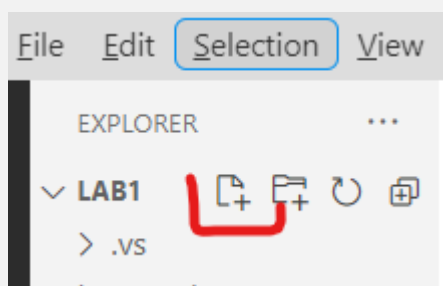
Kliknij prawym przyciskiem myszy na swoim projekcie i wybierz Add -> Class. Na kolejnym ekranie wpisz w polu „Name” nazwę pliku – nazwij nowy plik HighScore.cs.



Rys 1.5. Opcja dodawania nowej klasy

Wariant dla Visual Studio Code i narzędzi konsolowych

Kliknij na przycisk „New File” w swoim projekcie lub wybierz opcję z menu File. Nazwij nowy plik HighScore.cs.



Rys 1.6. Przycisk dodawania nowego pliku.

Wypełnij swój plik klasy zawartością – treścią klasy HighScore. Zawierać ona będzie 2 właściwości: Name oraz Trials, dla każdej z nich zostanie wygenerowany automatycznie getter oraz setter, każda z właściwości będzie publiczna, natomiast sama klasa będzie o widoczności internal, tj. będzie widoczna tylko w obrębie tego konkretnego projektu (a de facto tzw. *assembly*, czyli wygenerowanej jednostki kodu, pliku DLL). Nazwa przestrzeni nazw (namespace) powinna być zgodna z nazwą twojego projektu, np. Lab1.

```
namespace Lab1
{
    internal class HighScore
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
{  
    public string Name { get; set; }  
    public int Trials { get; set; }  
}  
}
```

W pliku `Program.cs`, po osiągnięciu wygranej przez gracza, zapytaj go lub ją o imię oraz zapisz tę informację i liczbę prób, która była potrzebna do wygranej do nowej zmiennej typu `HighScore`.

W C#, zwłaszcza korzystając z właściwości, nie ma konieczności tworzenia konstruktora do wypełniania obiektu danymi, ponieważ można skorzystać z mechanizmu listy inicjalizacyjnej, która będzie automatycznie wypełniać właściwości obiektu wartościami tuż po jego utworzeniu. Przykład wykorzystania tego mechanizmu zaprezentowano na listingu poniżej.

```
var hs = new HighScore { Name = name, Trials = trials };
```

Uwaga: Aby móc skorzystać z klasy `HighScore`, którą zagnieźdźono w przestrzeni nazw, należy dodać klauzulę `using` z nazwą przestrzeni nazw gdzieś na początku pliku `Program.cs`.

```
using Lab1;
```

Ostatnim krokiem będzie zapisanie tego obiektu do listy najlepszych wyników do pliku. Jeżeli jest to pierwsze uruchomienie gry, plik z najlepszymi wynikami może jednak nie istnieć, stąd warto dodać logikę, która to uwzględni.

Aby w miarę możliwości najłatwiejszy sposób przechowywać listę najlepszych wyników, zostanie wykorzystany format JSON (JavaScript Object Notation), tekstowa reprezentacja obiektów w postaci zestawu cech klucz-wartość.

Aby wczytać listę najlepszych wyników z pliku, możesz skorzystać z następującego kodu:

```
List<HighScore> highScores;  
const string FileName = "highscores.json";  
if (File.Exists(FileName))  
    highScores =  
    JsonSerializer.Deserialize<List<HighScore>>(File.ReadAllText(FileName));  
else  
    highScores = new List<HighScore>();
```

Musisz dodać jeszcze jeden element `using` gdzieś na szczycie pliku `Program.cs`:

```
using System.Text.Json;
```

Aby dodać swój wynik do listy najlepszych wyników, skorzystaj z metody `Add()`, niezbędne również będzie zserializowanie listy i zapisanie jej ponownie do pliku.

```
highScores.Add(hs);
```



```
File.WriteAllText(FileName,  
JsonSerializer.Serialize(highScores));
```

Wreszcie, aby wyświetlić listę najlepszych wyników wystarczy wykorzystać pętlę, w szczególności pętlę foreach:

```
foreach (var item in highScores)  
{  
    Console.WriteLine($"{item.Name} -- {item.Trials} prób");  
}
```

Ostatnia kwestia na którą możesz zwrócić uwagę to fakt, że lista ta jest wyświetlana w kolejności dodawania kolejnych wyników – w jaki sposób można zmodyfikować ten kod, aby wyświetlana była w kolejności od najlepszego (najmniej prób) do najgorszego wyniku?

Finalnie, program powinien działać w sposób przedstawiony na rysunku poniżej.

```
Wprowadz wartosc: 10  
Za malo!  
Wprowadz wartosc: 50  
Za malo!  
Wprowadz wartosc: 90  
Za duzo!  
Wprowadz wartosc: 69  
Za malo!  
Wprowadz wartosc: 88  
Za duzo!  
Wprowadz wartosc: 75  
Za malo!  
Wprowadz wartosc: 78  
Za malo!  
Wprowadz wartosc: 79  
Za malo!  
Wprowadz wartosc: 80  
Za malo!  
Wprowadz wartosc: 81  
Za malo!  
Wprowadz wartosc: 82  
Wygrana w 11 próbie!  
Podaj swoje imie: John  
Marcin -- 3 prób  
John -- 11 prób
```

Rys 1.7. Przykład działania programu wynikowego

LABORATORIUM 2. WYKORZYSTANIE MECHANIZMU LINQ DO PRZETWARZANIA DANYCH W APLIKACJI

Cel laboratorium:

Celem zajęć będzie wykorzystanie mechanizmu LINQ (Language Integrated Query) do przetwarzania danych z kolekcji w sposób funkcyjny.

Liczba punktów możliwych do uzyskania: 8 punktów

Zakres tematyczny zajęć:

Wykorzystanie metody fabrykującej i metody Random do tworzenia obiektów,
Przeciążanie funkcji w C#,
Wykorzystanie LINQ w C# do wybierania i sortowania danych,
Wykorzystanie metod asynchronicznych,
Wykorzystanie formatowania zależnego od regionu i języka,
Przygotowywanie projektu testów jednostkowych i pisanie prostych testów.

Pytania kontrolne:

1. W jaki sposób działają funkcje `map()` i `filter()` w programowaniu funkcyjnym?
2. Czym są metody asynchroniczne?
3. Co powinno być testowane w ramach testów jednostkowych?

Zadanie 2.1. Klasa modelu danych i metoda fabrykująca

Wygeneruj nowy projekt korzystając z szablonu aplikacji konsolowej. Następnie, dodaj do swojego projektu nowy plik `Fruit.cs`, który będzie zawierał klasę `Fruit`.

Klasa `Fruit` powinna zawierać trzy właściwości: `Name`, typu `string`, `IsSweet`, typu `bool` oraz `Price`, typu `double`.

W klasie `Fruit` utwórz publiczną statyczną metodę fabrykującą o nazwie `Create()`, która będzie generować i zwracać nowy obiekt typu `Fruit` o losowej nazwie spośród zbioru nazw, metoda powinna również ustawiać `IsSweet` oraz `Price` na wartości losowe. Możesz skorzystać z następującego podejścia:

```
public static Fruit Create()
{
    Random r = new Random();

    string[] names = new string[] { "Apple", "Banana",
    "Cherry", "Durian", "Edelberry", "Grape", "Jackfruit" };

    return new Fruit
    {
        Name = names[r.Next(names.Length)],
```



```
IsSweet = r.NextDouble() > 0.5,  
Price = r.NextDouble() * 10  
};  
}
```

Następnie, w pliku `Program.cs` wygeneruj 15 owoców i dodaj je do listy typu `List<Fruit>`. Spróbuj wyświetlić je po kolei w pętli `foreach`.

Zadanie 2.2. Przeciążanie konwersji do łańcucha tekstowego

Jak da się zauważyć, wynik działania programu z zadania 2.1 nie jest zadowalający, ponieważ wyświetlana jest nazwa klasy zamiast jej cech, jak przedstawiono na rysunku 2.1.

```
Lab2.Fruit  
Lab2.Fruit  
Lab2.Fruit  
Lab2.Fruit  
Lab2.Fruit  
Lab2.Fruit  
Lab2.Fruit  
Lab2.Fruit  
Lab2.Fruit  
Lab2.Fruit
```

Rys 2.1. Przykład działania zadania 2.1.

Z tego też powodu niezbędne będzie przeciążenie operacji `ToString()` wykonywanej na klasie `Fruit`. Dodaj do klasy `Fruit` publiczną metodę zwracającą typ `string` o nazwie `ToString()` – musisz jednak użyć także słowa kluczowego `override`, które wymusi polimorfizm (i jest niejako odpowiednikiem `@Override` z Javy czy też `override` z C++).

```
public override string ToString()
```

Zaimplementuj metodę `ToString()` w taki sposób, aby wyświetlała dane na przykład tak, jak przedstawiono na rysunku 2.2.

```
Fruit: Name=Banana, IsSweet=True, Price=9,91 zł
```

Rys 2.2. Formatowanie danych w metodzie `ToString()`

Do sformatowania wyświetlania ceny możesz wykorzystać metodę `ToString()` właściwości `Price` z parametrem `C2`, który wykorzysta format oparty o lokalnie aktywną walutę.

Zadanie 2.3. Wyświetlanie tylko wybranych elementów

Zmodyfikuj wyświetlanie elementów w taki sposób, aby wyświetlane były tylko elementy w których właściwość `IsSweet` jest ustawiona na `True`, posortowane względem ceny, malejąco.

Aby to zrobić, możesz skorzystać z LINQ. LINQ to zestaw funkcji (metod rozszerzających) dostarczanych przez bibliotekę `System.Linq`, która zawiera takie metody jak `Where()`, `Select()`, `OrderBy()` i inne, które pozwolą na manipulację danymi. Każda



z tych metod jako swojego parametru oczekuje funkcji (wykorzystywany jest paradygmat funkcyjny), np. w funkcji `Where()` oczekiwany jest predykat, tj. taka funkcja, która wykonuje klasyfikację elementu, a `Where()` zwraca z kolekcji tylko te elementy, dla których funkcja predykatu zwróciła `True`. Wynikiem działania każdej z tych operacji (projekcji, selekcji, sortowania) nie jest rzeczywiście istniejąca kolekcja, ale obiekt typu `IQueryable` lub `IEnumerable`, co pozwala na leniwą ewaluację – faktyczne operacje zostaną wykonane tylko wtedy, kiedy potrzeba – na przykład przed wykonaniem pętli `foreach`. Możesz zauważyć, że funkcje `Where()` czy `Select()` nazywają się podobnie do swoich odpowiedników w języku SQL, ale w zasadzie są odpowiednikami funkcji `map()` czy `filter()` z języków funkcyjnych.

Przykładowo, aby wybrać tylko takie elementy, w których ostatni element nazwy to „a” można zastosować taką funkcję:

```
var f = fruits.Where(x => x.Name.StartsWith("a"));
```

Zadanie 2.4. Metody asynchroniczne

Rzadko zdarza się, aby cały program działał całkowicie synchronicznie – mechanizmy zrównoleglania i asynchroniczności są bardzo popularne w wielu językach programowania. Dodaj do swojego projektu nową klasę, która będzie nazywać się `UsdCourse`. Zawierać ona będzie metodę, która komunikuje się z Internetem i pobiera aktualny kurs dolara. Zapytanie do zewnętrznego serwisu internetowego jest wykonywane w sposób asynchroniczny, stąd metoda `GetUsdCourseAsync()` ma wykorzystane słowo kluczowe `async`, zwraca `Task<float>`, czyli nie wartość typu `float`, ale „obietnicę” zwrócenia `float` w przyszłości.

```
class UsdCourse
{
    public static float Current = 0;

    public async static Task<float> GetUsdCourseAsync()
    {
        var wc = new HttpClient();
        var response = await
wc.GetAsync("http://www.nbp.pl/kursy/xml/LastA.xml");

        if (!response.IsSuccessStatusCode)
            throw new InvalidOperationException();

        System.Xml.XmlDocument xd = new
System.Xml.XmlDocument();

        xd.LoadXml(await
response.Content.ReadAsStringAsync());
    }
}
```




```
        foreach (System.Xml.XmlNode p in
xd.GetElementsByTagName("pozycja"))
        {
            if (p.NodeType == System.Xml.XmlNodeType.Element)
            {
                System.Xml.XmlElement pp =
(System.Xml.XmlElement)p;
                System.Xml.XmlElement w =
(System.Xml.XmlElement)pp.GetElementsByTagName("kod_waluty")[0];
                if (w.InnerText == "USD")
                {
                    return
Convert.ToSingle(pp.GetElementsByTagName("kurs_sredni")[0].InnerText);
                }
            }
        }
        throw new InvalidOperationException();
    }
}
```

Następnie, w swoim głównym programie `Program.cs` spróbuj uruchomić tę funkcję gdzieś na początku, za pomocą następującego kodu:

```
UsdCourse.Current = await UsdCourse.GetUsdCourseAsync();
```

Jak zauważysz, program nie działa już tak błyskawicznie – został na chwilę wstrzymany, aby pobrać dane z Internetu. Wstrzymanie to jednak nie jest blokujące – gdyby było dostępne GUI, program by się nie „zamroził” – co jest realizowane dzięki mechanizmowi `async/await` i słowu kluczowemu `await`.

Kod ten ustawia statyczną składową klasy `UsdCourse`, z której będziemy korzystać dalej w programie.

Zadanie 2.5. Właściwości obliczane na bieżąco

Dodaj do swojej klasy `Fruit` właściwość `UsdPrice` – będzie ona obliczała cenę w dolarach na podstawie ceny bazowej. Z uwagi na to, że właściwość ta nie będzie potrzebowała settera, można ją znacząco uprościć i zapisać w postaci funkcji strzałkowej:

```
public double UsdPrice => Price / UsdCourse.Current;
```

Spróbuj dodać do swojej implementacji `ToString()` wyświetlanie ceny w złotych oraz w dolarach.

Zadanie 2.6. Formatowanie zależne od kultury

Kultura („*locale*”) to zestaw informacji, które opisują w jaki sposób należy wyświetlać format daty, godziny, czy też liczb i walut w zależności od regionu świata i używanego języka. Jak prawdopodobnie wiesz, w Polsce waluta wyświetlana jest w formacie „0,99 zł”, natomiast w USA – „\$0.99”. Należy dodać do naszego programu mechanizm, aby cena bazowa (`Price`) była wyświetlana poprzez zwykłe formatowanie walut, a cena w dolarach w sposób odpowiedni dla USA.

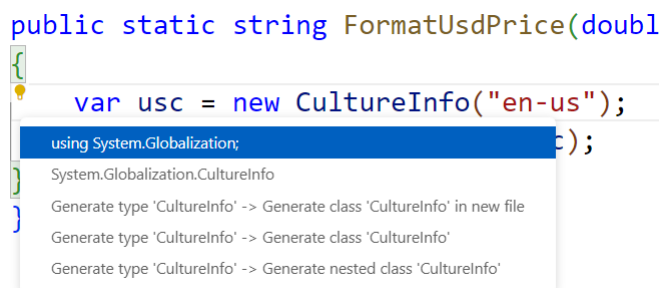
Dodaj do swojego programu nową klasę, `MyFormatter`.

Klasa ta będzie zawierać jedną metodę statyczną, `FormatUsdPrice`, która będzie przyjmować `double` i zwracać `string`.

Możesz skorzystać z następującego przykładu:

```
public static string FormatUsdPrice(double price)
{
    var usc = new CultureInfo("en-us");
    return price.ToString("C2", usc);
}
```

Po wklejeniu tego kodu możesz dostać informację, że edytor nie rozpoznaje typu `CultureInfo` – należy dodać odpowiedni `using`. Wystarczy kliknąć ikonkę „żarówki” (zarówno w Visual Studio, jak i VSCode) i z rozwijanego menu wybrać odpowiednią opcję dodania `using`.



```
public static string FormatUsdPrice(double price)
{
    var usc = new CultureInfo("en-us");
    using System.Globalization;
    return price.ToString("C2", usc);
}
```

Rys 2.3. Wykorzystanie opcji rozwiązywania problemów w kodzie

Zmodyfikuj metodę `ToString()` klasy `Fruit` w taki sposób, aby wykorzystywała klasę `MyFormatter`. Program powinien finalnie wyświetlać wyniki podobne do zaprezentowanych:

```
Fruit: Name=Banana, IsSweet=True, Price=9,91 zł, UsdPrice=$2.07
Fruit: Name=Jackfruit, IsSweet=True, Price=6,52 zł, UsdPrice=$1.36
Fruit: Name=Edelberry, IsSweet=True, Price=5,83 zł, UsdPrice=$1.22
Fruit: Name=Apple, IsSweet=True, Price=5,00 zł, UsdPrice=$1.04
Fruit: Name=Durian, IsSweet=True, Price=4,35 zł, UsdPrice=$0.91
Fruit: Name=Jackfruit, IsSweet=True, Price=1,73 zł, UsdPrice=$0.36
```

Rys 2.4. Przykład działania programu

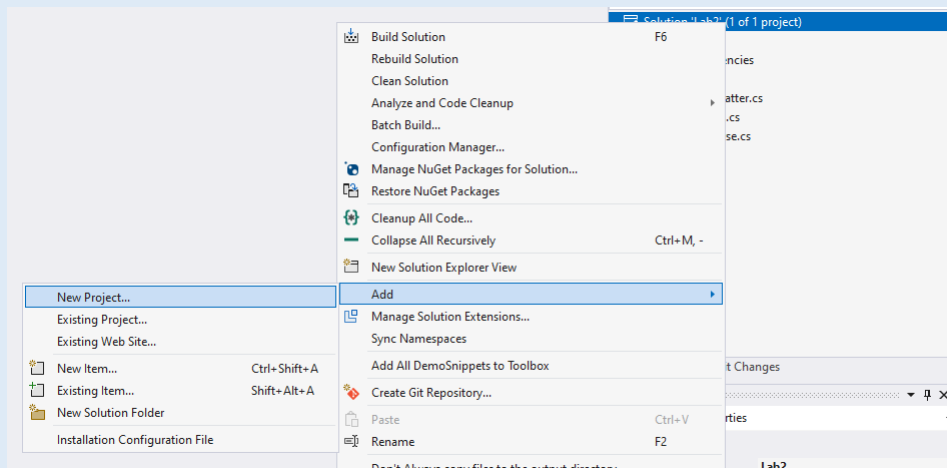


Zadanie 2.7. Testy jednostkowe

Samodzielne, ręczne, sprawdzanie działania programu często jest uciążliwe, monotonne i nieefektywne. Z tego też powodu stosuje się między innymi testy automatyczne, w szczególności testy jednostkowe. W tym zadaniu spróbujemy przygotować test jednostkowy, który sprawdzi, czy klasa `MyFormatter` i jej metoda `FormatUsdPrice()` działa odpowiednio.

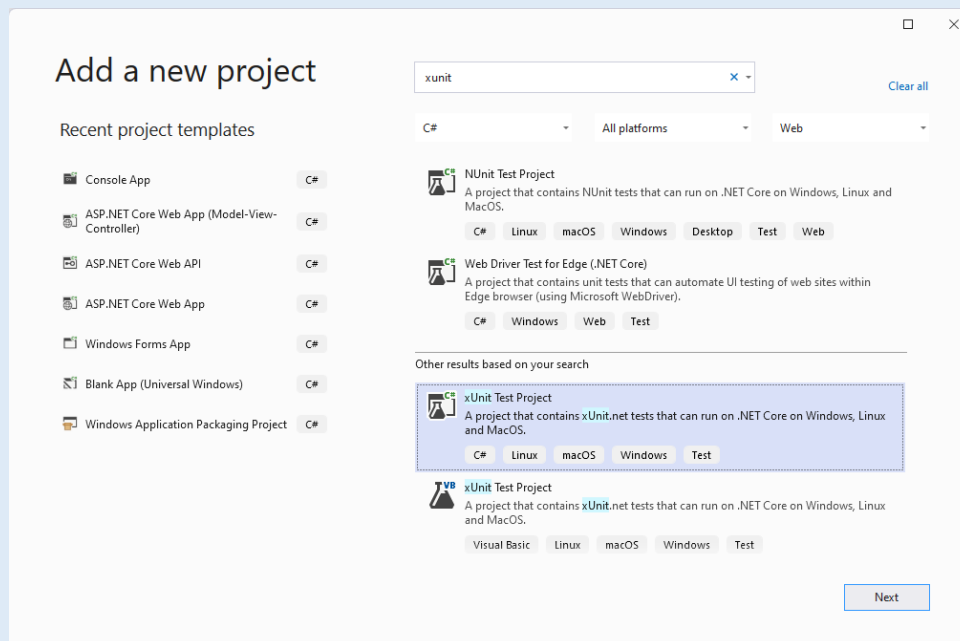
Wariant dla Visual Studio

Kliknij prawym przyciskiem myszy na solucji (nie na projekcie!) i wybierz opcję `Add -> New Project`.



Rys 2.5. Dodawanie nowego projektu w tej samej solucji

W następnym oknie wybierz szablon „xUnit Test Project” i nazwij go tak jak projekt główny, z przyrostkiem `.Test`, np. „Lab2.Test”.



Rys 2.6. Wybór projektu testów xUnit



Fundusze Europejskie
Wiedza Edukacja Rozwój

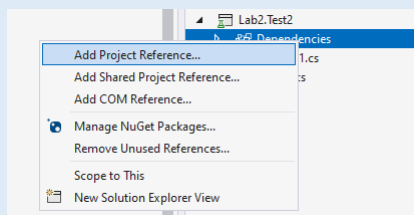


Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny

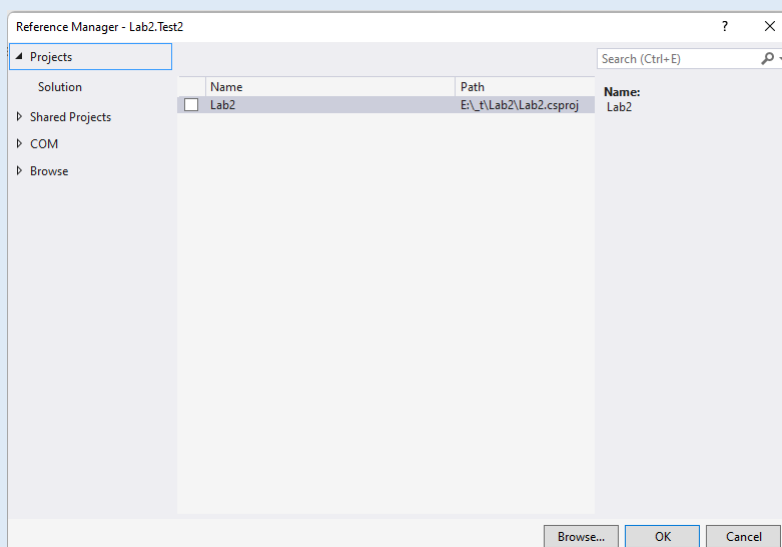


Następnie, dodaj powiązanie między projektem testów i projektem głównym – kliknij prawym przyciskiem myszy na „Dependencies” w projekcie testów i wybierz opcję „Add Project Reference”.



Rys 2.7. Dodawanie referencji do projektu

W następnym oknie zaznacz projekt główny, w tym przykładzie jest to Lab2.



Rys 2.8. Dodawanie referencji do projektu (2)

Wariant dla Visual Studio Code i narzędzi konsolowych

Utwórz nowy projekt w folderze zawierającym folder twojego projektu głównego korzystając z szablonu projektu xUnit, za pomocą komendy:

```
dotnet new xunit -o Lab2.Test -f net6.0
```

W miejsce Lab2.Test wpisz nazwę stworzoną na podstawie nazwy twojego głównego projektu z dodatkiem „Test”.

.NET wygeneruje nowy folder i projekt, ale nie zbuduje powiązania pomiędzy projektami. Przejdź do folderu z projektem testów i wydaj komendę:

```
dotnet add reference ../Lab2
```

Spowoduje to dodanie odwołania pomiędzy projektem testów i projektem Lab2 znajdującym się w tym samym folderze głównym (zmień Lab2 aby dostosować do utworzonego przez siebie projektu).

Po wygenerowaniu projektu testów otwórz go w swoim edytorze.

W projekcie głównym zmodyfikuj klasę MyFormatter, dodając do klasy słowo kluczowe public – pozwoli to na skorzystanie z niej w projekcie testów.



W projekcie testów, w pliku `UnitTest1.cs`, możliwe będzie dodanie testu jednostkowego. Przygotujemy prosty test, który sprawdzi, czy funkcja zwraca na podstawie liczby łańcuch, który zaczyna się znakiem „\$” i zawiera znak kropki.

Możesz skorzystać z następującego kodu:

```
[Fact]
public void
FormatUsdPrice_ProperFormat_ShouldReturnProperString()
{
    var data = 0.05;

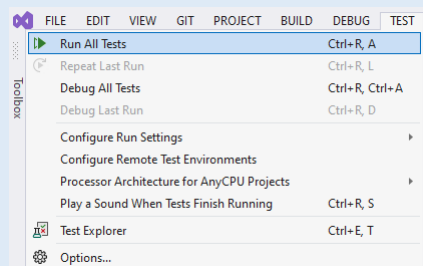
    var result = MyFormatter.FormatUsdPrice(data);

    Assert.StartsWith("$", result);
    Assert.Contains(".", result);
}
```

Spróbuj uruchomić ten test i sprawdź jego wynik.

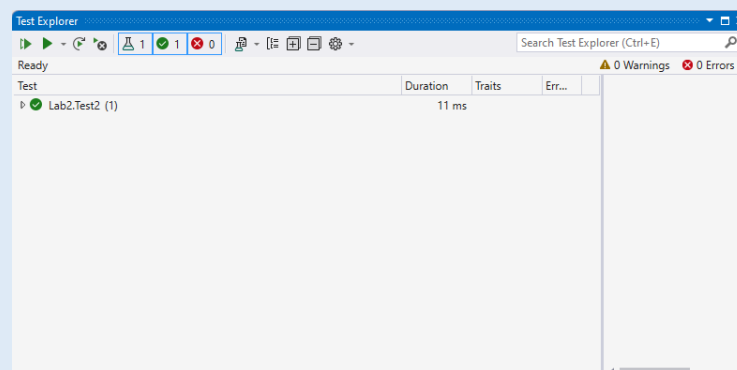
Wariant dla Visual Studio

Z menu Test wybierz opcję Run All Tests.



Rys 2.9. Opcja uruchamiania testów

Wyniki testów powinny pojawić się w nowym oknie, Test Explorer.



Rys 2.10. Okno eksploratora testów.

Wariant dla Visual Studio Code i narzędzi konsolowych

Przejdź w terminalu do folderu zawierającego projekt testów i wydaj komendę:

```
dotnet test
```

Po chwili powinny uruchomić się testy i pojawić ich rezultat.

```
E:\_t\Lab2.Test>dotnet test
Determining projects to restore...
All projects are up-to-date for restore.
E:\_t\Lab2\Fruit.cs(7,23): warning CS8618: Non-nullable property 'Name' must contain a non-null value when exiting constructor. Consider declaring the property as nullable. [E:\_t\Lab2\Lab2.csproj]
E:\_t\Lab2\UsdCourse.cs(22,47): warning CS8600: Converting null literal or possible null value to non-nullable type. [E:\_t\Lab2\Lab2.csproj]
E:\_t\Lab2\UsdCourse.cs(23,25): warning CS8602: Dereference of a possibly null reference. [E:\_t\Lab2\Lab2.csproj]
E:\_t\Lab2\UsdCourse.cs(25,49): warning CS8602: Dereference of a possibly null reference. [E:\_t\Lab2\Lab2.csproj]
Lab2 -> E:\_t\Lab2\bin\Debug\net6.0\Lab2.dll
Lab2.Test -> E:\_t\Lab2.Test\bin\Debug\net6.0\Lab2.Test.dll
Test run for E:\_t\Lab2.Test\bin\Debug\net6.0\Lab2.Test.dll (.NETCoreApp,Version=v6.0)
Microsoft (R) Test Execution Command Line Tool Version 17.2.0 (x64)
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed:    0, Passed:    1, Skipped:    0, Total:    1, Duration: < 1 ms - Lab2.Test.dll (net6.0)
```

Rys 2.11. Wynik testów w konsoli

Zadanie 2.8. Udoskonalenie testów

Zmodyfikuj test w taki sposób, aby sprawdzić, czy zawiera dokładnie tę wartość, która została podana do testowania, w dokładnie określonym formacie, np. jeżeli wartością testową było 0.05 to czy wynik zaczyna się od \$0, potem zawiera konkretnie kropkę, a potem zawiera 05.

LABORATORIUM 3. TWORZENIE APLIKACJI INTERNETOWEJ W ARCHITEKTURZE MVC

Cel laboratorium:

Celem zajęć jest zapoznanie się z tworzeniem aplikacji ASP.NET Core w architekturze MVC, dodawanie nowych modeli, metod kontrolera i widoków za pomocą automatycznych generatorów.

Liczba punktów możliwych do uzyskania: 9 punktów

Zakres tematyczny zajęć:

Generowanie nowego projektu MVC,
Wykorzystanie składni Razor,
Przekazywanie danych do widoku,
Wykorzystanie mechanizmu Dependency Injection,
Generowanie silnie typowanych widoków,
Dodawanie obsługi prostych formularzy opartych o modele.

Pytania kontrolne:

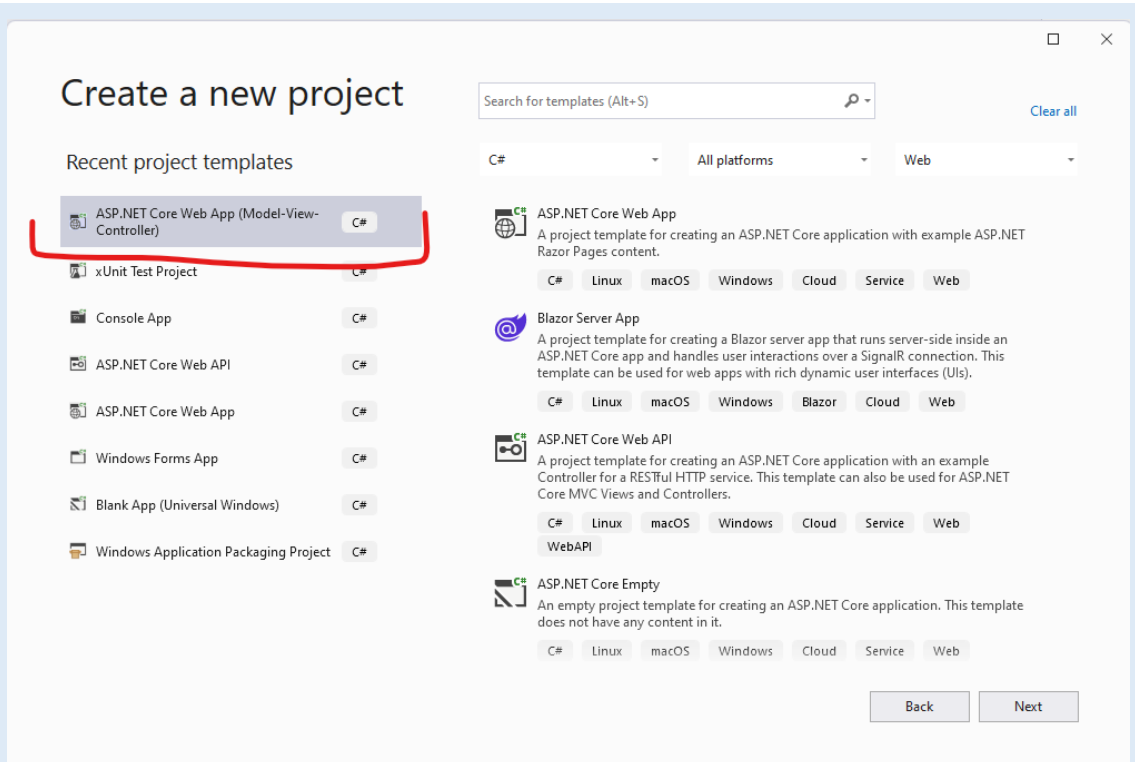
1. Na czym polega architektura Model-Widok-Kontroler (MVC)?
2. Czym są akcje kontrolera w ASP.NET Core MVC?
3. Czym charakteryzuje się składnia widoków, Razor?

Zadanie 3.1. Generowanie nowego projektu MVC

W tym laboratorium będzie budowana nowa aplikacja ASP.NET Core MVC – MVC służy do budowania aplikacji w architekturze Model-View-Controller, które generują kod HTML po stronie serwera i przesyłają do przeglądarki internetowej.

Wariant dla Visual Studio

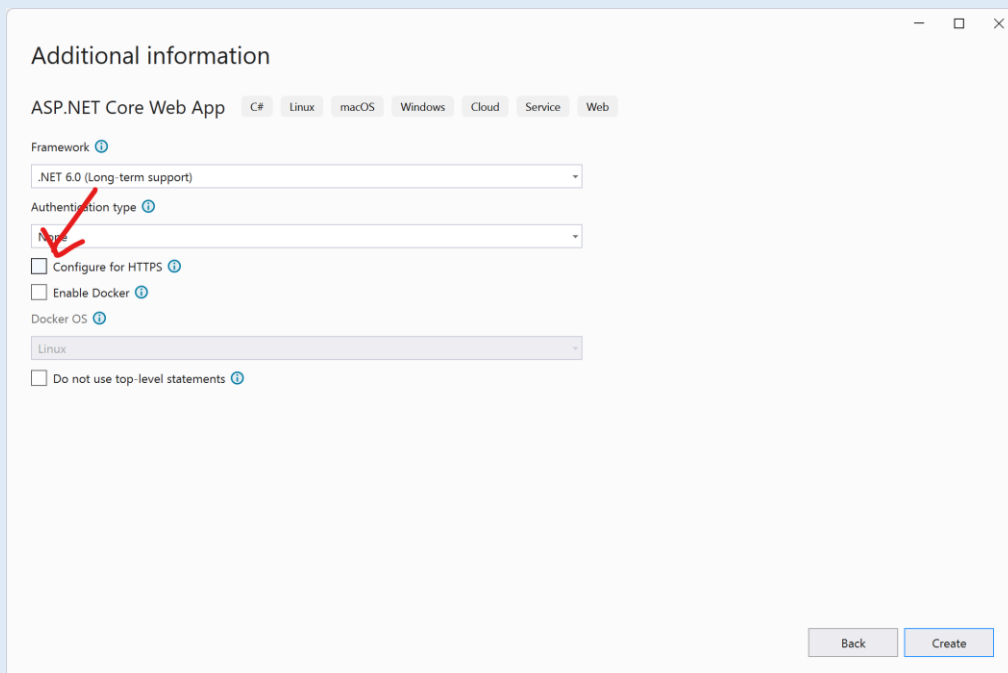
Utwórz nowy projekt, korzystając z szablonu „ASP.NET Core Web App (Model-View-Controller)” (uważaj, aby nie wybrać projektu „ASP.NET Core Web App”!)



Rys 3.1. Wybór szablonu aplikacji MVC

Nazwij go w wybrany przez siebie sposób (np. „Lab3”) i wybierz opcję „Next”.

Na kolejnym ekranie, pamiętaj, aby **odznaczyć** opcję „Configure for HTTPS” (projekty na laboratoriach nie będą korzystały z HTTPS aby nie musieć instalować tymczasowych certyfikatów).



Wariant dla Visual Studio Code oraz narzędzi konsolowych

Utwórz nowy folder w którym będzie znajdował się projekt, pamiętając, że, tak samo jak dotychczas, nazwa projektu będzie zgodna z nazwą folderu. Wygeneruj nowy projekt typu MVC korzystając z polecenia:

```
dotnet new mvc --no-https -f net6.0
```

Spróbuj uruchomić swój projekt wybierając przycisk „Play” lub komendę „dotnet run”. Powinna uruchomić się przeglądarka internetowa na głównej stronie twojego projektu, który uruchomi się za pośrednictwem wbudowanego serwera Kestrel na losowym porcie.

Zadanie 3.2. Silnik widoków Razor

Otwórz plik Views/Home/Index.cshtml. Jak widzisz, składa się on z HTML-a przemieszanego ze składnią Razor, służącą do wyświetlania elementów pochodzących z kodu C# za pomocą znaku @, na przykład:

```
@DateTime.Now
```

Taki kawałek kodu spowoduje odwołanie się do właściwości Now klasy statycznej DateTime, wywołanie na niej metody ToString() i umieszczenie wyniku tej operacji w wynikowym, generowanym kodzie HTML. Możliwe jest również ręczne wykonanie ToString(): @DateTime.Now.ToString("yyyy"). Spowoduje to wyświetlenie tylko aktualnego roku (w wersji czterocyfrowej).

Spróbuj wykorzystać składnię Razor do osiągnięcia następującego efektu (tzw. formatu daty długiej) dla aktualnej daty zamiast tekstu wprowadzającego.

Welcome

niedziela, 17 lipca 2022

Rys. 3.2. Przykład oczekiwanego działania aplikacji

Zmodyfikuj następnie plik Views/Home/Shared/_Layout.cshtml. Plik ten zawiera główny układ treści aplikacji. W stopce znajduje się aktualny rok zapisywany podczas generowania projektu – spróbuj zmodyfikować to w taki sposób, aby zawsze rok był aktualny. Zwróć uwagę, że zmiana w pliku _Layout jest widoczna we wszystkich już obecnych stronach aplikacji, np. na stronie polityki prywatności.

Zadanie 3.3. Przekazywanie danych pomiędzy kontrolerem, a widokiem

Otwórz plik Controllers/HomeController.cs. W metodzie Index() dodaj następujący fragment kodu:

```
Random r = new Random();  
ViewData["random"] = r.NextDouble();
```

Spowoduje to zapisanie do mechanizmu ViewData losowej liczby. Spróbujemy ją teraz wyświetlić – w pliku Views/Home/Index.cshtml dodaj fragment kodu, który będzie wyświetlał tę liczbę, na przykład w następujący sposób:



```
<p>@ViewData["random"]</p>
```

W składni Razor możesz skorzystać z instrukcji warunkowych i innych instrukcji sterujących.

Spróbuj opracować taki kod, który wyświetli liczbę losową z czerwonym tłem, jeśli liczba będzie większa niż 0,5. Zwróć uwagę, że dane przechodzące przez `ViewData` wymagają rzutowania, gdyż domyślnie są typu `object`.

Welcome

niedziela, 17 lipca 2022

0.5078717542319291

Rys 3.3. Przykład oczekiwanego działania aplikacji

Zadanie 3.4. Silnie typowane modele danych

Nie wszystkie dane powinny być przekazywane przez `ViewData` – do przekazywania silnie typowanych danych o które model jest oparty wykorzystuje się nieco inną konstrukcję – modele oparte o typ. Ich ręczne tworzenie może być uciążliwe, więc zastosujemy tutaj autogenerowanie (scaffolding), wymagane jednak będzie najpierw utworzenie modelu danych.

W folderze `Models` dodaj nową klasę, która powinna być utworzona w przestrzeni nazw z przyrostkiem `.Models` (np. `Lab3.Models`). Klasa ta będzie nazywać się `Contact` i będzie przechowywać dane osobowe.

```
using System.ComponentModel;

namespace Lab3.Models
{
    public class Contact
    {
        [DisplayName("Identyfikator")]
        public int Id { get; set; }
    }
}
```

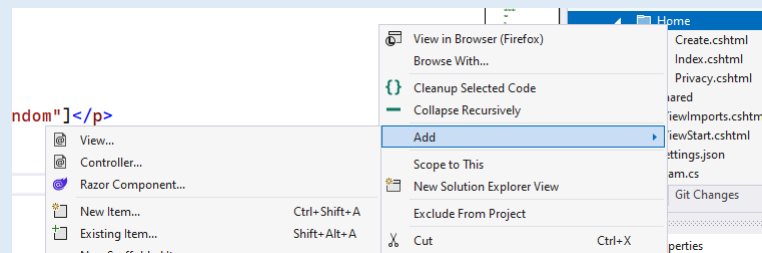
Oprócz właściwości `Id`, dodaj kolejne: `Name`, `Surname`, `Email`, `City` oraz `PhoneNumber`, wszystkie typu `string`. Każdą z nich opatrz także atrybutem `DisplayName`, który ją nazwie w sposób zrozumiały dla użytkownika – np. „Imię”, „Nazwisko” i tak dalej. Automatyczny generator wykorzysta ten mechanizm, aby wygenerować tabelę i formularze w odpowiedni sposób.

Usuń plik `Views/Home/Index.cshtml`, ponieważ będziemy generować go z wykorzystaniem generatora.



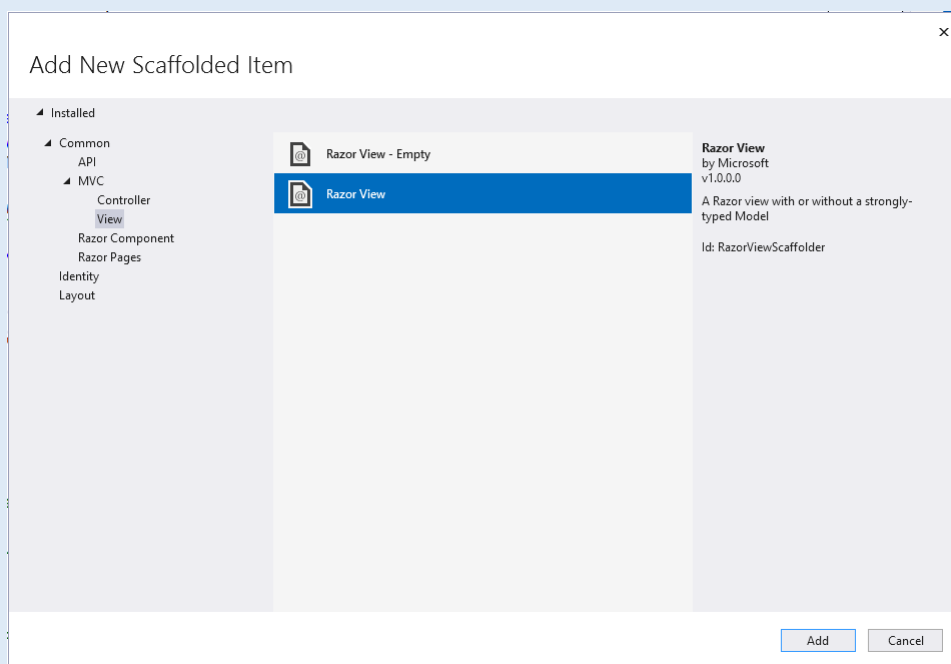
Wariant dla Visual Studio

Kliknij prawym klawiszem myszy na folderze Views/Home i wybierz z menu kontekstowego opcję Add -> View.



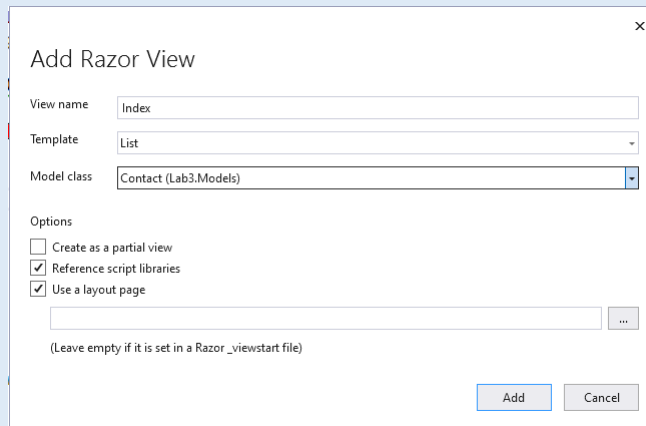
Rys 3.4. Opcja dodawania nowego widoku

Następnie, na kolejnym ekranie wybierz opcję „Razor View”.



Rys 3.5. Wybór widoku do dodania

A na kolejnym ekranie wpisz nazwę widoku „Index”, wybierz szablon „List” i wybierz swoją klasę modelu w polu „model class”. Zaznacz pole „use a layout page”, jeśli nie jest zaznaczone.



Rys 3.6. Wybór typu generowanego modelu

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Przejdź do głównego folderu swojego projektu i wydaj komendę:

```
dotnet add package  
Microsoft.VisualStudio.Web.CodeGeneration.Design --version 6.0
```

Spowoduje to dodanie do projektu biblioteki służącej do generowania kodu z internetowego serwera NuGet. Teraz, możesz już skorzystać z automatycznego generatora kodu. Przejdź do folderu głównego swojego projektu i wydaj komendę:

```
dotnet aspnet-codegenerator view Index List -m Contact -outDir  
Views/Home --useDefaultLayout
```

Wygeneruje ona szablon korzystając z szablonu List o nazwie Index, w oparciu o model Contact i zapisze do pliku Views/Home/Index.cshtml.

Zadanie 3.5. Repozytorium danych

Dane do prezentowania w widoku w typowych aplikacjach pobierane są z innych systemów informatycznych lub przede wszystkim z systemów baz danych. W tym laboratorium zbudujemy fikcyjne źródło danych oparte o listę. W folderze Models utwórz nową klasę, PhoneBookService i wypełnij ją następującą zawartością:

```
public class PhoneBookService  
{  
    private List<Contact> contacts;  
  
    public PhoneBookService()  
    {  
        contacts = new List<Contact>()  
        {  
            new Contact
```

```
{
    Id = 1,
    Name = "Jan",
    Surname = "Kowalski",
    City = "Lublin",
    Email = "jan@kowalski.com",
    PhoneNumber = "123456789"
},
new Contact
{
    Id = 2,
    Name = "Adam",
    Surname = "Paździoch",
    City = "Zamość",
    Email = "adam@wp.com",
    PhoneNumber = "987654321"
},
new Contact
{
    Id = 3,
    Name = "Mariusz",
    Surname = "Nowak",
    City = "Warszawa",
    Email = "mariusz.nowak@gmail.com",
    PhoneNumber = "837264917"
},
new Contact
{
    Id = 4,
    Name = "Jarosław",
    Surname = "Kamiński",
    City = "Zamość",
    Email = "kaminskij@test.pl",
    PhoneNumber = "83102849"
```



```
    },  
    new Contact  
    {  
        Id = 5,  
        Name = "James",  
        Surname = "Kaczmarek",  
        City = "Lublin",  
        Email = "kaczmarek.james@ir.com",  
        PhoneNumber = "027384716"  
    },  
    new Contact  
    {  
        Id = 6,  
        Name = "Rafał",  
        Surname = "Nowicki",  
        City = "Kraków",  
        Email = "rafalnowicki@wp.com",  
        PhoneNumber = "627193482"  
    },  
    new Contact  
    {  
        Id = 7,  
        Name = "Kamil",  
        Surname = "Malinowski",  
        City = "Poznań",  
        Email = "kamil.m@wp.com",  
        PhoneNumber = "038462817"  
    }  
};  
  
}  
  
public IEnumerable<Contact> GetContacts()  
{  
    return contacts;  
}
```



```
}
}
```

Klasa ta udostępnia listę kontaktów wypełnianą w swoim konstruktorze za pośrednictwem metody `GetContacts()`. Z uwagi na to, że lista kontaktów jest zwykłą listą generyczną, obiekt klasy `PhoneBookService` nie powinien zmieniać się przez cały czas życia obiektu, powinien być także jeden wspólny dla wszystkich żądań HTTP i jeden dla wszystkich obiektów całej aplikacji – powinniśmy zastosować tutaj wzorzec projektowy *singleton*. Aby uzyskać singleton w sposób automatyczny, w pliku `Program.cs`, przed linią `var app = builder.Build();` dodaj:

```
builder.Services.AddSingleton<PhoneBookService>();
```

Spowoduje to, że obiekt klasy `PhoneBookService` zostanie utworzony raz i zostanie „wstrzyknięty” przez mechanizm *Dependency Injection* to wszystkich klas, które go będą oczekiwały.

Zadanie 3.6. Wstrzykiwanie zależności i prezentacja danych

Aby zdefiniować, że kontroler `HomeController` będzie uzyskiwał dostęp do automatycznie tworzonego obiektu klasy `PhoneBookService` należy zmodyfikować jego konstruktor – dodaj, aby oczekiwany był obiekt klasy `PhoneBookService` i zapisz przekazany obiekt do prywatnego pola kontrolera.

```
private readonly PhoneBookService _phoneBook;

public HomeController(ILogger<HomeController> logger,
    PhoneBookService phoneBook)
{
    _logger = logger;
    _phoneBook = phoneBook;
}
```

Używany jest tutaj modyfikator `readonly` – powoduje on, że pole może być ustawione tylko w konstruktorze i nie może być potem zmodyfikowane. Ostatnim krokiem będzie przekazanie kontaktów z tego symulowanego repozytorium danych do widoku. Widok został automatycznie wygenerowany w taki sposób, aby opierać się o listę elementów – ponieważ został wykorzystany szablon listy.

Zmodyfikuj zatem metodę `Index()` (metodę o tej samej nazwie jak generowany wcześniej widok), aby wyglądała w następujący sposób:

```
public IActionResult Index()
{
    return View(_phoneBook.GetContacts());
}
```

Teraz spróbuj uruchomić swoją aplikację i powinna ona wyglądać w sposób następujący:

Lab3 Home Privacy

Index

[Create New](#)

Identyfikator	Imię	Adres e-mail	Nazwisko	Miasto	Numer telefonu	
1	Jan	jan@kowalski.com	Kowalski	Lublin	123456789	Edit Details Delete
2	Adam	adam@wp.com	Paździoch	Zamość	987654321	Edit Details Delete
3	Mariusz	mariusz.nowak@gmail.com	Nowak	Warszawa	837264917	Edit Details Delete
4	Jarosław	kaminskij@test.pl	Kamiński	Zamość	83102849	Edit Details Delete
5	James	kaczmarek.james@ir.com	Kaczmarek	Lublin	027384716	Edit Details Delete
6	Rafał	rafal.nowicki@wp.com	Nowicki	Kraków	627193482	Edit Details Delete
7	Kamil	kamil.m@wp.com	Malinowski	Poznań	038462817	Edit Details Delete

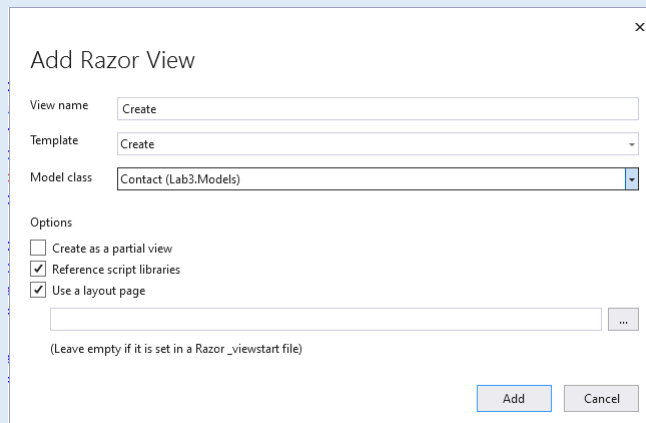
Rys 3.7. Wygląd gotowej aplikacji

Zadanie 3.7. Dodawanie nowych elementów

Kliknięcie na link „Create new” powinno przechodzić do pola dodawania nowego obiektu do bazy danych. Aby to osiągnąć, musisz wygenerować widok formularza dodawania nowego elementu.

Wariant dla Visual Studio

Kliknij prawym przyciskiem myszy na folderze Views/Home i wybierz ponownie opcję Add -> View, a następnie Razor View, tym razem jednak z listy szablonów wybierając „Create”, jak na rysunku 3.8.



Rys 3.8. Dodawanie widoku tworzenia nowego obiektu

Nazwij go „Create” i kliknij przycisk „Add”, co spowoduje wygenerowanie nowego pliku w Views/Home/Create.cshtml.

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Przejdź do głównego folderu swojego projektu i wydaj komendę:

```
dotnet aspnet-codegenerator view Create Create -m Contact -
outDir Views/Home --useDefaultLayout
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Wygeneruje ona nowy plik w folderze Views/Home o nazwie Create.cshtml, zgodny z szablonem Create, tj. szablonem służącym do tworzenia formularzy dodawania nowych elementów do systemu.

Zmodyfikuj widok Create.cshtml w taki sposób, aby usunąć z niego pole „Identyfikator” – identyfikator będzie generowany automatycznie w klasie PhoneBookService, użytkownik nie powinien mieć na niego wpływu.

Teraz w pliku HomeController.cs należy dodać dwie akcje – jedna będzie odpowiadała za wyświetlanie formularza dodawania obiektu, a druga za odbiór danych z tego formularza.

Należy dodać dwie metody, obydwie będą nazywały się Create, jednak jedna z nich będzie posiadała atrybut [HttpPost], który oznacza, że uruchamia się tylko w reakcji na żądanie HTTP typu POST, np. przesłanie danych z formularza. Niezbędne teraz będzie zmapowanie pól formularza na obiekt typu Contact – na szczęście można do tego wykorzystać automatyczny Model Binder, tak jak w przykładzie poniżej:

```
public IActionResult Create()
{
    return View();
}

[HttpPost]
public IActionResult Create(Contact contact)
{
    if (ModelState.IsValid)
    {
        _phoneBook.Add(contact);
        return RedirectToAction("Index");
    }

    return View();
}
```

W przykładzie tym została również wykorzystana metoda RedirectToAction(), która zwraca do przeglądarki internetowej nie widok, ale nagłówek HTTP oznaczający przekierowanie – w tym przypadku jest to przekierowanie do innej akcji i odpowiedni URL zostanie wygenerowany automatycznie. ModelState.IsValid zwraca, czy udało się poprawnie zmapować dane z formularza do klasy modelu oraz czy są one poprawne (np. czy wszystkie wymagane pola zostały zaznaczone) i jeśli nie, formularz zostanie wyświetlony poprawnie.

Ostatnim krokiem będzie dodanie do klasy PhoneBookService metody Add(), która przyjmie obiekt typu Contact i go doda do listy. Możesz wykorzystać następujący kod:



```
public void Add(Contact contact)
{
    contact.Id = contacts.Max(x => x.Id) + 1;
    contacts.Add(contact);
}
```

Generuje on identyfikator na podstawie największego obecnego identyfikatora +1 i dodaje element do listy – nie jest to podejście bezpieczne np. w sytuacji wielowątkowej, ale będzie wystarczające dla prostego testu.

Spróbuj uruchomić swoją aplikację i dodać nowy kontakt.

Zadanie 3.8. Usuwanie kontaktów

Wygenerowany widok listy zawiera odnośniki do akcji edycji, szczegółów i usuwania obiektów realizowane za pomocą metody `ActionLink()`.

```
@Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey
*/ })
```

Metoda ta automatycznie tworzy odnośniki w taki sposób, aby prowadziły do odpowiedniej akcji, zgodnie z zadeklarowanym w aplikacji routingiem i systemem URL-i. Niezbędne jest jednak przekazanie parametrów w taki sposób, aby było rozpoznane, który z elementów jest identyfikatorem. Typowym podejściem jest przygotowanie metod w taki sposób, aby były powiązane z adresami URL w postaci <https://example.com/Home/Delete/3>, gdzie 3 to właśnie identyfikator elementu do usunięcia. Usuń wszystkie odnośniki poza jednym, odpowiedzialnym za usuwanie elementów i wykorzystaj trzeci parametr metody `ActionLink()` do przekazania identyfikatora kontaktu.

```
@Html.ActionLink("Delete", "Delete", new { id = item.Id })
```

Możesz zwrócić uwagę, że wykorzystywany jest tutaj obiekt klasy anonimowej.

Kolejnym krokiem będzie przygotowanie akcji, która zareaguje na ten link. Utwórz w klasie `HomeController` nową metodę `Delete()`, może ona wyglądać w następujący sposób:

```
public IActionResult Delete(int id)
{
    _phoneBook.Remove(id);
    return RedirectToAction("Index");
}
```

Zmodyfikuj klasę `PhoneBookService` w taki sposób, aby dodać do niej metodę `Remove()` która usunie z listy element o wskazanym identyfikatorze. Przetestuj działanie swojej aplikacji.



Zadanie 3.9. Zabezpieczenie usuwania

Co się stanie, jeżeli użytkownik odwoła się do identyfikatora, który nie istnieje? Spróbuj zmodyfikować metodę `Delete()` klasy `HomeController` w taki sposób, aby w takim wypadku zwracany był kod 404.

LABORATORIUM 4. TWORZENIE APLIKACJI INTERNETOWEJ Z WYKORZYSTANIEM PODEJŚCIA RAZOR PAGES.

Cel laboratorium:

Celem zajęć jest opracowanie aplikacji internetowej w architekturze Razor Pages obsługującej przysyłanie plików graficznych i ich prezentację.

Liczba punktów możliwych do uzyskania: 7 punktów

Zakres tematyczny zajęć:

Tworzenie projektu Razor Pages,
Modele stron,
Przesyłanie plików na serwer,
Tworzenie i wykorzystanie widoków częściowych,
Obsługa routingu,
Wykorzystanie biblioteki Magick.NET do manipulacji obrazami.

Pytania kontrolne:

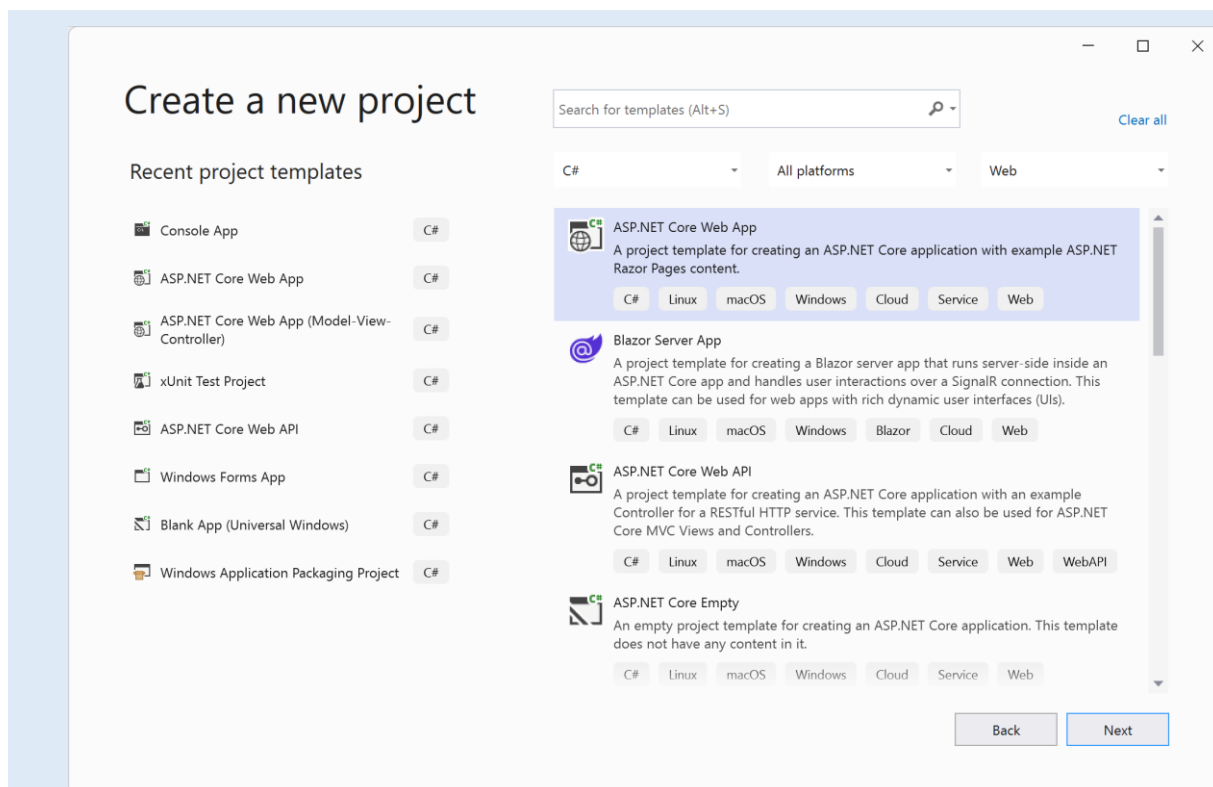
1. Czym różnią się aplikacje Razor Pages od aplikacji MVC?
2. Na czym polega routing w aplikacji internetowej?
3. Jak obsługuje się ścieżki plików w aplikacjach .NET?

Zadanie 4.1. Generowanie projektu Razor Pages

Razor Pages to alternatywne podejście do tworzenia aplikacji w stosunku do podejścia MVC – zamiast widoków i kontrolerów wykorzystywane są strony, które są *de facto* modelami danych wyposażonymi w metody `OnGet()` i `OnPost()` służące do reakcji na wysyłane żądania GET i POST.

Wariant dla Visual Studio

Wygeneruj nowy projekt wybierając w kreatorze tworzenia nowego projektu opcję ASP.NET Core Web App, jak na rysunku 4.1.



Rys 4.1. Opcja generowania projektu Razor Pages

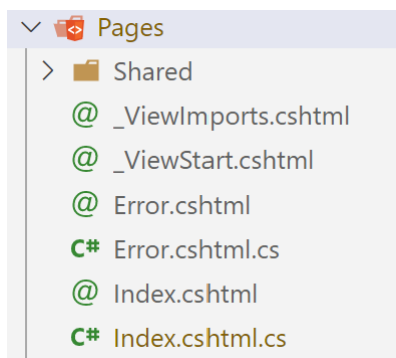
Na kolejnym ekranie wybierz nazwę swojej aplikacji, np. Lab4 i odznacz opcję używania HTTPS.

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Przejdź do nowego, pustego folderu o nazwie takiej, jak nazwa twojego projektu, np. Lab4 i wydaj komendę:

```
dotnet new razor --no-https -f net6.0
```

Jak zauważysz, projekt Razor Pages ma nieco inną strukturę niż projekt MVC – zamiast folderów takich jak Controllers czy Views dostępny jest tylko folder Pages, wewnątrz którego każda strona Razor Page jest reprezentowana przez dwa pliki – plik o rozszerzeniu .cshtml zawierający kod w składni Razor oraz odpowiadający mu plik „code behind” o rozszerzeniu .cs, zawierający kod w C#.



Rys 4.2. Pliki Razor i odpowiadające im pliki C#

Spróbuj uruchomić swoją aplikację i przeanalizuj kod strony Index oraz odpowiadającej jej klasy IndexModel.

Zadanie 4.2. Dodawanie atrybutu do modelu

Podobnie jak w aplikacjach MVC, strony Razor Pages są w zasadzie silnie typowanymi widokami. Aby móc udostępniać dane ze strony kodu w C# do strony kodu Razor niezbędne jest stworzenie odpowiednich pól, metod i właściwości. W pliku Index.cshtml.cs, w klasie IndexModel dodaj nową właściwość, która będzie prostą listą łańcuchów i posłuży do wyświetlania elementów. Skorzystaj z następującego kodu:

```
public List<string> Images { get; set; }
```

Teraz, w pliku Index.cshtml możesz zastosować pętlę, która będzie wyświetlać te elementy, może ona wyglądać następująco:

```
@foreach (var item in Model.Images)
{
    @item
}
```

Model powinien zostać jednak wypełniony jakimiś danymi. W tym laboratorium aplikacja będzie służyła do wyświetlania obrazków, które użytkownik będzie wysyłał do folderu images/ znajdującego się wewnątrz folderu wwwroot/. Utwórz zatem wewnątrz folderu wwwroot/ folder images/, ponieważ standardowo taki folder nie jest generowany.

Niezbędne będzie odnalezienie ścieżki do folderu wwwroot/ wewnątrz modelu strony aplikacji. Aby móc to wykonać, wykorzystany ponownie zostanie mechanizm Dependency Injection, który działa identycznie, jak w przypadku aplikacji MVC. W konstruktorze klasy IndexModel dodaj parametr:

```
IWebHostEnvironment environment
```

Do samej klasy dodaj prywatne pole:

```
private string imagesDir;
```

Natomiast w konstruktorze wykorzystaj następujący fragment kodu:

```
imagesDir = Path.Combine(environment.WebRootPath, "images");
```

Spowoduje to wypełnienie pola imagesDir ścieżką bezwzględną do folderu images/. Funkcja Path.Combine() zadba dodatkowo o to, aby ścieżki posiadały separatory zgodne z systemem, np. pod Windows jest to znak \, natomiast pod macOS znak /.

Wreszcie, możliwe jest wypełnienie pola Images – przygotuj do tego oddzielną prywatną metodę:

```
private void UpdateFileList()
{
    Images = new List<string>();
}
```



```
foreach (var item in
Directory.EnumerateFiles(imagesDir).ToList())
{
    Images.Add(Path.GetFileName(item));
}
}
```

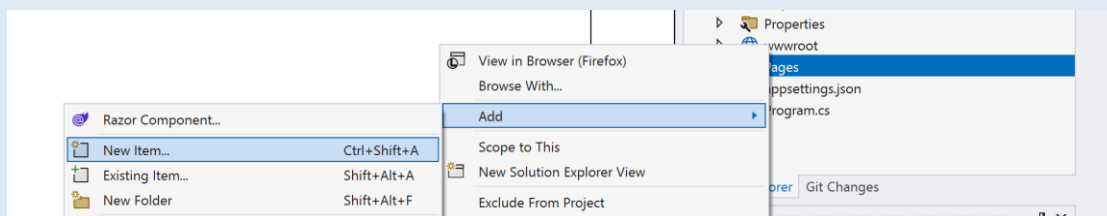
Następnie, uruchom tę metodę wewnątrz funkcji OnGet(). Spowoduje to, że lista plików w folderze z obrazkami znajdzie się w modelu danych, a co za tym idzie – będzie wyświetlona użytkownikowi.

Zadanie 4.3. Wysyłanie plików na serwer

W kolejnym kroku niezbędne będzie wysyłanie plików na serwer, aby były prezentowane użytkownikowi. Przygotuj zatem nową stronę Razor Page w folderze Pages.

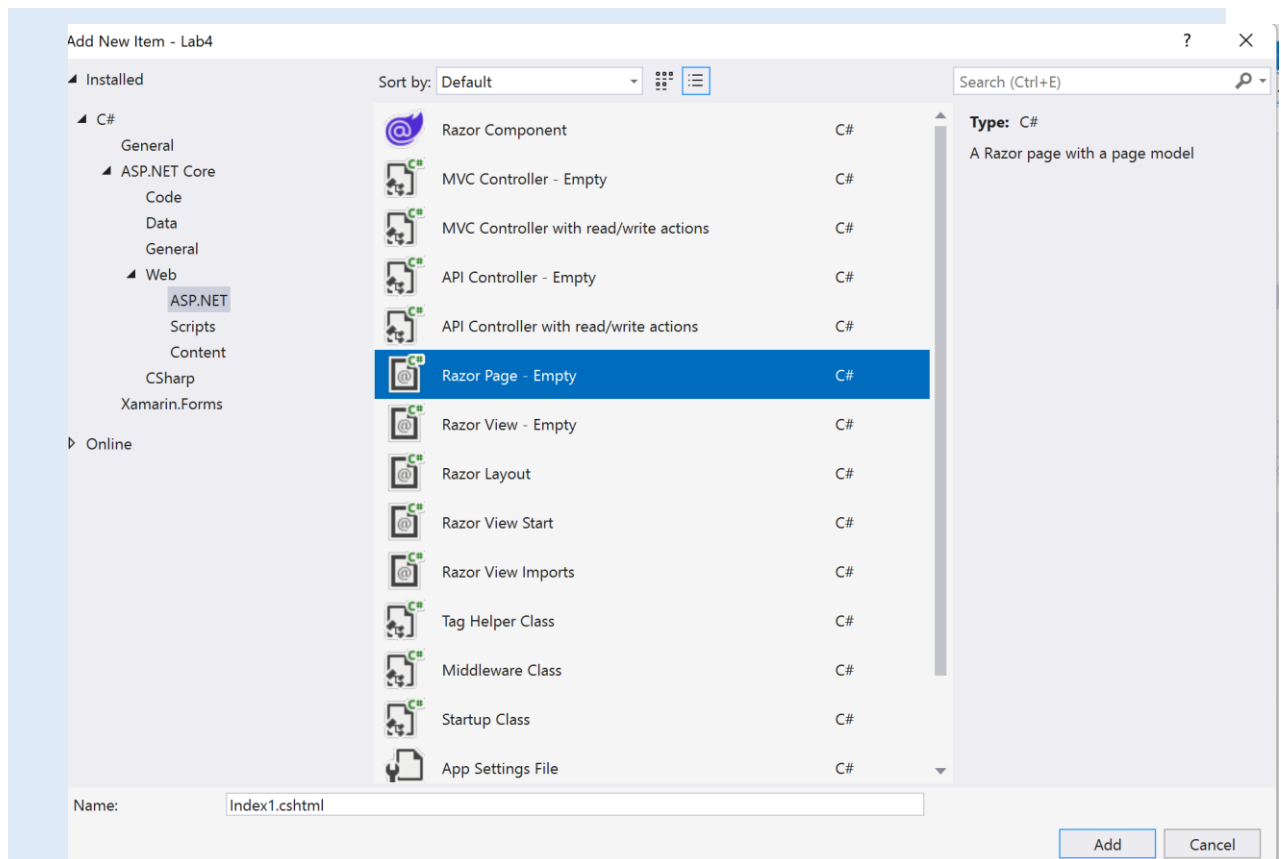
Wariant dla Visual Studio

Kliknij prawym klawiszem myszy na folderze Pages i wybierz Add -> New Item.



Rys 4.3. Dodawanie nowej strony Razor

W kolejnym kroku, wybierz „Razor Page – Empty” i nazwij ją „Upload.cshtml”.



Rys 4.4. Wybór szablonu dodawanego elementu

Spowoduje to wygenerowanie zarówno pliku Upload.cshtml, jak i jego odpowiednika w C#, Upload.cshtml.cs.

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Przejdź do głównego folderu swojego projektu i wydaj komendę:

```
dotnet add package  
Microsoft.VisualStudio.Web.CodeGeneration.Design --version 6.0
```

Aby dodać generatory do projektu, a następnie wydaj komendę:

```
dotnet aspnet-codegenerator razorpage Upload Empty -udl -  
outDir Pages
```

Co spowoduje wygenerowanie pustej strony Razor Page o nazwie Upload w folderze Pages. Strona będzie korzystać z domyślnego layoutu.

Teraz, niezbędne będzie dodanie formularza do wysyłania plików. Aby łatwo skorzystać z generatora ścieżek, który automatycznie utworzy formularz prowadzący do odpowiedniego parametru modelu zacznij od dodania w modelu UploadModel właściwości, która będzie przechowywała plik przekazany od użytkownika:

```
[BindProperty]  
public IFormFile Upload { get; set; }
```


Atrybut `[BindProperty]` spowoduje, że dane pochodzące z formularza użytkownika zostaną automatycznie przypisane do tego pola. Teraz, w metodzie `OnPost()` będzie można obsłużyć wysyłanie pliku na serwer i zapisanie go do finalnej lokalizacji. Podobnie, jak w poprzednim zadaniu, utwórz pole:

```
private string imagesDir;
```

Natomiast w konstruktorze dodaj parametr typu `IWebHostEnvironment` i wypełnij ścieżką do folderu `imagesDir`:

```
imagesDir = Path.Combine(environment.WebRootPath, "images");
```

Teraz, aby obsłużyć wysyłanie pliku na serwer, utwórz metodę `OnPost()` zwracającą `ActionResult` jeżeli taka nie istnieje i wypełnij ją treścią – możesz skorzystać z następującego kodu:

```
if (Upload != null)
{
    string extension = ".jpg";
    switch (Upload.ContentType)
    {
        case "image/png":
            extension = ".png";
            break;
        case "image/gif":
            extension = ".gif";
            break;
    }

    var fileName =
        Path.GetFileNameWithoutExtension(Path.GetRandomFileName()) +
        extension;

    using (var fs =
        System.IO.File.OpenWrite(Path.Combine(imagesDir, fileName)))
    {
        Upload.CopyTo(fs);
    }
}

return RedirectToPage("Index");
```



Jak widzisz, aplikacja jest przygotowana głównie do obsługi obrazków. Każdy plik wysyłany ma nadawaną nową, losową nazwę i jest kopiowany z pliku przekazywanego przez użytkownika do folderu z obrazkami. Kiedy kopiowanie pliku się powiedzie, użytkownik jest przekierowywany do strony głównej aplikacji

Formularz prezentowany użytkownikowi też będzie odpowiednio przygotowany. Aby go wygenerować, skorzystaj z następującego kodu:

```
<form asp-page="Upload" method="post" enctype="multipart/form-data">
    <input class="form-control" asp-for="Upload" type="file"
accept=".jpg, .png, .jpeg, .gif, .bmp, .tif, .tiff|image/*" />
    <input class="form-control" type="submit" value="Upload"
/>
</form>
```

Zostanie wygenerowany formularz o ścieżce prowadzącej do strony Upload, którego `enctype` zakłada przesyłanie plików. Element `input` również posiada atrybut `asp-for`, który powiąże go z nazwą w modelu strony. Atrybut `accept` określi jakie typy danych mogą być wysyłane z wykorzystaniem formularza.

Te dodatkowe atrybuty nieobecne w HTML, takie jak `asp-for` to tzw. tag helpers, mechanizm, który ułatwia tworzenie elementów odnoszących się do elementów kodu C# ze strony kodu Razor.

Zadanie 4.4. Wysyłanie plików na serwer

Zmodyfikuj plik `Index.cshtml`, aby dodać tam odnośnik do strony wysyłania plików. Znow możesz skorzystać z mechanizmu tag helpers, na przykład w taki sposób:

```
<a asp-page="Upload">Upload new file...</a>
```

Spowoduje to dodanie odnośnika do strony `Upload.cshtml` z uwzględnieniem reguł routingu.

Spróbuj wysłać kilka plików na serwer i zobacz, czy strona główna poprawnie prezentuje ich losowo wygenerowane nazwy.

Lab4 Home Privacy

[Upload new file...](#) 3sppdf44.jpg5la4qamz.jpg5ud0qowf.jpgqkuj1qis.jpgqmzmwofo.gifqprh0le.jpg

Rys 4.5. Oczekiwany wygląd aplikacji

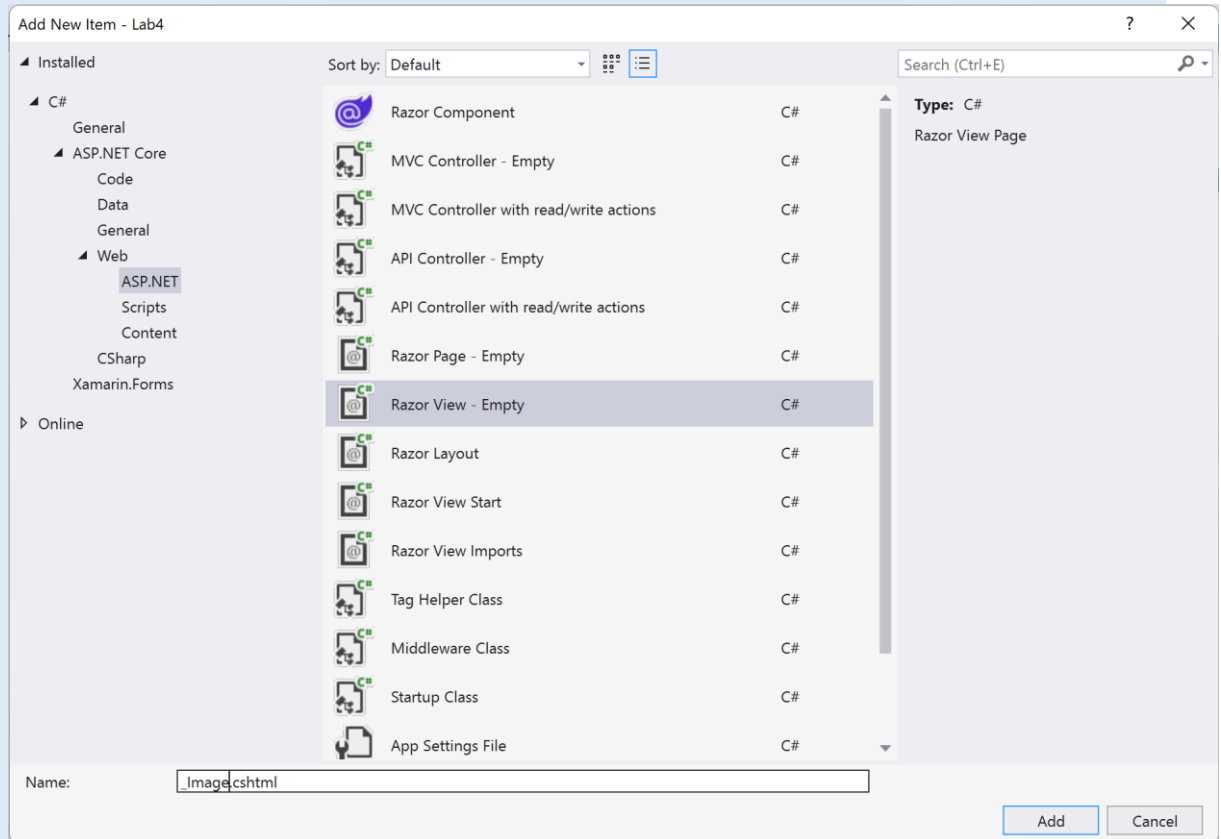
Zadanie 4.5. Zmiana wyświetlania obrazków z wykorzystaniem widoków częściowych

Zaprezentowany widok aplikacji nie jest zbyt atrakcyjny. Zamiast tego lepiej by było, aplikacja prezentowała obrazki w postaci znaczników ``. Aby to osiągnąć, skorzystamy z mechanizmu widoków częściowych. Widoki częściowe generują powtarzalne fragmenty kodu Razor – w naszym wypadku będzie to widok pojedynczego obrazka.



Wariant dla Visual Studio

Kliknij prawym klawiszem myszy na folderze Pages/Shared i wybierz Add -> New Item z menu podręcznego. W kolejnym kroku wybierz szablon „Razor View – Empty” i nazwij go `_Image.cshtml`.



Rys 4.6. Generowanie nowego Razor View

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Wydaj komendę:

```
dotnet aspnet-codegenerator razorpage _Image Empty -outDir  
Pages/Shared -npm
```

Spowoduje to wygenerowanie strony o nazwie `_Image` na podstawie szablonu `Empty` w folderze `Pages/Shared`, a parametr `-npm` powoduje, że nie zostanie wygenerowany model odpowiadający stronie.

Razor View jest identyczny z Razor Page, lecz nie posiada odpowiadającego mu modelu w postaci pliku C#. Nazwa zaczynająca się od `_` oznacza, że będzie to widok częściowy.

Usuń zawartość pliku `_Image.cshtml` i wykorzystaj następujący kod:



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
@model string
```

```

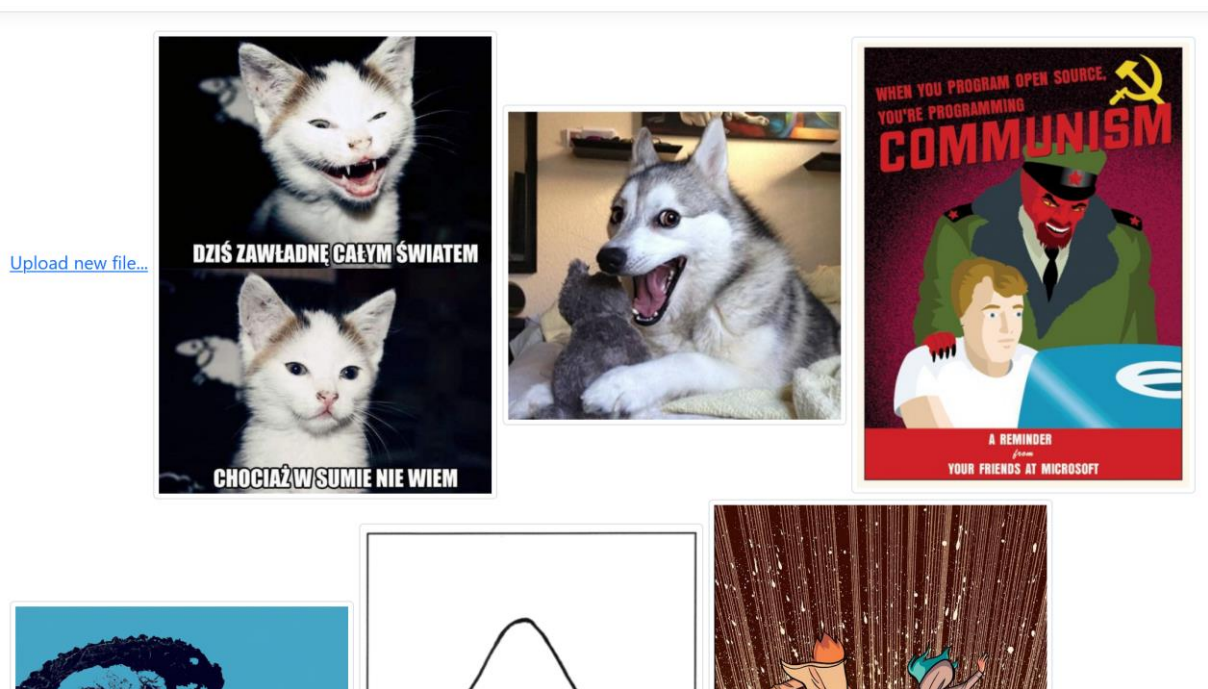
```

Wykorzystanie tutaj znaku ~ w ścieżce służy do automatycznego wygenerowania względnego odnośnika do folderu wwwroot/. @Model jest typu string i będzie to nazwa pliku z obrazkiem. Ostatecznie, zmodyfikuj Index.cshtml, aby wykorzystywał widok częściowy, na przykład w taki sposób:

```
@foreach (var item in Model.Images)
{
    <partial name="_Image" model="item" />
}
```

Aplikacja powinna prezentować się w sposób podobny do zaprezentowanego na rysunku:

Lab4 Home Privacy



Rys 4.7. Widok aplikacji z listą obrazków

Spróbuj zmodyfikować listę obrazków, pamiętając, że standardowy szablon oparty jest o framework Bootstrap, w taki sposób, aby wszystkie obrazki znajdowały się jeden pod drugim oraz były wyśrodkowane, tak jak zaprezentowano na rysunku 4.8.

[Upload new file...](#)



Rys 4.8. Oczekiwany wygląd aplikacji

Zadanie 4.6. Strona pojedynczego obrazka

Dodaj do swojej aplikacji jeszcze jedną stronę, o nazwie `Single`. Jak możesz zauważyć, domyślny routing jest realizowany w taki sposób, że strona `Upload` jest dostępna pod adresem <http://example.com/upload>, więc strona `Single` będzie dostępna pod adresem <http://example.com/single>.

Chcemy mieć możliwość przekazywania danych do strony poprzez wykorzystanie adresu URL – strona `Single` uzyska nazwę obrazka i wyświetli tylko jego, w pełnej szerokości; chcemy aby nazwa obrazka była przekazywana przez URL, na przykład <http://example.com/single/plik.jpg>. Aby to osiągnąć, niezbędne będą dwie rzeczy:

Zmodyfikuj dyrektywę `@page` w pliku `Single.cshtml`, aby wyglądała w sposób następujący:

```
@page "{image}"
```

Spowoduje to, że dane z URL-a zostaną zapisane do zmiennej modelu o nazwie `Image`. Następnie, w pliku `Single.cshtml.cs` w klasie `SingleModel` musisz dodać odpowiednią zmienną modelu:

```
[BindProperty(SupportsGet = true)]
public string Image { get; set; }
```

Atrybut `[BindProperty]` tutaj jest wykorzystywany, aby zmienna ta została automatycznie wypełniona podstawie danych pochodzących z URL.

Musimy dodać jeszcze mechanizm, aby obrazek był faktycznie wyświetlany – skorzystaj z następującego kodu w `Single.cshtml`:

```

```

Wreszcie, niezbędne jest dodanie mechanizmu, który będzie wyświetlał stronę lub zwracał kod błędu 404, jeżeli użytkownik próbuje odwołać się do nieistniejącego obrazka. Podobnie jak w poprzednich dwóch modelach dodaj prywatne pole `imagesDir` i je wypełnij w konstruktorze, natomiast w metodzie `OnGet()` wykorzystaj następujący kod:

```
public IActionResult OnGet()
{
    if (System.IO.File.Exists(Path.Combine(imagesDir, Image)))
    {
        return Page();
    }
    else
    {
        return NotFound();
    }
}
```

Jak możesz zauważyć, domyślnie metody `OnGet()` i `OnPost()` są typu `void`, jednak mogą również zwracać `IActionResult`, co zwiększa kontrolę nad możliwymi rezultatami działania strony – pozwala na przykład na zwracanie przekierowań, czy też kodów błędów HTTP.

Zmodyfikuj swój widok częściowy `_Image.cshtml`, aby każdy obrazek posiadał też hiperłącze prowadzące do `/single/nazwapliku.jpg`, na przykład w taki sposób:

```
<a asp-page="Single" asp-route-image="@Model"></a>
```

Wygeneruje to automatycznie łącze do strony `single`, gdzie do nazwy „image” zostanie wprowadzona nazwa pliku i mechanizm ten automatycznie wygeneruje odpowiednie hiperłącza zgodne z zadeklarowanym routingiem stron w aplikacji.

Zadanie 4.7. Znaki wodne

Dodaj do swojej aplikacji pakiet `Magick.NET-Q8-AnyCPU`. Posłuży on do manipulacji obrazami, w tym konkretnym przypadku będzie służył do dodawania znaków wodnych do obrazków przesłanych do systemu.

Dodaj do swojego projektu, do głównego folderu, mały obrazek, który będzie umieszczany jako znak wodny w rogu obrazka. Nazwij go np. `watermark.png`. Następnie, zmodyfikuj mechanizm umieszczania plików na serwerze – w obecnej chwili bezpośrednio zapisuje on przesłany strumień danych do pliku na serwerze, niezbędne jednak będzie przekazanie go najpierw do biblioteki `Magick.NET`.

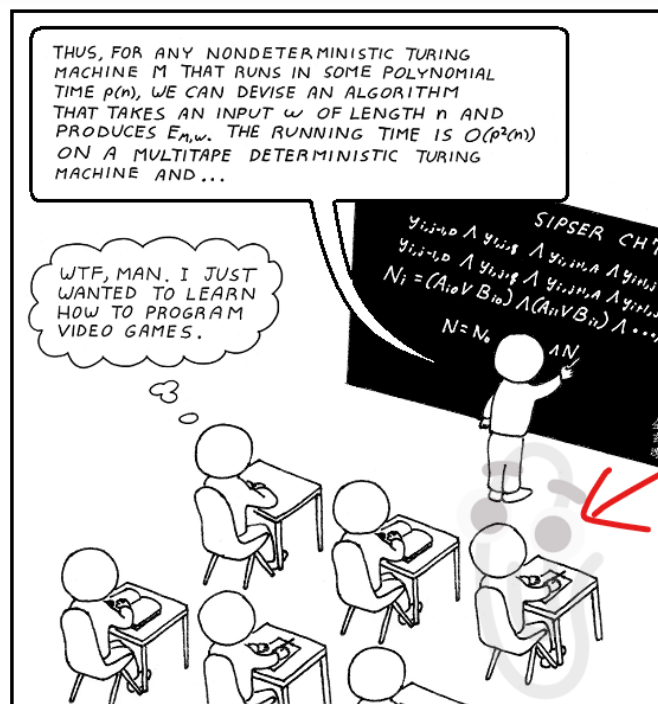
Do samego dodawania znaków wodnych możesz wykorzystać następujący fragment kodu:




```
using var image = new MagickImage("<ścieżka do pliku lub  
bezpośrednio strumień>");  
using var watermark = new MagickImage("watermark.png");  
  
// przezroczystosc znaku wodnego  
watermark.Evaluate(Channels.Alpha, EvaluateOperator.Divide,  
4);  
  
// narysowanie znaku wodnego  
image.Composite(watermark, Gravity.Southeast,  
CompositeOperator.Over);  
  
image.Write(path);
```

Umieszczane od tej pory pliki na serwerze powinny być opatrywane znakiem wodnym, tak jak zaprezentowano na rysunku 4.9.

Lab4 Home Privacy



Rys 4.9. Półprzezroczysty znak wodny

LABORATORIUM 5. TWORZENIE APLIKACJI INTERNETOWEJ KORZYSTAJĄCEJ Z BAZY DANYCH ORAZ MECHANIZMU ORM

Cel laboratorium:

Celem laboratorium jest nauka wykorzystania platformy Entity Framework oraz generatorów kodu do automatycznego tworzenia aplikacji typu CRUD.

Liczba punktów możliwych do uzyskania: 8 punktów

Zakres tematyczny zajęć:

Generowanie aplikacji MVC i obsługa zależności,
Tworzenie i wykorzystanie DbContext,
Wykorzystanie mechanizmu sekretów,
Generowanie kontrolera opartego o DbContext,
Modyfikacje widoków, szablony wyświetlania i edycji.

Pytania kontrolne:

1. Czym są zależności (dependencies) w dużych systemach informatycznych?
2. Do czego służą biblioteki typu ORM?
3. W jaki sposób prezentowane są dane słownikowe w HTML?

Zadanie 5.1. Generowanie aplikacji MVC i dodawanie zależności

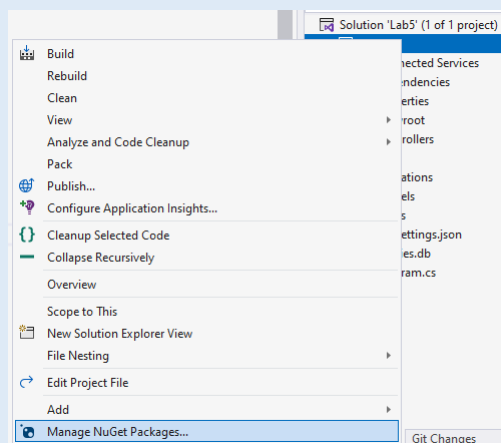
Utwórz nową aplikację opartą o podejście MVC (korzystającą z szablonu „ASP.NET Core Web App (Model-View-Controller)”), tak samo jak w Laboratorium 3. Należy do niej dodać zależności – zewnętrzne biblioteki, które posłużą do komunikacji z systemem baz danych. Platforma .NET wykorzystuje system o nazwie NuGet – jest to zarówno narzędzie konsolowe oraz składnik systemu budowania projektów, jak i zewnętrzne repozytorium bibliotek dostępne pod adresem <https://nuget.org>. Biblioteki dostępne w systemie NuGet mogą posiadać własne zależności i system będzie automatycznie dbał, aby były one spełnione. Wiele bibliotek firmy Microsoft i organizacji trzecich dostępnych jest jako pakiety NuGet.

W poprzednim laboratorium dodaliśmy zależności związane z ML.NET za pomocą edycji pliku `.csproj`, jest to jednak niezbyt wygodne – zamiast tego można zastosować okno zarządzania zależnościami lub odpowiednią komendę.

Wariant dla Visual Studio

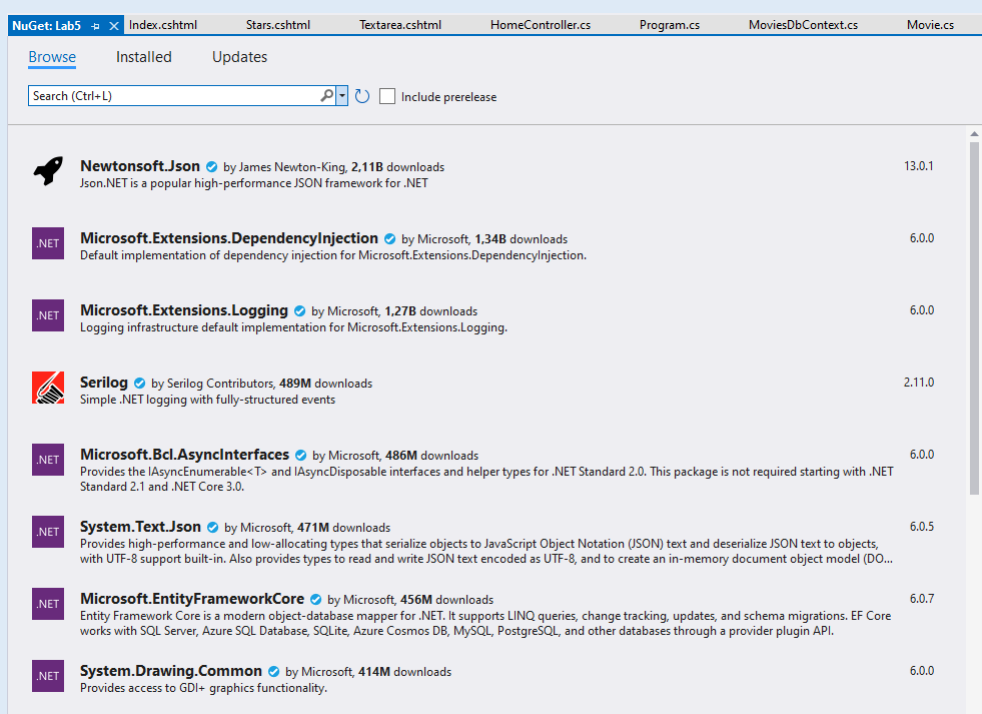
Kliknij prawym przyciskiem myszy na projekcie i wybierz opcję „Manage NuGet Packages”.





Rys 5.1. Opcja zarządzania pakietami NuGet

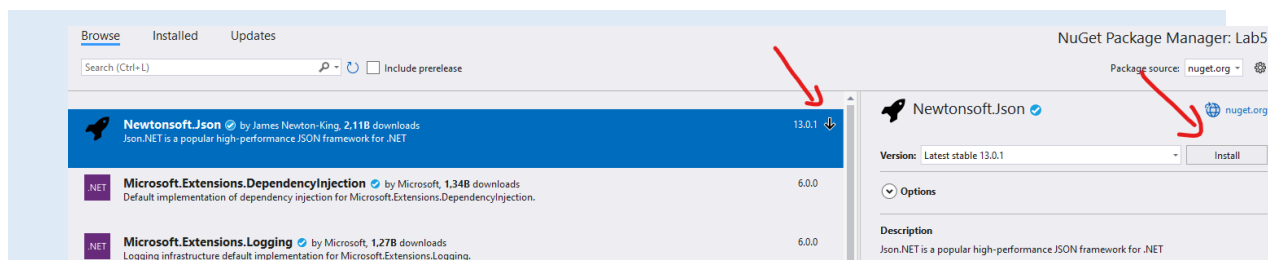
Uruchomi się okno przeglądarki pakietów. Wybierz zakładkę „Browse” i uzyskasz dostęp do biblioteki pakietów NuGet.



Rys 5.2. Okno biblioteki pakietów NuGet

Teraz możesz za pomocą wyszukiwarki znaleźć poszukiwaną bibliotekę i po najechaniu kursorem na nią wybrać opcję dodania do projektu lub kliknąć na pozycję i skorzystać z panelu po prawej stronie do oceny biblioteki i jej ewentualnej instalacji.





Rys 5.3. Informacje o bibliotece i przyciski instalacji

Zainstaluj w ten sposób bibliotekę `Microsoft.EntityFrameworkCore.Sqlite` w wersji 6.0.x.

Możesz też, alternatywnie, używać narzędzi konsolowych w samym Visual Studio – kliknij prawym klawiszem myszy na projekcie i wybierz opcję „Open in Terminal”, a w uruchomionym oknie Terminala możesz korzystać z narzędzi linii komend i na przykład komendy:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite --
version 6.0
```

Która również spowoduje dodanie biblioteki do projektu.

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Przejdź do głównego folderu swojej aplikacji i wydaj komendę:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite -
version 6.0
```

Zadanie 5.2. Tworzenie modelu danych

Utwórz nową publiczną klasę, model danych, w folderze `Models/`, o nazwie `Movie`. Będzie musiała przechowywać informacje o filmach, ale jednocześnie będzie reprezentowana w systemie baz danych. Dodaj do niej następujące właściwości: identyfikator (`int`), tytuł (`string`), opis (`string`), ocena (`int`) oraz odwołanie (`link`) do zwiastuna na platformie YouTube (`string`). Możesz je nazwać np. `Id`, `Title`, `Description`, `Rating` oraz `TrailerLink`.

Uwaga: jeżeli właściwość identyfikatora nie będzie nazywała się `Id`, należy dodać do niej (nad nią) atrybut `[Key]`, aby Entity Framework skorzystał z tego elementu do utworzenia klucza głównego.

Następnie, musisz dodać kolejną klasę, tym razem do folderu `Data`, który musisz ręcznie utworzyć, która będzie tzw. kontekstem danych, czyli klasą, która gromadzi wszystkie tabele bazodanowe, z których będzie korzystać aplikacja, oraz je opisywać. Nazwij ją `MoviesDbContext` i skorzystaj z następującego kodu (dodaj odpowiednie elementy do sekcji `using` w razie potrzeby):

```
public class MoviesDbContext : DbContext
{
    public DbSet<Movie> Movies { get; set; }
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
public MoviesDbContext(DbContextOptions options) :
base(options)
{

}
}
```

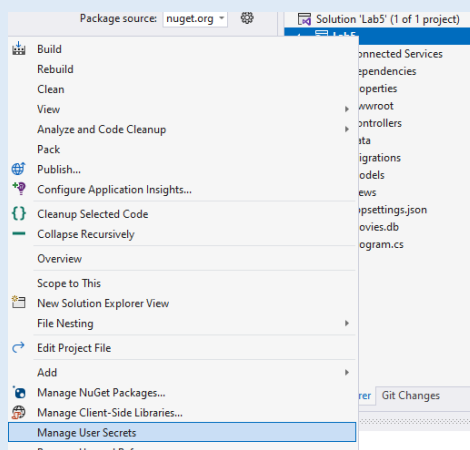
Uwaga: bardzo ważne jest to, aby `DbSet<T>` był właściwością, a nie polem – w innym przypadku biblioteka Entity Framework nie będzie mogła z niej skorzystać. Podobnie, klasa `Movie` musi być publiczna, ponieważ klasa `MoviesDbContext` jest publiczna.

Zadanie 5.3. Sekrety i połączenie do bazy danych

W tym laboratorium wykorzystywany będzie system bazodanowy SQLite, który jest plikową bazą danych, gdzie wszystkie dane przechowywane są w pojedynczym pliku na serwerze, stąd kwestia bezpieczeństwa nie jest tak istotna – w rzeczywistym świecie jednak aplikacje serwerowe potrzebują informacji dostępowych do bazy danych, które są tajne, np. jest nimi hasło i login, stąd nie mogą być przechowywane bezpośrednio w kodzie źródłowym – oddzielenie tych informacji pozwala również na np. łatwe zmiany pomiędzy środowiskiem testowym i środowiskiem produkcyjnym. W ASP.NET Core będziemy wykorzystywali mechanizm sekretów aby przechować dane dostępowe do bazy danych.

Wariant dla Visual Studio

Kliknij prawym klawiszem myszy na projekcie i wybierz opcję „Manage User Secrets”.



Rys 5.4. Opcja w menu kontekstowym

Spowoduje to uruchomienie edytora i otwarcie w nim pliku `secrets.json`. Plik ten jest przechowywany w profilu lokalnego użytkownika i chroniony mechanizmami systemu operacyjnego. W pliku `secrets.json` dodaj następujący kod:

```
{
  "ConnectionStrings": {
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
"Default": "Data Source=movies.db"
}
}
```

Wariant dla Visual Studio Code oraz narzędzi konsolowych

W głównym folderze swojego projektu wydaj komendę:

```
dotnet user-secrets init
```

Spowoduje to utworzenie pliku „sekreów” w lokalnym folderze użytkownika. Następnie ustaw ciąg połączenia do bazy danych wydając komendę:

```
dotnet user-secrets set ConnectionStrings:Default "Data
Source=movies.db"
```

Ciąg wykorzystywany tutaj to tzw. „connection string” – jest to mechanizm wykorzystywany przez wiele bibliotek .NET do określenia wszystkich parametrów połączenia z bazą danych, takich jak nazwa serwera, login użytkownika, hasło i tym podobne, z których dostawca SQLite wykorzystuje tylko nazwę pliku bazy danych. Bardziej rozbudowane *connection string* mogą wyglądać na przykład tak:

```
Server=example.com;Port=3306;Database=iaitemperature;Uid=iai;P
wd=BfKFjrDIj165z IKNeF9x!;Caching=true
```

Mechanizm ten jest bardzo zbliżony do ciągów definiujących połączenie ODBC czy JDBC.

Następnie, ostatnim krokiem będzie wykorzystanie tej wartości w programie. W pliku `Program.cs`, powyżej liniiki `var app = builder.Build();` zarejestruj kontekst bazy danych oraz przekaz mu informację z konfiguracji, tj. nazwę ciągu *connection string*:

```
builder.Services.AddDbContext<MoviesDbContext>(
    options => options.UseSqlite(
        builder.Configuration.GetConnectionString("Default")
    );
```

Mechanizm ten wykorzystuje konfigurację – jeżeli ciąg *connection string* o nazwie `Default` zostanie przekazany przez opcje programu, konfigurację w pliku `appsettings.json` lub zmienne środowiskowe, to zostanie wykorzystany – jeżeli nie, to aplikacja spróbuje go automatycznie pobrać z lokalnych „sekreów”.

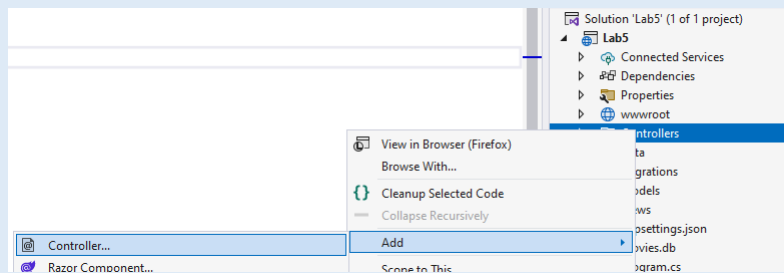
Zadanie 5.4. Generowanie kontrolera

Wykorzystany w tym laboratorium będzie automatyczny generator, pozwalający na wygenerowanie kontrolera typu CRUD (Create, Read, Update, Delete) w stosunku do modelu bazodanowego, wraz ze wszystkimi akcjami. Usuń kontroler `HomeController.cs` w folderze `Controllers/` oraz cały folder `Views/Home/`.

Wariant dla Visual Studio

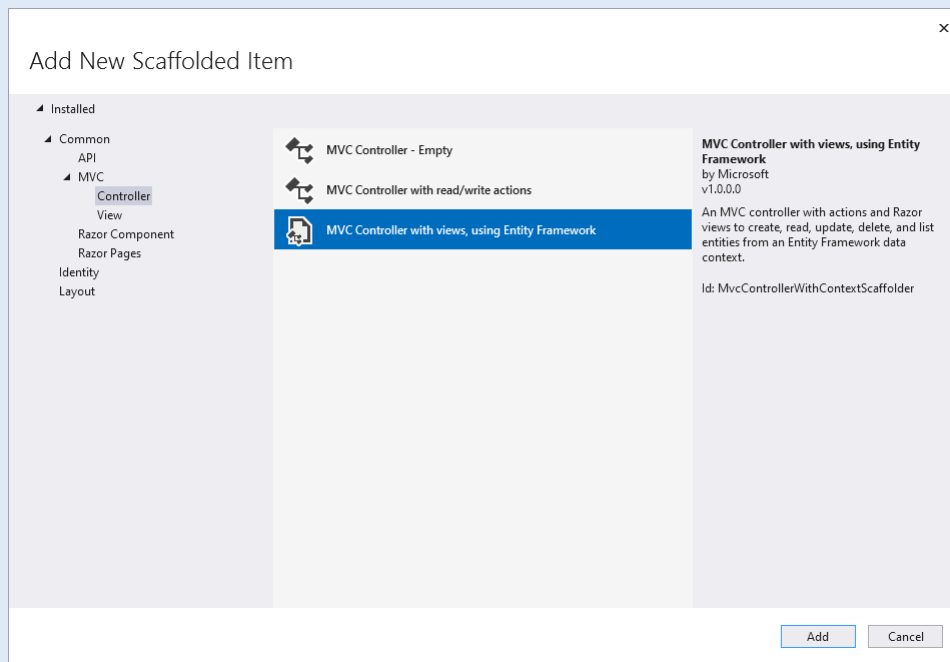
Kliknij prawym klawiszem myszy na folderze `Controllers` w projekcie i wybierz opcję `Add -> Controller`.





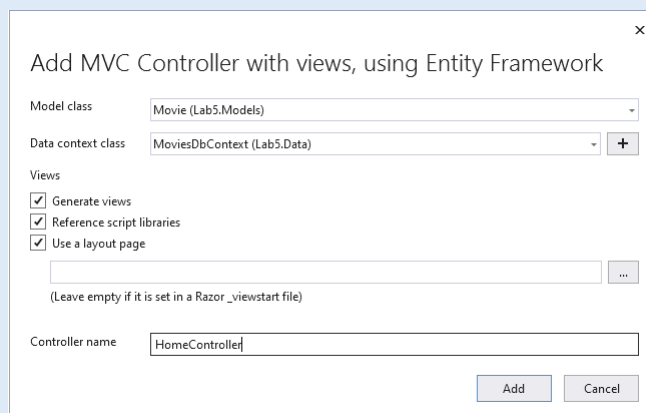
Rys 5.5. Opcja dodawania kontrolera.

W kolejnym kroku wybierz szablon „MVC Controller with views, using Entity Framework”.



Rys 5.6. Wybór szablonu kontrolera

W ostatnim kroku wybierz klasę modelu, klasę kontekstu danych, oraz nazwij generowany wynik HomeController.



Rys 5.7. Wybór parametrów generowania kontrolera

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Wydadź w głównym folderze swojego projektu komendy, aby zainstalować narzędzia generatora kodu:

```
dotnet add package  
Microsoft.VisualStudio.Web.CodeGeneration.Design --version 6.0  
dotnet add package Microsoft.EntityFrameworkCore.Design --  
version 6.0
```

A następnie wydaj komendę:

```
dotnet aspnet-codegenerator controller -name HomeController -m  
Movie -dc MoviesDbContext -udl -sqlite -outDir Controllers
```

Aplikacja jednak nie może się jeszcze uruchomić, ponieważ na tym etapie baza danych jeszcze nie istnieje i nie zawiera tabeli **Movies**. Aby utworzyć tabelę skorzystamy z mechanizmu migracji. Także w Visual Studio wymaga to użycia narzędzi konsolowych, stąd otwórz okno konsoli opcją „Open in Terminal” z menu kontekstowego.

Następnie, w oknie terminala (lub twojej otwartej konsoli w głównym folderze aplikacji) wydaj komendę:

```
dotnet ef migrations add Initial
```

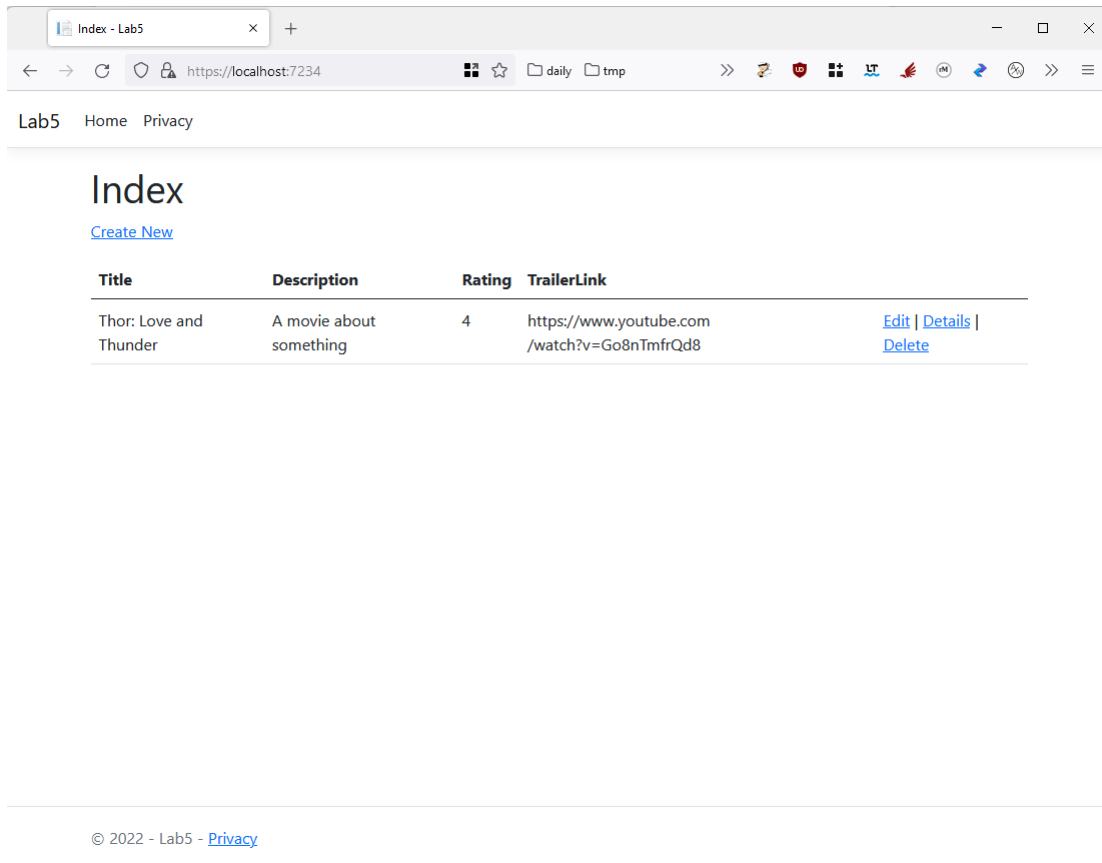
Spowoduje to wygenerowanie folderu **Migrations** i w nim klas odpowiedzialnych za migracje bazodanowe. Migracje to mechanizm utrzymywania informacji o zmianach w schemacie bazy danych, co pozwala zmieniać strukturę bazy danych utrzymując zmiany w kodzie.

Następnie, utwórz bazę danych za pomocą komendy:

```
dotnet ef database update
```

Spróbuj uruchomić swoją aplikację i przetestuj, czy działają wszystkie opcje oraz czy działa system persystencji danych.





Rys 5.8. Przykład działania aplikacji

Zadanie 5.5. Zmiana formy wprowadzania danych

Jak możesz zauważyć, domyślnie wszystkie pola edytowania pól filmu wyświetlane są w taki sam sposób – jako elementy HTML `<input>`. Chcielibyśmy zmienić to dla pola opisu filmu w taki sposób, aby wykorzystywany był element `<textarea>`. Jak możesz zauważyć, w `Views/Home/Create.cshtml` oraz `Views/Home/Edit.cshtml` wykorzystywany jest jawnie element `<input>` za pomocą mechanizmu *tag helpers*. Zmodyfikuj tę linię na wykorzystanie metody `EditorFor` w obydwu plikach widoku:

```
@Html.EditorFor(m => m.Description)
```

Następnie, utwórz folder `EditorTemplates` w folderze `Views/Shared` (wielkość liter w nazwie ma znaczenie), a w nim utwórz plik (w Visual Studio możesz skorzystać z opcji `Add -> View -> Razor View – Empty`) o nazwie `LongText.cshtml`.

W pliku `LongText.cshtml` musimy zdefiniować jak będzie wyglądał edytor dla elementów, które będą oznaczone flagą `LongText` – chcemy wykorzystać element `<textarea>` i można tutaj skorzystać z odpowiedniej metody klasy `Html`, która go wygeneruje, na przykład:

```
@model string
@Html.TextArea("", Model, new { cols = 40, rows = 10,
@class="form-control" })
```

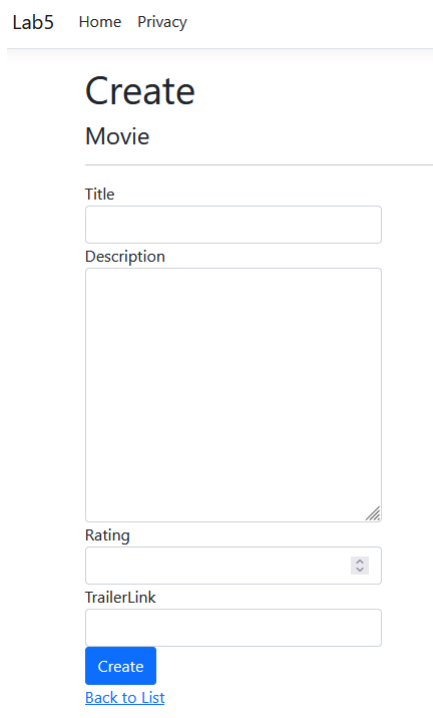
Dodatkowe parametry w metodzie `TextArea()` są przekazywane w postaci anonimowego obiektu i zostaną bezpośrednio zmapowane na atrybuty HTML – możesz zauważyć, że w C# można nazwać zmienne tak jak słowa kluczowe, o ile zostanie wykorzystany znak `@`, co spowoduje utworzenie atrybutu `class`, który jest niezbędny do określenia stylu elementu w bibliotece Bootstrap.

Ostatnim krokiem jest zdefiniowanie, że konkretny atrybut modelu powinien korzystać z konkretnego szablonu edytora – dodaj do pola opisu w swoim modelu atrybut `UIHint`:

```
[UIHint("LongText")]
```

(może być wymagane dodanie `System.ComponentModel.DataAnnotations` do sekcji `using`)

Teraz, edycja i tworzenie nowego filmu powinno wyglądać w następujący sposób:



The screenshot shows a web application interface for creating a movie. At the top, there is a navigation bar with links for 'Lab5', 'Home', and 'Privacy'. Below this, the main heading is 'Create Movie'. The form contains four input fields: 'Title' (a single-line text box), 'Description' (a multi-line text area), 'Rating' (a dropdown menu), and 'TrailerLink' (a single-line text box). At the bottom of the form is a blue 'Create' button and a link labeled 'Back to List'.

Rys 5.9. Zaktualizowany formularz edycji i tworzenia filmu

Zadanie 5.6. Zmiana formy wyświetlania danych

Chcemy, aby ocean nie była wyświetlana jako liczba, a jako odpowiednia liczba „gwiazdek” na liście elementów.

Dodaj do folderu `Views/Shared/` folder `DisplayTemplates`, a wewnątrz niego utwórz plik `Stars.cshtml` (ponownie korzystając z pustego szablonu). Teraz, możesz skorzystać z następującego kodu:

```
@model int
@for (int i = 0; i < Model; i++)
{
    @Html.Raw("&#x2606;")
}
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



}

Następnie, dodaj do swojego modelu, do pola oceny filmu, atrybut `UIHint`, który spowoduje wykorzystanie odpowiedniego szablonu do wyświetlania:

```
[UIHint("Stars")]
```

Od teraz ocena filmu powinna być wyświetlana w następujący sposób:

Index

[Create New](#)

Title	Description	Rating	TrailerLink	
Thor: Love and Thunder	A movie about something	☆☆☆☆	https://www.youtube.com/watch?v=Go8nTmfrQd8	Edit Details Delete

Rys 5.10. Przykładowe działanie aplikacji

Spróbuj zmodyfikować ten sposób wyświetlania, aby wyświetlane były wypełnione lub gwiazdki uzupełniając wynik do maksymalnie 5, tak jak zaprezentowano na rysunku 5.11. Wykorzystaj encje HTML: `☆`; oraz `★`; – zauważ, że należy je przekazywać przez funkcję `Html.Raw()` lub można obejmować w znaczniki, np. ``. Zastosuj atrybut HTML `title` do wyświetlenia oceny w formie liczbowej po najechnaniu myszą na ocenę.

Details

Movie

Title	Thor: Love and Thunder
Description	A movie about something
Rating	★★★★☆
TrailerLink	https://www.youtube.com/watch?v=Go8nTmfrQd8

Rys 5.11. Oczekiwane działanie aplikacji

Zadanie 5.7. Dodawanie hiperłączy

Zmodyfikuj aplikację w taki sposób, aby na liście filmów nie był prezentowany adres URL w formie tekstowej, ale aby było to hiperłącze prowadzące do danego adresu, o treści np. „Trailer”, jak zaprezentowano na rysunku.

Index

[Create New](#)

Title	Description	Rating	TrailerLink	
Thor: Love and Thunder	A movie about something	★★★★☆	Trailer	Edit Details Delete

Rys 5.12. Oczekiwane działanie aplikacji



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



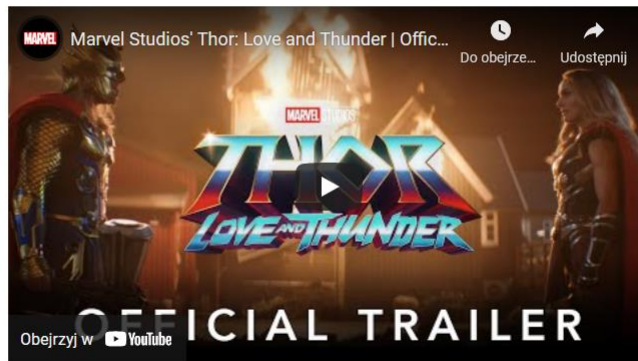
Zadanie 5.8. Wyświetlanie zwiastuna jako ramki wewnętrznej

W tym momencie zwiastun jest prezentowany jako odnośnik na ekranie szczegółów filmu – chcemy jednak, aby był prezentowany w postaci elementu `<iframe>` i został osadzony na stronie, jak zaprezentowano na rysunku 5.13, poniżej.

Details

Movie

Title Thor: Love and Thunder
Description A movie about something
Rating ★★★★★
TrailerLink



[Edit](#) | [Back to List](#)

Rys 5.13. Oczekiwane działanie aplikacji

Możesz zauważyć, że do umieszczenia elementu `<iframe>` z odnośnikiem do filmu stosowany jest następujący kod HTML generowany przez platformę YouTube:

```
<iframe width="640" height="360"
src="https://www.youtube.com/embed/dQw4w9WgXcQ"
frameborder="0"
allow="accelerometer; autoplay; clipboard-write;
encrypted-media; gyroscope; picture-in-picture"
allowfullscreen></iframe>
```

Spróbuj wykorzystać ten przykład oraz funkcję `Replace()` dla łańcuchów tekstowych do osiągnięcia oczekiwanego efektu.

LABORATORIUM 6. WALIDACJA DANYCH UŻYTKOWNIKA PO STRONIE APLIKACJI

Cel laboratorium:

Celem zajęć jest zapoznanie się z mechanizmami walidacji danych na podstawie atrybutów oraz wykorzystanie relacji pomiędzy tabelami w Entity Framework Core.

Liczba punktów możliwych do uzyskania: 8 punktów

Zakres tematyczny zajęć:

Ograniczenia danych i ich walidacja sterowana atrybutami,
Walidacja danych po stronie serwera i stronie klienta,
Obsługa relacji w Entity Framework Core,
Obsługa migracji w Entity Framework Core,
Modele pośrednie w komunikacji z użytkownikiem.

Pytania kontrolne:

1. Na czym polega walidacja danych pochodzących od użytkownika?
2. W jaki sposób zapewniana jest integralność danych w systemach bazodanowych?
3. Czym jest ViewModel w architekturze MVVM?

Zadanie 6.1. Dodawanie ograniczeń do pól modelu

W poprzednim laboratorium została przygotowana aplikacja, która nie wykorzystywała żadnego mechanizmu walidacji danych. Czasami niezbędne jest jednak określenie, że konkretne pola są wymagane lub muszą posiadać wartość z określonego zakresu. ASP.NET Core pozwala na wykorzystywanie mechanizm walidacji, także w połączeniu ze skryptami JavaScript, do sprawdzania poprawności danych zarówno po stronie serwera, jak i stronie klienta.

Skorzystaj z programu, który udało się przygotować na poprzednim laboratorium lub pobierz z platformy Moodle plik „Lab6.zip” i wyodrębnij jego zawartość. Jest to program identyczny do ostatniego zadania Laboratorium 5, pozbawiony wykorzystania mechanizmu sekretów dla uproszczenia przykładu.

Ograniczenia w stosunku do pól dodawane są za pomocą atrybutów, takich jak [Required], [Range(min, max)] lub [MaxLength(liczba)] w podobny sposób, jak w Laboratorium 5 wykorzystywany był atrybut [UIHint].

Zmodyfikuj plik `Movie.cs` z folderu `Movies/` i dodaj do niego ograniczenia:

- właściwość `Title` ma być wymagana i mieć co najwyżej 50 znaków,
- właściwość `Description` ma być wymagana,
- właściwość `Rating` ma mieć zakres od 1 do 5.

Typ `string` w C#10 jest opisywany jako typ, który musi zawsze zawierać wartość – co najwyżej będzie to wartość pusta (pusty łańcuch), ale nie może przyjąć wartości `null`. Aby dopuścić, że właściwość może zawierać wartość `null` należy wykorzystać inny typ, `Nullable<T>`, poprzez wykorzystanie znaku `?`. Zmodyfikuj właściwość `TrailerLink`, aby była typu pozwalającego na `null` (*nullable type*), zapisując to w następujący sposób:



```
public string? TrailerLink { get; set; }
```

Do właściwości Id dodaj atrybut [Key], który określi, że będzie ona kluczem głównym.

Wykonane zmiany i dodane ograniczenia spowodują, że struktura modelu danych i struktura bazy danych nie będą sobie odpowiadać, należy zatem przygotować i przeprowadzić migrację. Uruchom okno terminala w Visual Studio lub otwórz terminal w głównym folderze swojego projektu i wydaj komendy:

```
dotnet ef migrations add Limits
dotnet ef database update
```

Spowoduje to dodanie nowej migracji, która zdefiniuje na nowo wymagalność i niewymagalność pól oraz zaktualizuje bazę danych do nowego schematu.

Spróbuj uruchomić swoją aplikację i sprawdź, czy działają ograniczenia przy tworzeniu i edycji filmów.

Zadanie 6.2. Własne opisy błędów

Możesz zauważyć, że błędy są wyświetlane w języku angielskim w standardowy sposób. Aby zastosować własny komunikat błędu, możesz ustawić właściwość ErrorMessage atrybutu walidacji, na przykład tak:

```
[UIHint("Stars")]
[Range(1, 5, ErrorMessage = "Ocena filmu musi być liczbą
pomiedzy 1 a 5")]
public int Rating { get; set; }
```

Spróbuj zmodyfikować aplikację tak, aby wyświetlane były własne komunikaty błędów dla wszystkich pól.

Zadanie 6.3. Walidacja po stronie klienta

Możesz zauważyć, że domyślnie walidacja jest wykonywana po stronie serwera – aplikacja przesyła formularz i dopiero wtedy następuje sprawdzanie czy został on wypełniony poprawnie. Aby dodać skrypt JavaScript, który będzie współpracował z mechanizmem walidacji w ASP.NET Core można zastosować bibliotekę Unobtrusive Validation, która wewnętrznie opiera się o bibliotekę jQuery Validation, której kopia jest dołączona do generowanego projektu MVC z domyślnego szablonu.

Aby włączyć wykorzystanie skryptów do walidacji, w pliku Views/Shared/_Layout.cshtml poniżej wszystkich elementów HTML <script> dodaj:

```
<partial name="_ValidationScriptsPartial" />
```

Spowoduje to dodanie kolejnych odpowiednich elementów <script>, które odpowiadają za mechanizm walidacji danych. Analizując kod HTML generowany przez ASP.NET Core możesz zwrócić uwagę, że walidacja opiera się o atrybuty danych (data attributes) postaci data-val-.*.



Przetestuj, czy walidacja działa teraz także po stronie klienta, aplikacja nie powinna wykonywać przesyłania danych z formularza, kiedy zawiera niepoprawne dane.

Zadanie 6.4. Dodawanie kolejnej tabeli oraz relacji pomiędzy tabelami w bazie danych

Chcemy dodać do systemu jeszcze jeden element – film będzie posiadał jeszcze jedną właściwość, tj. jego gatunek (np. horror). Chcemy jednak, aby mechanizm ten opierał się o jeszcze jedną tabelę w systemie baz danych, na podstawie której będą proponowane gatunki lub użytkownik będzie mógł wprowadzić własny, jak zaprezentowano na rysunku 7.1.

Rating

0

TrailerLink

Genre

a

fantasy

action

[Back to List](#)

Rys 7.1. Proponowanie nazw gatunków filmowych

Należy zacząć od dodania nowego modelu danych – dodaj nową publiczną klasę w folderze `Models/` o nazwie `Genre`, która będzie posiadać dwie właściwości – `Id` oraz `Name`.

```
public class Genre
{
    [Key]
    public int Id { get; set; }

    public string Name { get; set; }
}
```

W klasie kontekstu danych, `MovieDbContext`, dodaj odwołanie do tabel bazodanowej gatunków filmowych dodając właściwość:

```
public DbSet<Genre> Genres { get; set; }
```

Następnie, w klasie `Movie` dodaj odwołanie do gatunku filmowego, dodając właściwość:

```
public Genre Genre { get; set; }
```

Jak się domyślasz, należy przeprowadzić migrację bazy danych, ponieważ model danych uległ zmianie. Otwórz terminal i wydaj komendę:



```
dotnet ef migrations add Genre
```

Zostaną przygotowane wszystkie kolumny nowej tabeli i powiązania pomiędzy tabelami, nie jest jednak możliwe wykonanie komendy aktualizacji bazy danych, ponieważ nastąpi błąd, jeżeli w bazie danych są już jakieś rekordy – ponieważ właściwość `Genre` nie dopuszcza wartości `NULL`, i nie mamy zdefiniowanego domyślnego gatunku filmowego oraz nie mamy mechanizmu aktualizacji tej kolumny w bazie, nastąpi błąd na poziomie integralności klucza obcego, gdyby wykonać polecenie aktualizacji bazy (`dotnet ef database update`), jak zaprezentowano na rysunku 7.2.

```

"Title" TEXT NOT NULL,
"TrailerLink" TEXT NULL,
CONSTRAINT "FK_Movies_Genres_GenreId" FOREIGN KEY ("GenreId") REFERENCES "Genres" ("Id") ON DELETE CASCADE
);
fail: Microsoft.EntityFrameworkCore.Database.Command[20102]
Failed executing DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
INSERT INTO "ef_temp_Movies" ("Id", "Description", "GenreId", "Rating", "Title", "TrailerLink")
SELECT "Id", "Description", "GenreId", "Rating", "Title", "TrailerLink"
FROM "Movies";
Failed executing DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
INSERT INTO "ef_temp_Movies" ("Id", "Description", "GenreId", "Rating", "Title", "TrailerLink")
SELECT "Id", "Description", "GenreId", "Rating", "Title", "TrailerLink"
FROM "Movies";
Microsoft.Data.Sqlite.SqliteException (0x80004005): SQLite Error 19: 'FOREIGN KEY constraint failed'.
at Microsoft.Data.Sqlite.SqliteException.ThrowExceptionForRC(Int32 rc, sqlite3 db)
at Microsoft.Data.Sqlite.SqliteDataReader.NextResult()
at Microsoft.Data.Sqlite.SqliteCommand.ExecuteReader(CommandBehavior behavior)
at Microsoft.Data.Sqlite.SqliteCommand.ExecuteReader()
at Microsoft.Data.Sqlite.SqliteCommand.ExecuteNonQuery()
at Microsoft.EntityFrameworkCore.Storage.RelationalCommand.ExecuteNonQuery(RelationalCommandParameterObject parameter
Object)
at Microsoft.EntityFrameworkCore.Migrations.MigrationCommand.ExecuteNonQuery(IRelationalConnection connection, IReadOnly
Dictionary`2 parameterValues)
at Microsoft.EntityFrameworkCore.Migrations.Internal.MigrationCommandExecutor.ExecuteNonQuery(IEnumerable`1 migration
Commands, IRelationalConnection connection)
at Microsoft.EntityFrameworkCore.Migrations.Internal.Migrator.Migrate(String targetMigration)
at Microsoft.EntityFrameworkCore.Design.Internal.MigrationsOperations.UpdateDatabase(String targetMigration, String c
onnectionString, String contextType)
at Microsoft.EntityFrameworkCore.Design.OperationExecutor.UpdateDatabaseImpl(String targetMigration, String connectio
nString, String contextType)
at Microsoft.EntityFrameworkCore.Design.OperationExecutor.UpdateDatabase.<>c__DisplayClass0_0.<.ctor>b__0()
at Microsoft.EntityFrameworkCore.Design.OperationExecutor.OperationBase.Execute(Action action)
SQLite Error 19: 'FOREIGN KEY constraint failed'.
E:\_t\Lab6>

```

Rys 7.2. Błąd aktualizacji bazy danych

W tym zadaniu problem można by rozwiązać wydając przed komendą aktualizacji bazy komendę:

```
dotnet ef database drop
```

Spowoduje to jednak usunięcie wszystkich danych z bazy danych, co nie jest odpowiednim podejściem w systemach produkcyjnych. Niezbędne zatem jest przygotowanie ręcznego obejścia do problemu.

Zadanie 6.5. Modyfikacja migracji Genre

Migracje generowane są jako klasy w folderze `Migrations/`. Znajdź plik migracji o nazwie „Genre”, jego nazwa pliku będzie miała postać „data_Genre.cs”. W pliku tym, w metodzie `Up()` zmodyfikuj domyślną wartość dodawanej kolumny `GenreId` na 1.



Następnie, po poleceniu tworzenia tabeli Genres (postaci `migrationBuilder.CreateTable(name: "Genres", ...)` dodaj ręczne dodawanie danych:

```
migrationBuilder.InsertData(
    "Genres",
    new string[] { "Id", "Name" },
    new object[] { "1", "unknown" }
);
```

Teraz, możesz już wykonać aktualizację bazy danych do nowego schematu, wydając komendę:

```
dotnet ef database update
```

A w jej wyniku wszystkie filmy, które nie miały przypisanego gatunku uzyskają gatunek „unknown”.

Możesz uruchomić aplikację, ale możesz zauważyć, że nie działa ona poprawnie, ponieważ formularz dodawania nowego filmu i edycji starego nie działają, a gatunek filmowy nie jest nigdzie wyświetlany.

Zadanie 6.6. Wyświetlanie gatunku filmowego na liście

Zmodyfikuj widok `Views/Home/Index.cshtml` dodając jeszcze jedną kolumnę w nagłówku tabeli na przedostatnim miejscu:

```
<th>
    @Html.DisplayNameFor(model => model.Genre)
</th>
```

Oraz wypełniając ją zawartością wewnątrz pętli `foreach`:

```
<td>
    @Html.DisplayFor(modelItem => item.Genre.Name)
</td>
```

Niezbędne jest jednak przekazanie do widoku modelu danych, który będzie zawierał wypełnioną wartość `Genre`, jako że domyślnie nie jest pobierana z bazy danych (*lazy loading*).

W kontrolerze `HomeController`, w metodzie `Index`, zmodyfikuj wybieranie wartości z bazy danych, aby uwzględniało także gatunki filmowe:

```
_context.Movies.Include(x => x.Genre).ToListAsync()
```

Zadanie 6.7. Obsługa dodawania nowego filmu i jego gatunku filmowego

Możesz zauważyć, że model danych `Movie` różni się od tego, co chcielibyśmy uzyskać – zawiera właściwość typu `Genre`, podczas gdy chcemy, aby użytkownik miał możliwość ręcznego wpisania wartości do pola tekstowego, które zostanie zmapowane na

istniejący rekord w bazie lub na nowododany rekord. Niezbędne zatem będzie przygotowanie innego modelu reprezentującego encję bazodanową oraz obiekt odbierany od użytkownika. Takie podejście nazywa się modelami pośrednimi, DTO (*Data Transfer Objects*) lub czasami używane jest określenie ViewModel na wzór architektury MVVM w której również istnieją takie klasy pośrednie pomiędzy modelem i widokiem.

Przygotuj nową klasę, **MovieDto**, która będzie prawie identyczna jak klasa **Movie**:

```
public class MovieDto
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Tytuł jest wymagany")]
    [MaxLength(50)]
    public string Title { get; set; }

    [UIHint("LongText")]
    [Required(ErrorMessage = "Opis jest wymagany")]
    public string Description { get; set; }

    [UIHint("Stars")]
    [Range(1, 5, ErrorMessage = "Ocena filmu musi być liczbą
    pomiędzy 1 a 5")]
    public int Rating { get; set; }

    public string? TrailerLink { get; set; }

    public string Genre { get; set; }
}
```

Możesz zauważyć, że różni się tylko brakiem atrybutu **[Key]** oraz właściwością **Genre** która tutaj jest typu **string**.

Zmodyfikuj w **HomeController** metodę **Create()** (są dwie – chodzi o tę oznaczoną atrybutem **[HttpPost]**), aby jej parametr wejściowy nie był typu **Movie**, ale **MovieDto**. Następnie, do listy elementów w atrybucie **[Bind]** parametru wejściowego dodaj **Genre**, aby miało to finalnie następującą postać:

```
public async Task<IActionResult> Create(
    [Bind("Id,Title,Description,Rating,TrailerLink,Genre")]
    MovieDto movie
)
```



Niezbędne będzie przygotowanie mapowania pomiędzy typem przekazanym przez widok, a typem bazodanowym. Wewnątrz warunku dla poprawności przekazanego modelu od użytkownika (`ModelState.IsValid`) możesz skorzystać z następującego kodu:

```
var genre = _context.Genres.FirstOrDefault(x => x.Name ==
movie.Genre);
if (genre == null)
{
    genre = new Genre { Id = 0, Name = movie.Genre };
}

Movie m = new Movie
{
    Id = 0,
    Title = movie.Title,
    Description = movie.Description,
    Rating = movie.Rating,
    TrailerLink = movie.TrailerLink,
    Genre = genre
};

_context.Add(m);
await _context.SaveChangesAsync();
```

Kod ten wyszukuje gatunek filmowy w liście gatunków, lub tworzy nowy, jeżeli on nie istnieje, a następnie tworzy obiekt klasy `Movie`, tj. odpowiednika encji bazodanowej, przepisując do niego cechy z obiektu klasy `MovieDto`.

Możesz zwrócić uwagę, że takie ręczne przepisywanie nie jest najlepszym podejściem, zwłaszcza, jeżeli biblioteka będzie posiadała wiele takich cech. W środowiskach produkcyjnych stosuje się w takich przypadkach automatyczne biblioteki takie jak `Automapper` (<https://automapper.org/>).

W widoku `Views/Home/Create.cshtml` dodaj jeszcze jedno pole typu `<input>` odpowiedzialne za gatunek filmowy:

```
<div class="form-group">
    <label asp-for="Genre" class="control-label"></label>
    <input asp-for="Genre" class="form-control" />
    <span asp-validation-for="Genre" class="text-
danger"></span>
</div>
```

Oraz zmień model zdefiniowany w dyrektywie `@model` na szczycie pliku na `MovieDto`, aby widok opierał się już nie o model bazodanowy, ale o utworzony tutaj model pośredni.

Zadanie 6.8. Wykorzystanie elementu `<datalist>` do tworzenia podpowiedzi

Chcemy, aby użytkownik miał możliwość wybrania już istniejącego elementu z listy gatunków, który zostanie mu lub jej podpowiedziany po wprowadzeniu fragmentu tekstu do pola typu `<input>`. Można tutaj wykorzystać element `<datalist>` ze standardu HTML5. Niezbędne jest jednak przekazanie wszystkich obecnych w bazie gatunków filmowych do generowania tej listy.

Dodaj do klasy `MovieDto` jeszcze jedną właściwość:

```
public List<string>? AllGenres { get; set; }
```

Następnie zmodyfikuj widok `Views/Home/Create.cshtml`, aby generował element `<datalist>` wewnątrz elementu `<div>` w którym znajduje się `<input>` dla `Genre` za pomocą następującego podejścia:

```
<datalist id="genres">
    @foreach (var item in Model.AllGenres)
    {
        @Html.Raw($"<option value=\"{item}\">")
    }
</datalist>
```

Wreszcie, dodaj do elementu `<input>` dla `Genre` atrybut `list` wskazujący na element `<datalist>`:

```
<input asp-for="Genre" class="form-control" list="genres" />
```

Ostatnim krokiem będzie wypełnianie listy `AllGenres` przy generowaniu formularza dodawania nowego filmu. W `HomeController`, w metodzie `Create()` (wariancie nieoznaczonym atrybutem `[HttpPost]`) możesz to zrealizować w następujący sposób:

```
public IActionResult Create()
{
    var m = new MovieDto { AllGenres =
        _context.Genres.Select(x => x.Name).ToList() };
    return View(m);
}
```

Spróbuj uruchomić swoją aplikację aby przetestować jej działanie.

Zadanie 6.9. Modyfikacja formularza edycji filmu

Zmodyfikuj widok `Home/Views/Edit.cshtml` oraz metody `Edit()` w kontrolerze na wzór zadania 6.8, tak, aby ustawianie gatunku filmowego działało także w edycji rekordu.



Podpowiedź: standardowa metoda do wyszukiwania elementu na liście używa metody `Find()`, tutaj będzie niezbędne zastosowanie wyboru danych w bardziej rozbudowany sposób:

```
var movie = _context.Movies.Include(x =>
x.Genre).FirstOrDefault(x => x.Id == id);
```

Przetestuj działanie swojej aplikacji.



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



LABORATORIUM 7. TWORZENIE APLIKACJI INTERNETOWEJ PRZY WYKORZYSTANIU PODEJŚCIA „DATABASE FIRST”

Cel laboratorium:

Celem zajęć jest opracowanie aplikacji, która połączy się z istniejącą bazą danych za pomocą generatora modeli z bazy oraz pozwoli na zarządzanie użytkownikami z wykorzystaniem ASP.NET Core Identity.

Liczba punktów możliwych do uzyskania: 11 punktów

Zakres tematyczny zajęć:

Generowanie i wykorzystanie ASP.NET Core Identity,
Generowanie modeli na podstawie istniejącej bazy danych,
Wykorzystanie Entity Framework Core i adnotacji danych do prezentacji danych.

Pytania kontrolne:

1. Czym jest uwierzytelnianie, a czym jest autoryzacja użytkownika?
2. Czym jest kontekst danych (DbContext) w Entity Framework Core?
3. W jaki sposób korzysta się z relacji wiele-do-wielu w bazach danych?

Zadanie 7.1. Generowanie i testowanie projektu Identity

W tym laboratorium skorzystamy z biblioteki ASP.NET Core Identity, odpowiedzialnej za obsługę uwierzytelnienia i autoryzacji użytkowników. Pozwala ona na stosowanie uwierzytelnienia opartego o nazwę użytkownika i hasło, tokeny JWT i inne, jest rozszerzalna i dostarcza wbudowane mechanizmy obsługi rejestracji, logowania, konta użytkownika i wylogowania, także uwzględniając takie elementy systemu jak uwierzytelnianie dwuskładnikowe (2FA) czy potwierdzenie adresu e-mail hasłem. Identity obsługuje zarówno lokalne konta użytkowników, z których będzie korzystało to laboratorium, jak i zewnętrzne podsystemy uwierzytelniania takie jak OAuth czy uwierzytelnienie domeny Windows lub Azure AD. W tym laboratorium wygenerowana będzie aplikacja korzystająca z lokalnego systemu kont użytkowników.

Wariant dla Visual Studio

Domyślny kreator projektu używany w Visual Studio tworzy projekt z wykorzystaniem bazy danych SQL Server LocalDB, jednak dla celów tego laboratorium chcemy skorzystać z projektu wykorzystującego bazę danych SQLite, stąd musisz utworzyć nowy projekt korzystając z narzędzia konsolowego – zacznij od uruchomienia narzędzia „Wiersz polecenia” lub „Terminal Windows” i wybrania odpowiedniego folderu dla swojego projektu z wykorzystaniem poleceń `cd` i `md`. Następnie **postępuj zgodnie z poleceniami dla wariantu dla narzędzi konsolowych** i po utworzeniu projektu otwórz plik `.csproj` w środowisku Visual Studio.

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Utwórz folder, w którym ma zostać wygenerowany projekt np. o nazwie Lab7 i przejdź do niego. Wydadaj polecenie:



```
dotnet new mvc -au Individual --no-https -f net6.0
```

Spowoduje to wygenerowanie projektu ASP.NET Core z wykorzystaniem wzorca MVC oraz dodaną obsługą kont użytkowników zapisywanych w lokalnej bazie danych. Szablon automatycznie zawiera migracje bazodanowe, które pozwolą na wygenerowanie bazy danych.

Zadanie 7.2. Modyfikacja domyślnej obsługi użytkownika

W tym laboratorium chcemy, aby użytkownik w systemie Identity był powiązany z użytkownikiem, który będzie istniał w drugiej, już istniejącej, bazie danych, stąd niezbędne jest przechowywanie dodatkowych informacji, tutaj – identyfikatora klienta w innym systemie. W Identity jest możliwość dodania kolejnych cech użytkownika poprzez utworzenie klasy dziedziczącej po użytkowniku domyślnym.

Możesz zauważyć, że projekt Identity zawiera pliki w folderze `Areas/Identity`. Mechanizm obszarów (*areas*) pozwala na podzielenie bardziej skomplikowanej aplikacji na mniejsze fragmenty odpowiedzialne za konkretny obszar aplikacji.

Utwórz nowy folder, `Areas/Identity/Data`, a w nim nową publiczną klasę o nazwie `ApplicationUser`, dziedziczącą po `IdentityUser` z dodaną jedną publiczną właściwością typu `long` o nazwie `CustomerId`.

Następnie, musimy zmodyfikować podsystem Identity aby wykorzystywał klasę `ApplicationUser`:

- W pliku `Data/ApplicationDbContext.cs` zmień, aby klasa `ApplicationDbContext` dziedziczyła po klasie `IdentityDbContext<ApplicationUser>` zamiast `IdentityDbContext`,
- W pliku `Program.cs` zmodyfikuj wywołanie metody `builder.Services.AddDefaultIdentity()` aby korzystało z klasy `ApplicationUser`,
- W pliku `Views/Shared/_LoginPartial.cshtml` zmień dyrektywy `@inject` aby wykorzystywały klasę `ApplicationUser` – w razie potrzeby użyj pełnej nazwy klasy, np. `Lab7.Areas.Identity.Data.ApplicationUser`.

Teraz można już wykonać migrację bazodanową i utworzyć bazę danych – przejdź do narzędzia terminal w Visual Studio lub użyj zewnętrznej konsoli i w głównym folderze swojego projektu wydaj komendy:

```
dotnet ef migrations add ApplicationUser  
dotnet ef database update
```

Spowoduje to dodanie nowej migracji bazodanowej oraz aktualizację bazy danych. Możesz teraz uruchomić aplikację i utworzyć nowe konto. Zwróć uwagę na wymagania dotyczące hasła oraz na zaimplementowaną zaślepkę mechanizmu potwierdzenia adresu e-mail dla konta.

Zadanie 7.3. Generowanie modelu z istniejącej bazy danych

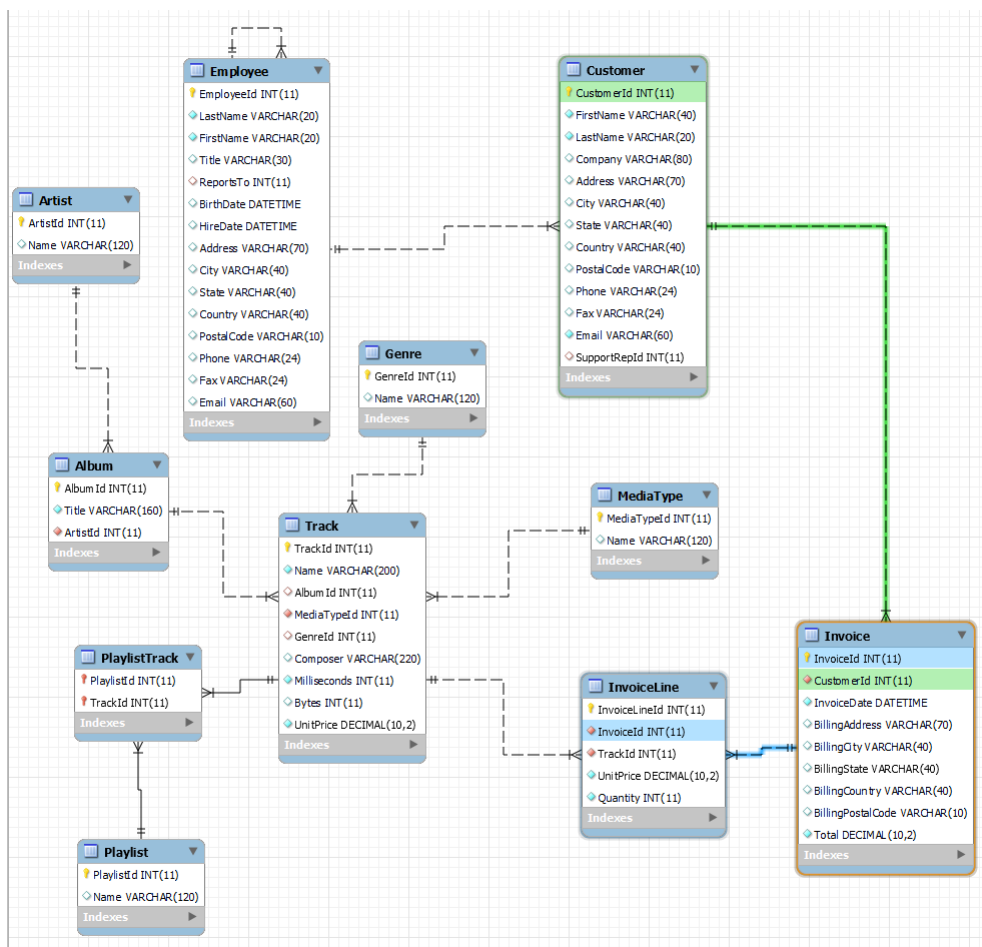
W systemach produkcyjnych podejście „code first”, gdzie najpierw jest opisywana baza danych za pomocą klas i właściwości, a potem generowana, nie zawsze jest możliwe do



wykonania – bazy danych często już są gotowe i tworzona aplikacja musi się do niej podłączyć. Entity Framework potrafi wykonać operację generowania kodu i opisu modeli danych na podstawie istniejącej bazy danych.

W tym laboratorium zostanie wykorzystana przykładowa baza danych „Chinook” (<https://github.com/lerocha/chinook-database>), dostępna na wolnej licencji MIT, reprezentująca system sprzedaży utworów muzycznych, podobny do usługi Apple Music/iTunes Store. Chinook jest alternatywą dla popularnego przykładu „Northwind” i jest dostępna dla MySQL, SQL Server, Oracle, PostgreSQL oraz SQLite.

Struktura bazy danych, która zostanie wykorzystana w zadaniu została przedstawiona na rysunku 7.1, poniżej.



Rys 7.1. Diagram ERD bazy „Chinook”

Baza danych zawiera 11 tabel, które posiadają różne informacje takie jak pracownicy, klienci i wykonane przez nich zakupy utworów, same utwory, ich typy plików, powiązania z albumami i artystami oraz listami utworów.

Pobierz bazę danych w postaci pliku `chinook.db` z platformy Moodle i umieść w głównym folderze swojej aplikacji.

Następnie, uruchom narzędzie terminal w Visual Studio lub ekran konsoli i w głównym folderze swojej aplikacji wydaj polecenie:

```
dotnet ef dbcontext scaffold "Data Source=chinook.db"
Microsoft.EntityFrameworkCore.Sqlite -c ChinookDbContext -o
Models --context-dir Data
```

Wygenerowane zostaną modele twojej aplikacji do folderu **Models**, a klasa kontekstu danych do folderu **Data**. Aby wygenerować modele z istniejącej bazy należy podać ciąg *connection string* dotyczący połączenia do niej, jak i nazwę klasy-dostawcy obsługi konkretnej bazy danych dla Entity Framework Core. Możesz zauważyć, że wygenerowany plik **ChinookDbContext.cs** zawiera polecenie kompilatora **#warning**, które dodaje ostrzeżenie, że *connection string* nie powinien być zawarty bezpośrednio w kodzie – zamiast tego możesz skorzystać z mechanizmu sekretów, tak samo jak wykorzystuje go domyślny projekt zawierający Identity lub możesz usunąć ostrzeżenie z kodu, ponieważ w tym przykładzie nie ma to znaczenia.

Następnie, w pliku **Program.cs** musisz zarejestrować drugi kontekst danych w mechanizmie *dependency injection*, dodając linię:

```
builder.Services.AddDbContext<ChinookDbContext>();
```

Poniżej rejestracji **ApplicationDbContext**.

Teraz, spróbuj wyświetlić kilku klientów z bazy Chinook w swojej aplikacji. W klasie **HomeController**, dodaj wstrzykiwanie kontekstu bazy danych modyfikując konstruktor i pola klasy:

```
private readonly ILogger<HomeController> _logger;
private readonly ChinookDbContext _chinook;

public HomeController(ILogger<HomeController> logger,
ChinookDbContext chinook)
{
    _logger = logger;
    _chinook = chinook;
}
```

Natomiast w metodzie **Index()** przekaz do widoku listę klientów:

```
return View(_chinook.Customers.ToList());
```

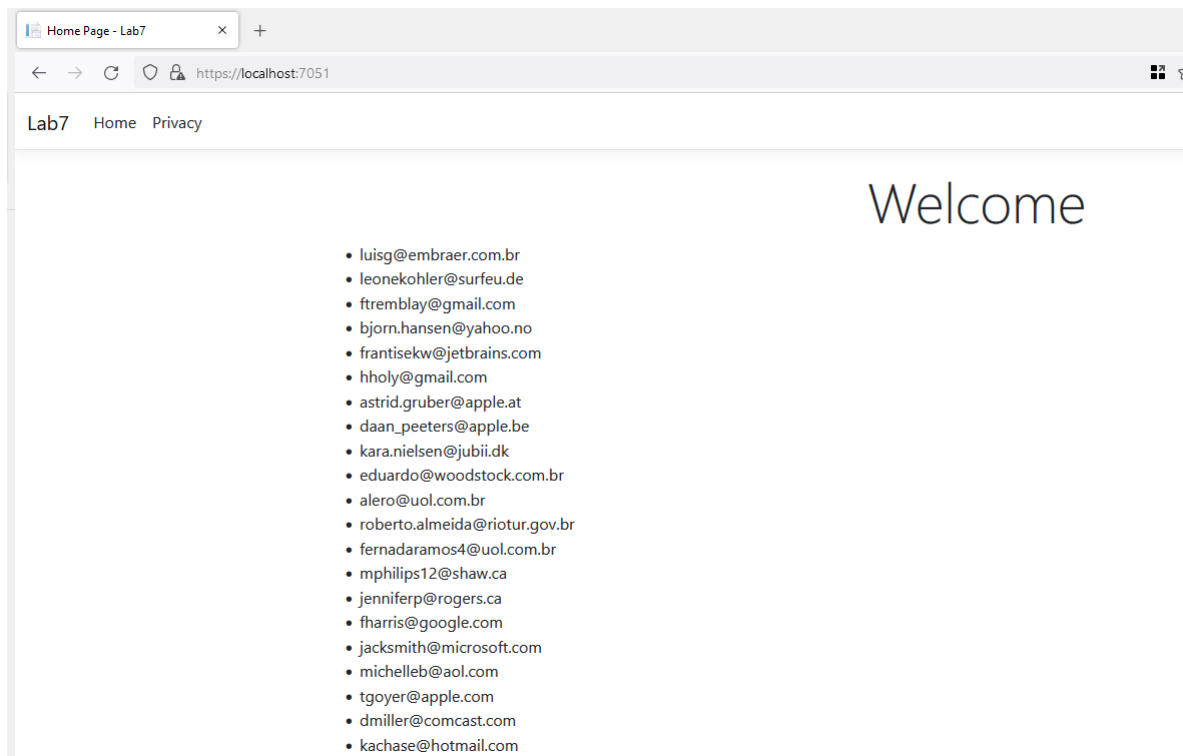
W widoku, **Views/Home/Index.cshtml** skorzystaj z listy nieuporządkowanej do wyświetlenia elementów:

```
<div>
    <ul>
        @foreach (var item in Model)
        {
            <li>@item.Email</li>
        }
    </ul>
</div>
```




```
</ul>
</div>
```

Spróbuj uruchomić aplikację, główna strona aplikacji powinna prezentować się w następujący sposób:



Rys 7.2. Przykładowe działanie aplikacji

Zadanie 7.4. Dodawanie klientów z bazy Chinook do Identity

W tym laboratorium chcemy, aby użytkownicy zapisani w bazie danych Chinook mogli zalogować się poprzez system Identity i mogli przeglądać swoje zamówienia. Aby powiązać klientów pomiędzy systemami każdy rejestrujący się nowy klient powinien uzyskiwać zmianę pola CustomerId w klasie ApplicationUser. Klienci jednak już są utworzeni w bazie danych Chinook, więc należy ich zaimportować do Identity.

W tym przykładzie każdy importowany użytkownik uzyska hasło P@ssw0rd (wielkość liter ma znaczenie). Dodaj do pliku Program.cs, przed liniijką app.Run();, następujący kod:

```
using (var scope = app.Services.CreateScope())
{
    using (var context =
scope.ServiceProvider.GetService<ChinookDbContext>())
    {
        var userManager =
scope.ServiceProvider.GetService<userManager<ApplicationUser>>
();
    }
}
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny




```
        if (
            await userManager.FindByEmailAsync(
                context.Customers.OrderBy(x =>
x.CustomerId).First().Email
            ) == null
        )
        {
            foreach (var item in context.Customers)
            {
                var user = new ApplicationUser
                {
                    UserName = item.Email,
                    NormalizedUserName = item.Email,
                    Email = item.Email,
                    NormalizedEmail = item.Email,
                    EmailConfirmed = true,
                    LockoutEnabled = false,
                    SecurityStamp = Guid.NewGuid().ToString(),
                    CustomerId = item.CustomerId
                };
                await userManager.CreateAsync(user,
"P@ssw0rd");
            }
        }
    }
}
```

Kod ten przy uruchomieniu aplikacji sprawdzi, czy klienci zostali już zaimportowani i jeśli nie, skorzysta z klasy `UserManager<>` generując dla nich konta w systemie Identity.

Uruchom swoją aplikację i spróbuj zalogować się na któreś z kont klientów podając jako login e-mail z prezentowanej na głównej stronie listy i hasło `P@ssw0rd`.

Zadanie 7.5. Wyświetlanie listy zamówień zalogowanego klienta

W klasie `HomeController` dodaj nową metodę o nazwie `MyOrders()`, która będzie prezentowała listę zamówień aktualnie zalogowanego klienta. Możesz skorzystać z następującego kodu:

```
[Authorize]
```

```
public async Task<IActionResult> MyOrders()
{
    var user = await _userManager.GetUserAsync(User);
    var customerId = user.CustomerId;
    return View(await _chinook.Invoices.Where(x =>
x.CustomerId == customerId).ToListAsync());
}
```

Atrybut `[Authorize]` wymusza, aby użytkownik był zalogowany przed odwiedzeniem tej metody i spowoduje przekierowanie do strony logowania, jeżeli użytkownik nie jest zalogowany.

Następnie, wygeneruj widok `Views/Home/MyOrders.cshtml` korzystając z opcji w menu kontekstowym Visual Studio lub narzędzia `dotnet aspnet-codegenerator`. Wygeneruj widok oparty o model `Invoice` i szablon `List`.

Następnie, zmodyfikuj wygenerowany szablon aby usunąć hiperłącza do „Create”, „Edit” i „Delete” oraz aby identyfikator faktury był odnośnikiem do strony z jej szczegółami tak samo jak łączy o nazwie „Details” – akcja powinna nazywać się `OrderDetails`.

Wreszcie, w widoku `Views/Home/Index.cshtml` dodaj odnośnik do akcji `MyOrders`, aby łatwo przejść na tę stronę. Najłatwiej skorzystać będzie z mechanizmu *tag helpers*:

```
<a asp-action="MyOrders">My Orders</a>
```

Teraz możesz uruchomić swoją aplikację, zalogować się i przejść do adresu `/Home/MyOrders`. Powinna pojawić się strona aplikacji podobna do tej na rysunku 7.3.

My Orders

InvoiceId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalCode	Total	
112	System.Byte[]	627 Broadway	New York	NY	USA	10012-2612	49465756	Details
135	System.Byte[]	627 Broadway	New York	NY	USA	10012-2612	51465754	Details
157	System.Byte[]	627 Broadway	New York	NY	USA	10012-2612	53465752	Details
209	System.Byte[]	627 Broadway	New York	NY	USA	10012-2612	48465757	Details
330	System.Byte[]	627 Broadway	New York	NY	USA	10012-2612	49465756	Details
341	System.Byte[]	627 Broadway	New York	NY	USA	10012-2612	4951465654	Details
396	System.Byte[]	627 Broadway	New York	NY	USA	10012-2612	56465749	Details

Rys 7.3. Działanie aplikacji

Zadanie 7.6. Modyfikacja wyświetlania zamówień

Problem jaki widzisz – niepoprawne wyświetlanie daty i kwoty – jest spowodowany szczegółami działania generatora Entity Framework Core – w definicji bazy danych kolumna `InvoiceDate` jest zdefiniowana jako `DATETIME`, jednak generator nie potrafił wygenerować odpowiedniego typu. Podobny problem w kolumnie `Total` dotyczy typu `NUMERIC(10,2)`. Generator nie zawsze bezbłędnie buduje mapowania pomiędzy bazą danych i obiektami, jest to też zależne od typu systemu zarządzania bazą danych, jak i zastosowanej biblioteki.



Zmodyfikuj klasę `Invoice` zmieniając ręcznie `InvoiceDate` na typ `DateTime`, a `Total` na typ `decimal`. Dodatkowo, zastosuj atrybuty `[DataType]` aby ulepszyć wyświetlanie danych.

```
[DataType(DataType.DateTime)]
public DateTime InvoiceDate { get; set; }

[DataType(DataType.Currency)]
public decimal Total { get; set; }
```

Zmodyfikuj wyświetlanie faktur na liście zamówień w taki sposób, aby wyświetlana była tylko data zamówienia (korzystając np. z `ToShortDateString()`). Aplikacja powinna działać jak na rysunku 7.4.

My Orders

InvoiceId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalCode	Total	
112	12.05.2010	627 Broadway	New York	NY	USA	10012-2612	1,98 zł	Details
135	14.08.2010	627 Broadway	New York	NY	USA	10012-2612	3,96 zł	Details
157	16.11.2010	627 Broadway	New York	NY	USA	10012-2612	5,94 zł	Details
209	07.07.2011	627 Broadway	New York	NY	USA	10012-2612	0,99 zł	Details
330	28.12.2012	627 Broadway	New York	NY	USA	10012-2612	1,98 zł	Details
341	07.02.2013	627 Broadway	New York	NY	USA	10012-2612	13,86 zł	Details
396	08.10.2013	627 Broadway	New York	NY	USA	10012-2612	8,91 zł	Details

Rys 7.4. Poprawione działanie aplikacji

Następnie, skorzystaj z atrybutu `[Display]` aby dodać opisy do kolumn, które będą wykorzystywane przy wyświetlaniu elementów klasy `Invoice`, na przykład tak:

```
[Display(Name = "Billing Address")]
public string? BillingAddress { get; set; }
```

Następnie, tak zmodyfikuj wyświetlanie faktur na liście, aby uzyskać tylko 5 kolumn:

- identyfikator będący odnośnikiem do szczegółów,
- data,
- pełny adres w kolejności: ulica, miasto, stan, kraj,
- kwota łączna,
- odnośnik do strony szczegółów.

Zadanie 7.7. Modyfikacja ustawień regionalnych aplikacji

Możesz zauważyć, że prezentacja daty oraz ceny używają lokalnego dla serwera formatu danych, np. w języku polskim. Chcemy, aby aplikacja działała tylko w systemie angielskim amerykańskim. Problem wielojęzyczności aplikacji (globalizacja i lokalizacja) jest bardzo szerokim zagadnieniem. Mechanizm `[Display]` wraz z odpowiednimi słownikami łańcuchów to jeden z elementów rozwiązania tego problemu, który nie zostanie zastosowany



w tym laboratorium. Drugim elementem jest wymuszenie konkretnego *locale* (kultury) aplikacji – w tym przykładzie wymusimy zawsze obsługę *locale* en-us, tj. języka angielskiego z amerykańskimi opcjami regionalnymi.

W pliku Program.cs, powyżej wywołania metody dodającej obsługę Identity, builder.Services.AddDefaultIdentity() dodaj:

```
builder.Services.AddLocalization();
```

Natomiast poniżej wywołania var app = builder.Build(); dodaj:

```
var supportedCultures = new[] { new CultureInfo("en-US") };
app.UseRequestLocalization(
    new RequestLocalizationOptions
    {
        DefaultRequestCulture = new RequestCulture("en-US"),
        SupportedCultures = supportedCultures,
        FallBackToParentCultures = false
    }
);
CultureInfo.DefaultThreadCurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
```

W rezultacie wykonania tego i poprzedniego zadania aplikacja powinna prezentować listę zamówień w sposób, jaki zaprezentowano na rysunku 7.5.

My Orders

Invoice ID	Invoice Date	Billing Address	Total	
112	5/12/2010	627 Broadway 10012-2612, New York, NY, USA	\$1.98	Details
135	8/14/2010	627 Broadway 10012-2612, New York, NY, USA	\$3.96	Details
157	11/16/2010	627 Broadway 10012-2612, New York, NY, USA	\$5.94	Details
209	7/7/2011	627 Broadway 10012-2612, New York, NY, USA	\$0.99	Details
330	12/28/2012	627 Broadway 10012-2612, New York, NY, USA	\$1.98	Details
341	2/7/2013	627 Broadway 10012-2612, New York, NY, USA	\$13.86	Details
396	10/8/2013	627 Broadway 10012-2612, New York, NY, USA	\$8.91	Details

Rys 7.5. Oczekiwanie działanie aplikacji z amerykańskimi ustawieniami regionalnymi

Zadanie 7.8. Strona szczegółów zamówienia

W kontrolerze HomeController dodaj nową akcję, która będzie pobierać i zwracać do widoku wszystkie informacje o fakturze o danym identyfikatorze.

```
[Authorize]
public async Task<IActionResult> OrderDetails(int id)
{
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
var user = await _userManager.GetUserAsync(User);
var customerId = user.CustomerId;

var invoice = _chinook.Invoices
    .Include(x => x.InvoiceLines)
    .ThenInclude(x => x.Track)
    .FirstOrDefault(x => x.InvoiceId == id);

return View(invoice);
}
```

Możesz zauważyć, że wykorzystywane są tutaj metody `Include()` oraz `ThenInclude()`, które pobierają z bazy danych elementy włączając tabele podrzędne (operacją JOIN języka SQL). Następnie, wygeneruj widok `Views/Home/OrderDetails.cshtml` korzystając z modelu `Invoice` i szablonu `Details`.

Z rysunku 7.1 możesz zauważyć, że każda faktura połączona jest z utworem za pomocą relacji wiele-do-wielu poprzez tabelę wiążącą `InvoiceLines`. Rekordy w tabeli `InvoiceLines` zawierają odwołanie do utworu (ścieżki), które tutaj w EF jest dołączane przez `ThenInclude()` oraz m.in. informacje o cenie konkretnego utworu (ścieżki).

Zmodyfikuj widok `OrderDetails.cshtml` aby osiągnąć rezultat przedstawiony na rysunku 7.6 – pokazuj tylko niezbędne elementy informacji o fakturze, wyświetlaj datę i adres w sposób podobny do głównej listy i wyświetl listę utworów wchodzących w skład zamówienia w postaci tabeli. Może być niezbędne zmienienie typu danych w modelu `InvoiceLines` z `byte[]` na `decimal`.

Order Details

Invoice

Invoice ID	112		
Invoice Date	5/12/2010		
Billing Address	627 Broadway 10012-2612 New York NY USA		
Total	\$1.98		
Ordered Songs	Track ID	Track	Price
	208	Terra	\$0.99
	209	Eclipse Oculito	\$0.99

[Back to List](#)

Rys 7.6. Oczekiwanie działanie aplikacji

Zadanie 7.9. Ograniczenia przeglądania zamówień

Akcja `OrderDetails()` nie posiada obecnie ograniczeń – jeżeli ręcznie odwołasz się do adresu URL zawierającego identyfikator faktury, to zostanie ona wyświetlona, lub aplikacja wyrzuci wyjątek, jeżeli faktura nie istnieje. Użytkownik nie powinien mieć jednak możliwości dostępu do faktury innego użytkownika.



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Korzystając z faktu, że funkcja `FirstOrDefault()` zwraca null jeżeli faktura nie zostanie znaleziona oraz ze znajomości identyfikatora aktualnie zalogowanego użytkownika przygotuj aplikację w taki sposób, aby zwracała jako swój wynik np. `Forbid()` lub `NotFound()` jeżeli użytkownik próbuje odwołać się do nieistniejącej lub „nieswojej” faktury/zamówienia.

Zadanie 7.10. Zmiana ścieżek w mechanizmie routingu

W obecnej chwili aby wejść na stronę z listą zamówień, użytkownik musi odwiedzić adres `http://example.com/Home/MyOrders` oraz `http://example.com/Home/OrderDetails/12`. Chcemy zmienić domyślny schemat routingu. W pliku `Program.cs`, powyżej definicji `app.MapControllerRoute(name: "default", pattern: "{controller=Home}/{action=Index}/{id?}");` dodaj następujący kod:

```
app.MapControllerRoute(
    name: "orderslist",
    pattern: "orders",
    defaults: new { controller = "Home", action = "MyOrders" }
);
app.MapControllerRoute(
    name: "order",
    pattern: "orders/{id}",
    defaults: new { controller = "Home", action =
"OrderDetails" }
);
```

W tym przykładzie „name” oznacza nazwę, która jest jednoznacznym identyfikatorem ścieżki routingu. „pattern” służy do wyznaczenia wyrażenia, do którego adres URL ma się dopasować, natomiast „defaults” wyznacza obiekt anonimowy definiujący kontroler, akcję i ewentualne domyślne parametry. Spróbuj uruchomić swoją aplikację z nowymi ścieżkami. Jeżeli wcześniej użyte były mechanizmy takie jak `@Html.ActionLink()` lub `<a asp-for>` ścieżki powinny zaktualizować się automatycznie.

Zadanie 7.11. Widoczność tylko dla osób zalogowanych

Kiedy wylogujesz się z aplikacji i spróbujesz odwiedzić `http://example.com/orders` aby przejść do listy zamówień, mechanizm Identity dzięki atrybutowi `[Authorize]` automatycznie przekieruje do strony logowania. Chcemy jednak ograniczyć widoczność odwołania na stronie głównej, aby pojawiało się tylko, kiedy użytkownik jest zalogowany.

Kiedy zanalizujesz zawartość pliku `Views/Shared/_LoginPartial.cshtml`, zauważysz, że wykorzystywana jest tam instrukcja warunkowa zależna od `SignInManager.IsSignedIn(User)`. Dodatkowo, jest wykorzystywana klauzula `@inject`. `@inject` służy do wstrzyknięcia do widoku obiektu pochodzącego z mechanizmu wstrzykiwania zależności.

Skorzystaj z tych elementów w pliku `Views/Home/Index.cshtml` i wyświetlaj odnośnik do listy zamówień tylko jeśli użytkownik jest zalogowany.

LABORATORIUM 8. TWORZENIE APLIKACJI INTERNETOWEJ DOSTARCZAJĄCEJ WEB API

Cel laboratorium:

Celem zajęć jest opracowanie aplikacji internetowej w stylu REST API, przetwarzającej dane w formacie JSON.

Liczba punktów możliwych do uzyskania: 8 punktów

Zakres tematyczny zajęć:

Tworzenie projektu WebAPI,
Wykorzystanie bazy danych LiteDB i wzorca Repozytorium,
Obsługa żądań HTTP,
Obsługa uwierzytelnienia typu Basic.

Pytania kontrolne:

1. Na czym polega podejście SPA do budowy aplikacji internetowych?
2. Do czego służy format danych JSON?
3. Jakie są dostępne typy żądań (*verbs*) w HTTP?

Zadanie 8.1. Tworzenie projektu WebAPI

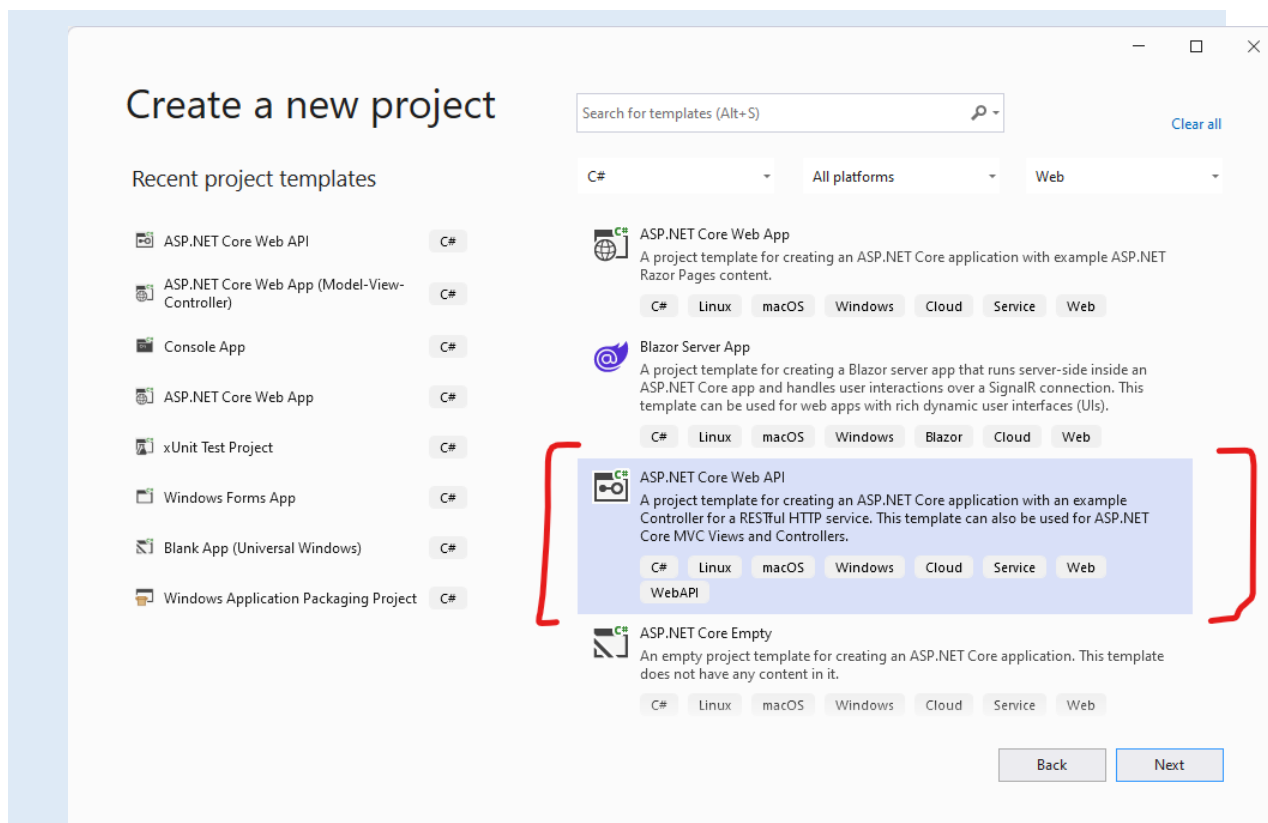
WebAPI to alternatywne podejście w stosunku do tworzenia aplikacji internetowej w podejściu MVC lub Razor Pages – opiera się nie o generowanie wyjściowego kodu HTML, a o udostępnianie przez aplikację serwerową danych w postaci podejścia REST (Representational State Transfer), gdzie obiekty są najczęściej serializowane do postaci dokumentów JSON (JavaScript Object Notation), co pozwala na współpracę z innymi systemami i językami programowania. Dodatkowo, WebAPI udostępnia także mechanizm OpenAPI, który służy do opisu zdalnego API, demonstrując jakie są dostępne zasoby i typy operacji, które mogą być na nich wykonywane. W trybie pracy dla programistów udostępniany jest także interfejs Swagger w postaci aplikacji internetowej, który może posłużyć do testowania tworzonego API – a oprócz tego do testów będzie można wykorzystywać narzędzia takie jak Postman lub Insomnia.

W tym zadaniu zacznij od wygenerowania nowego projektu aplikacji WebAPI.

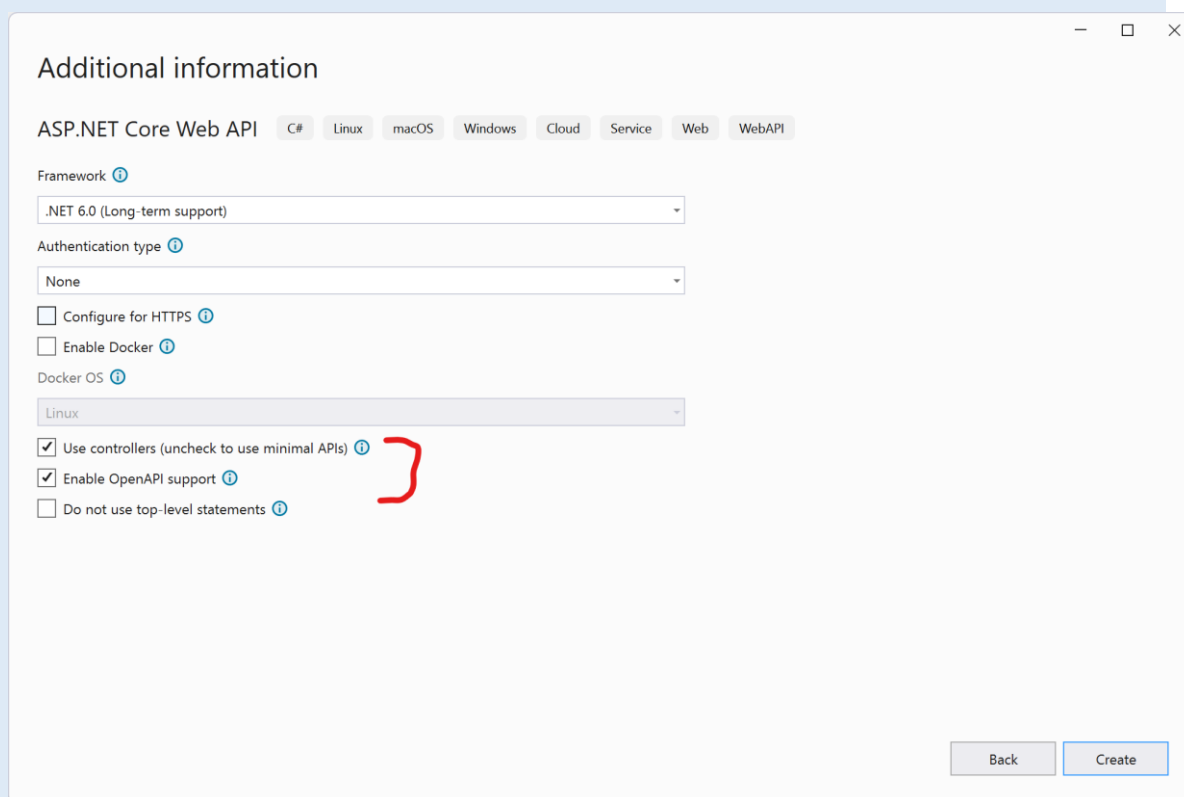
Wariant dla Visual Studio

Wygeneruj nową aplikację, używając szablonu „ASP.NET Core WebAPI”. Następnie, na kolejnym ekranie, wybierz nazwę dla swojej aplikacji (np. Lab8). Na ostatnim ekranie, przedstawionym na rysunku 8.2, zaznacz opcje „Use controllers” oraz „Enable OpenAPI support” (i **odznacz** opcję wykorzystania HTTPS).





Rys 8.1. Wybór szablonu projektu WebAPI



Rys 8.2. Wybór elementów szablonu WebAPI

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Utwórz nowy folder, w którym znajdzie się twoja aplikacja, np. o nazwie Lab8 i przejdź do niego w narzędziu terminal, a następnie wydaj komendę:

```
dotnet new webapi --no-https -f net6.0
```

Wygenerowany szablon zawiera przykładowy kontroler w folderze Controllers/ o nazwie WeatherForecastController, jak i model WeatherForecast, które możesz usunąć.

Zadanie 8.2. Dodawanie interfejsu oraz klasy repozytorium danych

W tym zadaniu do przechowywania danych zostanie wykorzystana dokumentowa, nierelacyjna, plikowa baza danych LiteDB (<https://litedb.org>). LiteDB jest wieloplatformowym systemem do przechowywania danych, który nie jest dostępny w systemie ORM Entity Framework, ale dostarcza swój własny interfejs podobny w stylu działania.

Jednakże, aby aplikacja była niezależna od wykorzystywanego systemu bazodanowego, często stosowany jest mechanizm ORM lub wzorzec Repozytorium, który ukrywa implementację przed logiką biznesową aplikacji. W tym laboratorium zostanie wykorzystany wzorzec Repozytorium, który pozwoli na „podmianę” systemu persystencji danych w razie potrzeby.

Utwórz folder Models/ i utwórz w nim nową publiczną klasę, Fox, która będzie posiadać 4 właściwości:

Id(int),
Name(string),
Loves(int)
oraz Hates(int).

Model danych będzie przechowywał imię zwierzęcia oraz przypisaną do niego liczbę „polubieni” i „znielubień”, które później posłużą do sortowania wyników.

Dodaj do swojego projektu nową bibliotekę z repozytorium NuGet, o nazwie LiteDB.

Następnie, utwórz folder Data/, a w nim dodaj nowy interfejs IFoxesRepository, który będzie wyglądał następująco:

```
public interface IFoxesRepository
{
    void Add(Fox f);
    Fox Get(int id);
    IEnumerable<Fox> GetAll();
    void Update(int id, Fox f);
}
```

Teraz, dodaj nową klasę, FoxesRepository, implementującą interfejs IFoxesRepository. Będzie ona mogła wyglądać następująco, aby wykorzystywać bazę LiteDB:



```
public class FoxesRepository : IFoxesRepository
{
    private readonly LiteDatabase db;

    public FoxesRepository()
    {
        db = new LiteDatabase(@"foxes.litedb");
    }

    public void Add(Fox f)
    {
        var all = db.GetCollection<Fox>();
        f.Id = all.Count() > 0 ? all.Max(x => x.Id) + 1 : 1;
        db.GetCollection<Fox>().Insert(f);
    }

    public Fox Get(int id)
    {
        return db.GetCollection<Fox>().FindById(id);
    }

    public IEnumerable<Fox> GetAll()
    {
        return db.GetCollection<Fox>().FindAll();
    }

    public void Update(int id, Fox f)
    {
        var c = db.GetCollection<Fox>().FindById(id);
        c.Loves = f.Loves;
        c.Hates = f.Hates;
        db.GetCollection<Fox>().Update(c);
    }
}
```



Wreszcie, w pliku `Program.cs`, powyżej wywołania metody `Build()` dodaj rejestrację klasy implementującej interfejs:

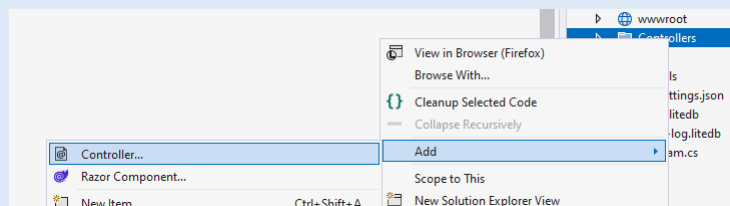
```
builder.Services.AddSingleton<IFoxesRepository,  
FoxesRepository>();
```

Zadanie 8.3. Obsługa zwracania i dodawania danych

Wygeneruj nowy, pusty kontroler API, o nazwie `FoxController`.

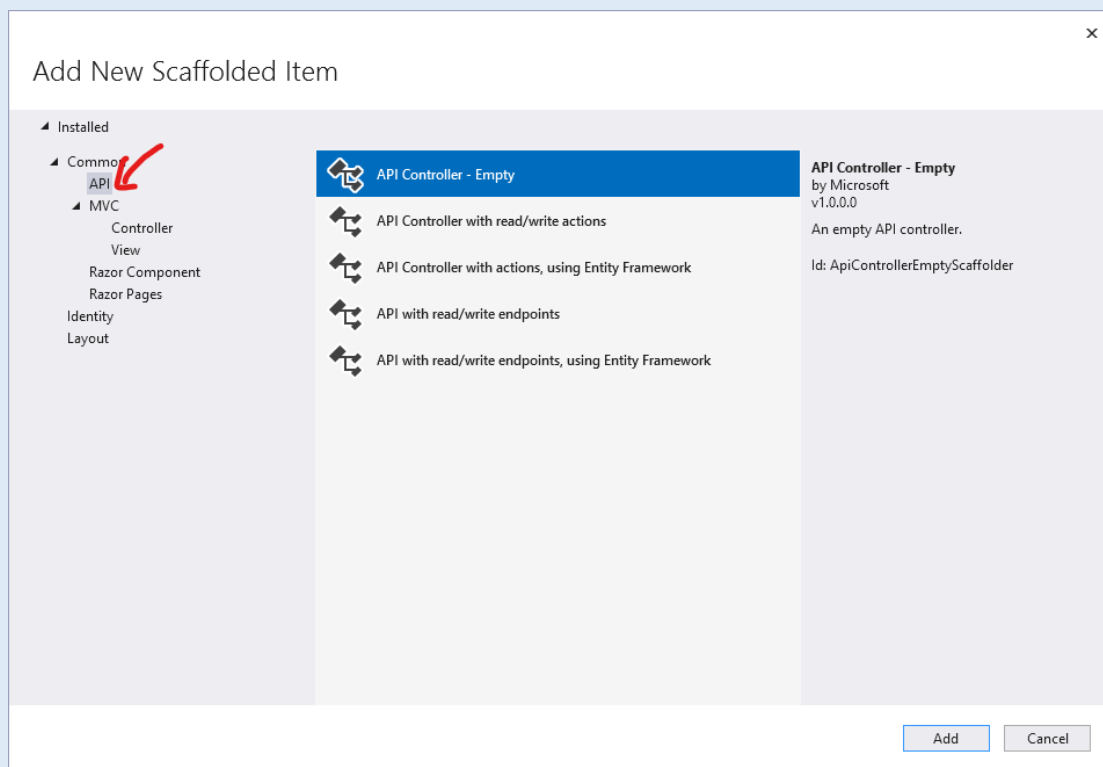
Wariant dla Visual Studio

Wybierz z menu kontekstowego folderu `Controllers/` opcję `Add -> Controller`.



Rys 8.3. Opcja dodawania kontrolera

Na kolejnym ekranie przedź do kategorii `API` i wybierz „API Controller – Empty”.



Rys 8.4. Wybór pustego szablonu kontrolera API

Wariant dla Visual Studio Code oraz narzędzi konsolowych

Przejdź do głównego folderu swojej aplikacji i wydaj komendę:



```
dotnet add package  
Microsoft.VisualStudio.Web.CodeGeneration.Design --version 6.0
```

Aby zainstalować generator, a następnie wydać komendę:

```
dotnet aspnet-codegenerator controller -name FoxController -  
api -outDir Controllers
```

Aby wygenerować pusty kontroler API.

W wygenerowanym kontrolerze niezbędne będzie:

- Dodanie pola typu `IFoxesRepository`,
- Obsługa wstrzykiwania zależności, aby obiekt typu `IFoxesRepository` został wstrzyknięty do konstruktora, a potem wykorzystywany dalej,
- Dodanie funkcji obsługujących pobieranie wszystkich obiektów, pobieranie pojedynczego oraz dodawanie nowego.

Do opracowania funkcji pobierania i przyjmowania danych możesz skorzystać z funkcji kontrolera takich jak `Ok()` czy `NotFound()`, które zwracają odpowiednie typy wyników (wraz z np. kodami 200 czy 404). Oprócz tego, możesz też przygotować funkcje aby zwracały po prostu typ `Fox` czy `List<Fox>` czy `IEnumerable<Fox>`, a ASP.NET Core sam „opakuje” to w przetworzenie danych do formatu JSON. Aby przyjąć dane wystarczy skorzystać z mechanizmu *model binding* i atrybutu `[FromBody]`.

Jedyną istotną zmianą będzie to, że funkcje będą musiały mieć atrybuty `[HttpGet]`, `[HttpPost]`, `[HttpGet("{id}")]` i inne, aby opisać ścieżkę żądania HTTP oraz typ (*verb*) żądania.

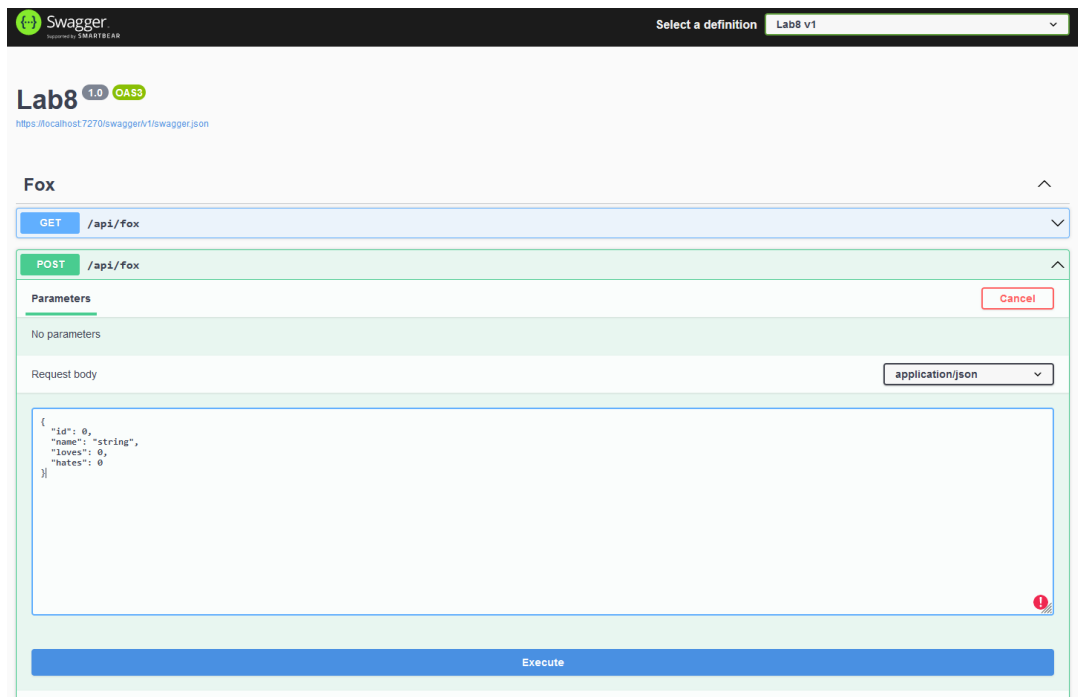
Przykładowo, aby dodać obsługę dodawania nowego obiektu możesz skorzystać z takiej funkcji:

```
[HttpPost]  
public IActionResult Post([FromBody] Fox fox)  
{  
    _repo.Add(fox);  
  
    return CreatedAtAction(nameof(Get), new { id = fox.Id },  
fox);  
}
```

Opracuj funkcje `Get()` zwracającą listę wszystkich obiektów, oraz `Get(int id)` zwracającą określony obiekt po podanym identyfikatorze.

Kiedy przygotujesz aplikację, możesz ją uruchomić – po odwiedzeniu adresu `/swagger` zostanie ci zaprezentowany interfejs testowy OpenAPI. Możesz go wykorzystać, aby dodać kilka przykładowych obiektów do bazy danych i spróbować je wybrać. Spróbuj też wykorzystać narzędzie takie jak curl, Postman, Insomnia lub Nightingale do pobrania danych z utworzonej aplikacji.





Rys 8.5. Przykład działania przeglądarki interfejsu API Swagger

Zadanie 8.4. Metody obsługi dodawania „polubień” i „zniełubień”

Kolejnym krokiem będzie dodanie akcji, które będą odpowiadać za zwiększenie liczby „polubień” oraz „zniełubień” danego obiektu. Wykorzystane tutaj będzie żądanie HTTP typu PUT, ale dla uproszczenia chcemy, aby identyfikator obiektu do aktualizacji był przekazywany w ścieżce żądania. Możesz skorzystać z następującego kodu jako wzorca:

```
[HttpPut("love/{id}")]
public IActionResult Love(int id)
{
    var fox = _repo.Get(id);

    if (fox == null)
        return NotFound();

    fox.Loves++;
    _repo.Update(id, fox);

    return Ok(fox);
}
```

Opracuj implementację dla metod Love() oraz Hate().

Zadanie 8.5. Zmiana i dostosowanie routingu

Standardowo generowany routing zakłada, że ścieżki będą zgodne z nazwą kontrolera, który zgodnie z konwencjami języka C# jest nazwany z wielkiej litery. W URL wielkość liter ma znaczenie, a typową konwencją są małe litery. Stąd, zmodyfikuj atrybut [Route] kontrolera aby jawnie używał słowa z małej litery:

```
[Route("api/fox")]
```

Następnie przetestuj, czy wszystkie operacje są pod adresami:

- /api/fox – GET powinno zwrócić wszystkie obiekty, POST powinno dodać nowy obiekt,
- /api/fox/1 – GET powinno zwrócić obiekt o identyfikatorze 1,
- /api/fox/love/1 – PUT powinno zwiększyć liczbę „polubień” dla obiektu o identyfikatorze równym 1,
- /api/fox/hate/1 – PUT powinno zwiększyć liczbę „zniechęceń” dla obiektu o identyfikatorze równym 1.

Zadanie 8.6. Dodawanie aplikacji klienckiej HTML

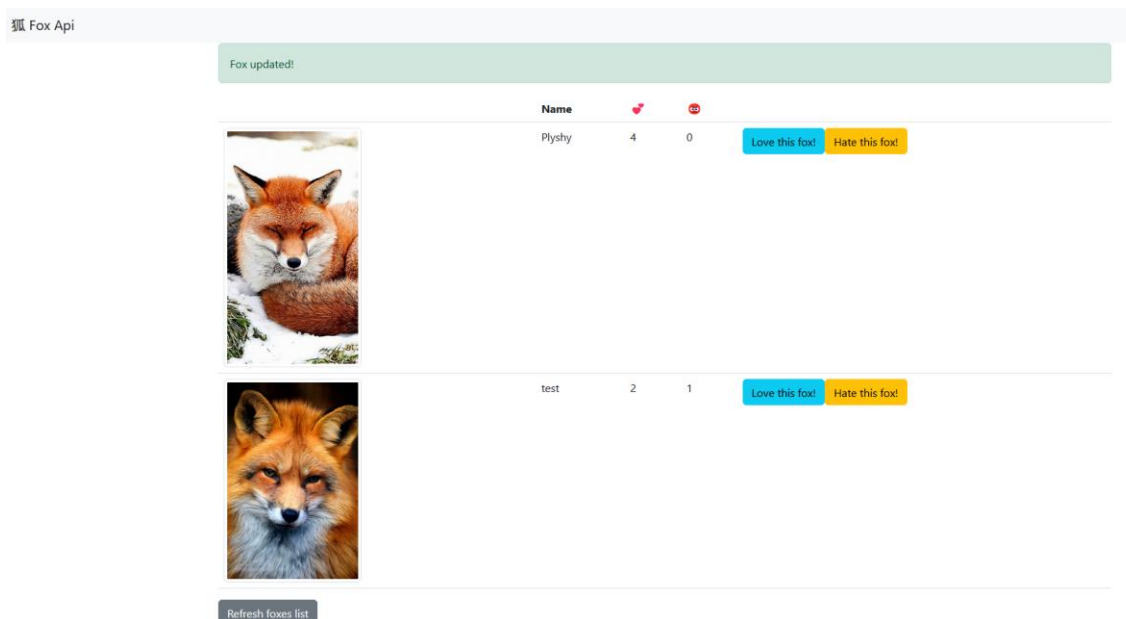
W tym zadaniu przygotowana zostanie aplikacja wykorzystująca API. Z platformy Moodle pobierz plik „wwwroot.zip” i utwórz na jego podstawie folder „wwwroot” w głównym folderze swojego projektu – wewnątrz niego powinien znaleźć się plik index.html zawierający aplikację przygotowaną w języku JavaScript oraz folder images/ zawierający przykładowe obrazki.

Następnie, aby aplikacja potrafiła wysyłać do przeglądarki dokument index.html oraz aby był traktowany jako domyślny po odwiedzeniu głównego adresu witryny, w pliku Program.cs dodaj:

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

Powyżej wywołania metody MapControllers().

Uruchom swoją aplikację i przejdź pod adres /. Jeżeli routing jest skonfigurowany poprawnie oraz dodane jest kilka przykładowych obiektów do bazy, aplikacja powinna prezentować się i działać jak zaprezentowano na rysunku 8.6.



Rys 8.6. Oczekiwanie działanie aplikacji

Zadanie 8.7. Obsługa uwierzytelnienia Basic

Chcemy dodać, aby tylko uwierzytelnieni użytkownicy mieli dostęp do dodawania nowych obiektów do bazy danych. W kolejnym laboratorium zostanie to odpowiednio przygotowane, tutaj zostanie wykorzystany prosty przykład uwierzytelnienia typu HTTP Basic. Określa ono, że klient HTTP wysyła do serwera żądanie przekazując nagłówek **Authorization**, którego wartość to nazwa użytkownika oraz hasło połączone znakiem : oraz zakodowane za pomocą algorytmu BASE64.

Metoda ta nie jest uznawana za bezpieczną, stąd nie powinna być wykorzystywana produkcyjnie, jednak zademonstrowany zostanie mechanizm dodania obsługi uwierzytelnienia.

Dodaj do swojego projektu bibliotekę `AspNetCore.Authentication.Basic`, a następnie w pliku `Program.cs` dodaj rejestrację obsługi uwierzytelnienia poniżej wywołania `builder.Services.AddControllers()`:

```
builder.Services
    .AddAuthentication(BasicDefaults.AuthenticationScheme)
    .AddBasic(options =>
    {
        options.Realm = "Fox API";

        options.Events = new BasicEvents
        {
            OnValidateCredentials = async (context) =>
            {
```



```
var user = context.Username;

var isValid = user == "user" &&
context.Password == "password";
if (isValid)
{
    context.Response.Headers.Add(
        "ValidationCustomHeader",
        "From OnValidateCredentials"
    );
    var claims = new[]
    {
        new Claim(
            ClaimTypes.NameIdentifier,
            context.Username,
            ClaimValueTypes.String,
            context.Options.ClaimsIssuer
        ),
        new Claim(
            ClaimTypes.Name,
            context.Username,
            ClaimValueTypes.String,
            context.Options.ClaimsIssuer
        )
    };
    context.Principal = new ClaimsPrincipal(
        new ClaimsIdentity(claims,
context.Scheme.Name)
    );
    context.Success();
}
else
{
    context.NoResult();
}
```



```
}  
};  
});
```

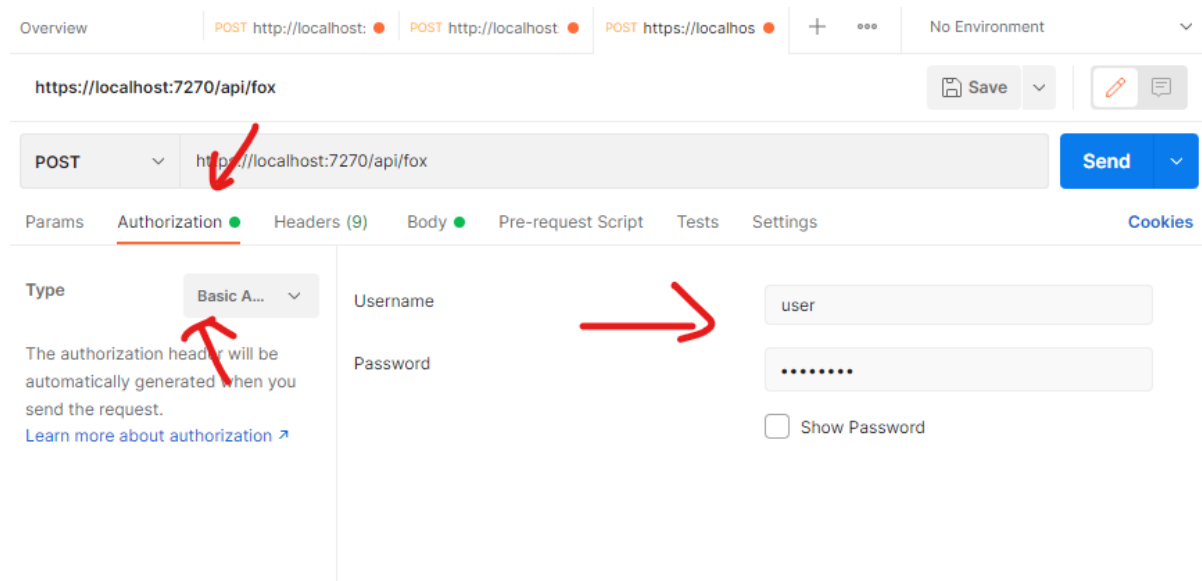
Jak zademonstrowano na przykładowym listingu, biblioteka stwierdza, że użytkownik jest poprawny, jeżeli nazwa to „user”, a hasło to „password”, po czym generowane są tzw. *claims*. Definiują one nazwę użytkownika i inne elementy, np. rolę czy funkcję, co pozwala na rozróżnianie użytkowników dalej w systemie.

Następnie, dodaj do Program.cs obsługę uwierzytelnienia i autoryzacji, powyżej wywołania UseFileServer():

```
app.UseAuthentication();  
app.UseAuthorization();
```

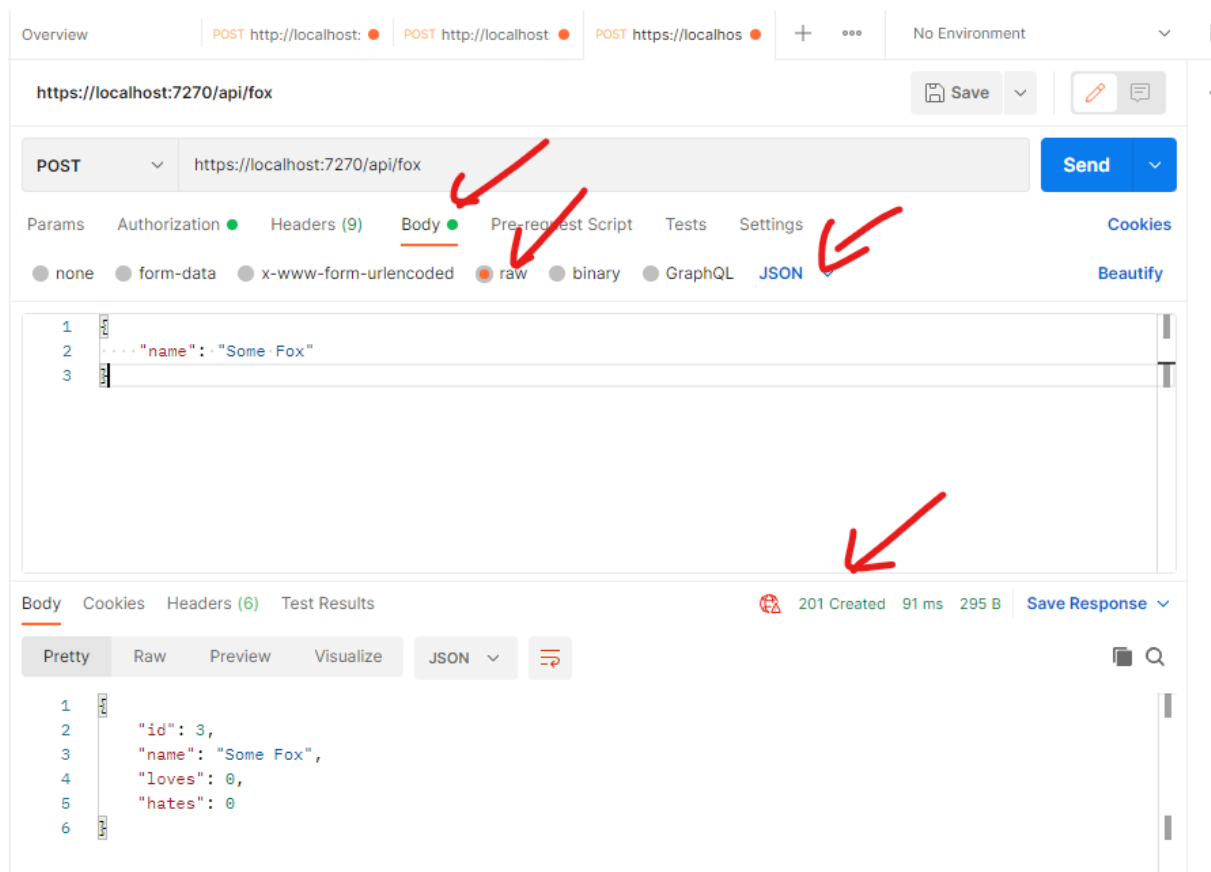
Uwaga: kolejność wywoływania operacji UseAuthentication() oraz UseAuthorization() ma bardzo duże znaczenie – wymagana jest najpierw obsługa uwierzytelnienia, zanim będzie można skorzystać z autoryzacji.

Teraz możesz dodać atrybut [Authorize] do akcji dodawania nowych obiektów w kontrolerze FoxController. Przetestuj działanie aplikacji z wykorzystaniem narzędzia do testowania API i sprawdź, czy zwracany jest kod 401 dla użytkownika niewierzytelnionego oraz poprawny kod 201 kiedy przekazana jest odpowiednia nazwa użytkownika i hasło.



Rys 8.7. Ustawianie hasła Basic Authentication w narzędziu Postman





Rys 8.8. Ustawianie treści żądania i kod poprawnej odpowiedzi 201 w narzędziu Postman

Zadanie 8.8. Zmiana sortowania obiektów

Chcemy, aby dane zwracane do klientów były sortowane w określony sposób, a nie wedle wewnętrznego identyfikatora. Zmodyfikuj kontroler i jego metodę `Get()` w taki sposób, aby dane były sortowane najpierw po liczbie „polubień” malejąco, a następnie po liczbie „zniechęceń”, rosnąco. Wykorzystaj do tego metody `OrderByDescending()` oraz `ThenBy()`.

LABORATORIUM 9. WYKORZYSTANIE UWIERZYTELNIENIA W APLIKACJI TYPU API

Cel laboratorium:

Celem zajęć jest opracowanie aplikacji internetowej, która wykorzystuje uwierzytelnianie i autoryzację za pomocą tokenów (znaczników) JWT.

Liczba punktów możliwych do uzyskania: 6 punktów

Zakres tematyczny zajęć:

Dodawanie podsystemu Identity do istniejącego projektu,
Modyfikacja wymagań podsystemu Identity,
Generowanie i testowanie tokenów JWT,
Uwierzytelnianie za pomocą tokenów JWT,
Sprawdzanie roli przechowywanej w tokenie.

Pytania kontrolne:

1. W jaki sposób działa mechanizm „ciasteczek” (*cookies*)?
2. W jaki sposób działają podpisy cyfrowe?
3. Na czym polega bezstanowość protokołu HTTP?

Zadanie 9.1. Dodawanie ASP.NET Identity do istniejącego projektu

Pobierz z platformy Moodle przykładowy projekt „Lab9”. Jest to projekt wzorowany na projekcie, który opracowywany był w Laboratorium 8, jednak został nieznacznie zmodyfikowany, m.in. implementacja interfejsu `IFoxesRepository` opiera się o listę w pamięci, a nie o rzeczywistą implementację bazy danych. Usunięty został też mechanizm uwierzytelnienia w oparciu o HTTP Basic Authentication.

W tym laboratorium będziemy wykorzystywać ASP.NET Identity do przechowywania danych użytkowników, stąd niezbędne będzie jego dodanie do już istniejącego projektu.

Przejdź do emulatora konsoli, narzędzia Terminal lub Wiersza Polecenia, a w nim do folderu projektu Lab9 i wydaj komendy:

```
dotnet add package
Microsoft.VisualStudio.Web.CodeGeneration.Design --version 6.0
dotnet add package Microsoft.EntityFrameworkCore.Design --
version 6.0
dotnet add package
Microsoft.AspNetCore.Identity.EntityFrameworkCore --version
6.0
dotnet add package Microsoft.AspNetCore.Identity.UI --version
6.0
dotnet add package Microsoft.EntityFrameworkCore.Sqlite --
version 6.0
```



```
dotnet add package Microsoft.EntityFrameworkCore.Tools --  
version 6.0  
  
dotnet aspnet-codegenerator identity -sqlite  
dotnet ef migrations add Init
```

Spowoduje to wygenerowanie elementów systemu Identity, utworzenie migracji schematu bazy danych i dołączenie niezbędnych bibliotek do działania systemu. Utworzone zostaną także migracje, które należy zmodyfikować w celu dodania wartości początkowych.

Otwórz plik w folderze Migrations/ o nazwie kończącej się na `_Init.cs` i w metodzie `Up()`, po utworzeniu wszystkich tabel, relacji i indeksów, dodaj następujący kod:

```
migrationBuilder.InsertData(  
    "AspNetRoles",  
    new string[] { "Id", "Name", "NormalizedName",  
"ConcurrencyStamp" },  
    new object[]  
    {  
        Guid.NewGuid().ToString(),  
        "Admin",  
        "ADMIN",  
        Guid.NewGuid().ToString()  
    }  
);
```

Spowoduje on dodanie danych do bazy przy aplikowaniu migracji – zostanie dodana rola o nazwie „Admin”. Role to mechanizm grupowania uprawnień do wykonywania czynności, wykorzystywany w ASP.NET Identity. W dalszej części laboratorium będziemy sprawdzać, czy użytkownik przynależy do pewnej roli.

W pliku `Program.cs` dodaj:

```
app.UseRouting();
```

Powyżej wywołania `UseAuthentication()`, natomiast poniżej `MapControllers()` dodaj:

```
app.MapRazorPages();
```

Spowoduje to możliwość działania jednocześnie kontrolerów API, jak i stron Razor Pages w jednej aplikacji – mechanizm Identity opiera się o Razor Pages.

Wykonaj utworzenie bazy danych wydając komendę w głównym folderze swojej aplikacji:

```
dotnet ef database update
```



Zadanie 9.2. Dostosowywanie mechanizmu Identity

Jak pamiętasz z poprzedniego laboratorium wykorzystującego Identity, domyślne wymagania dotyczące hasła są bardzo skomplikowane. Można to jednak zmodyfikować, modyfikując ustawienia podsystemu Identity.

W pliku `Program.cs` znajdź wywołanie dodawania obsługi podsystemu Identity, o nazwie `builder.Service.AddDefaultIdentity()` i je zmodyfikuj, używając następującego fragmentu kodu:

```
builder.Services
    .AddDefaultIdentity<IdentityUser>(options =>
    {
        options.SignIn.RequireConfirmedAccount = true;
        options.Password.RequireUppercase = false;
    })
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<Lab9IdentityDbContext>();
```

Funkcja `AddRoles()` powoduje uaktywnienie podsystemu ról. Z kolei ustawienia parametrów `options.Password` pozwalają na sterowanie wymogami dotyczącymi złożoności hasła.

Zmodyfikuj te opcje tak, aby nie były wymagane wielkie litery, znaki niealfanumeryczne i cyfry, ale za to minimalna długość hasła powinna wynosić 8 znaków – tak, aby mogło zadziałać np. hasło „password”.

Zadanie 9.3. Kontroler dostarczający tokeny dla użytkownika

Utwórz nową klasę, `UserDto`, która będzie posiadać dwie właściwości typu `string`: `UserName` oraz `Password`. Będzie ona modelem danych, który będzie przysyłał użytkownik do specjalnego kontrolera, który na tej podstawie będzie generował token, a użytkownik uwierzytelniał się tokenem od tej pory.

Dodaj do swojego projektu bibliotekę do obsługi tokenów JWT:

```
dotnet add package
Microsoft.AspNetCore.Authentication.JwtBearer --version 6.0
```

Utwórz nowy, pusty kontroler typu API o nazwie `UserController`. Dodaj w konstruktorze aby wstrzykiwane do niego były `userManager<IdentityUser>` oraz `IConfiguration`, które zapiszesz do prywatnych pól o nazwie `_user` oraz `_configuration`, odpowiednio.

Dodaj do kontrolera akcję `Token()`, która będzie zwracać użytkownikowi token na podstawie obiektu DTO:

```
[HttpPost]
public async Task<IActionResult> Token([FromBody] UserDto dto)
{
```



```
var user = await _user.FindByEmailAsync(dto.UserName);
var result = await _user.CheckPasswordAsync(user,
dto.Password);

if (result)
{
    var claims = new List<Claim>()
    {
        new Claim(JwtRegisteredClaimNames.Sub,
user.Email),
        new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
        new Claim(JwtRegisteredClaimNames.Email,
user.Email),
    };

    var key = new SymmetricSecurityKey(
        Encoding.UTF8.GetBytes(_configuration["Tokens:Key"
])
    );

    var creds = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(
        issuer: _configuration["Tokens:Issuer"],
        audience: _configuration["Tokens:Audience"],
        claims: claims,
        expires: DateTime.UtcNow.AddMinutes(60),
        signingCredentials: creds
    );

    return Ok(
        new
        {
            token = new
JwtSecurityTokenHandler().WriteToken(token),
            expiration = token.ValidTo
        }
    );
}
```

```

        );
    }
    else
    {
        return BadRequest("Login Failure");
    }
}

```

Wykorzystaj następujące przestrzenie nazw:

```

using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using Microsoft.IdentityModel.Tokens;

```

Metoda ta sprawdza czy kombinacja nazwy użytkownika i hasła jest poprawna, a jeśli tak, to generuje token JWT, wystawiając go dla konkretnego serwera i podpisując tajnym kluczem szyfrującym, który jest pobierany z konfiguracji. Ważność tokena została ustawiona na 60 minut.

Tokeny JWT (JSON Web Tokens) to otwarty standard przenoszenia informacji o użytkowniku. Ciąg znaków, którym jest *de facto* token, to podpisany cyfrowo zakodowany obiekt, który może informować kto go wystawił, dla jakich odbiorców, dla kogo i jakie informacje ma jego właściciel.

Stąd, w pliku `appsettings.json` dodaj konfigurację ustawień wymaganych przez ten mechanizm – informacje wykorzystywane jako dane dla wystawcy, odbiorcy i tajny klucz podpisu cyfrowego (możesz go zmienić):

```

"Tokens": {
  "Audience": "http://localhost:5010",
  "Issuer": "http://localhost:5010",
  "Key": "bardzo tajny klucz szyfrujący"
}

```

Ustawienia te są traktowane jako tajne i nie powinny być przechowywane w kodzie aplikacji. Tutaj stosujemy plik konfiguracyjny, w aplikacjach produkcyjnych dane te powinny być dostarczone przez mechanizm sekretów.

Z uwagi na to, że ustaliliśmy, że dostawca i odbiorca tokena będzie pod znanym adresem URL należy zmienić, aby aplikacja korzystała z tego adresu, a nie z losowo generowanego portu jak zazwyczaj, aby sprawdzanie wystawcy i odbiorcy tokena działało prawidłowo. W pliku `Properties/launchSettings.json` zmień opcję `applicationUrl` na <http://localhost:5010> wewnątrz profilu o nazwie Lab9.

Zadanie 9.4. Migracja użytkowników

W pliku `Program.cs` powyżej wywołania `app.Run()` dodaj fragment kodu:

```

using (var scope = app.Services.CreateScope())

```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny




```
{
    using (var roleManager =
scope.ServiceProvider.GetService<RoleManager<IdentityRole>>())
    using (var userManager =
scope.ServiceProvider.GetService<UserManager<IdentityUser>>())
    {
        roleManager.CreateAsync(new
IdentityRole("Admin")).Wait();

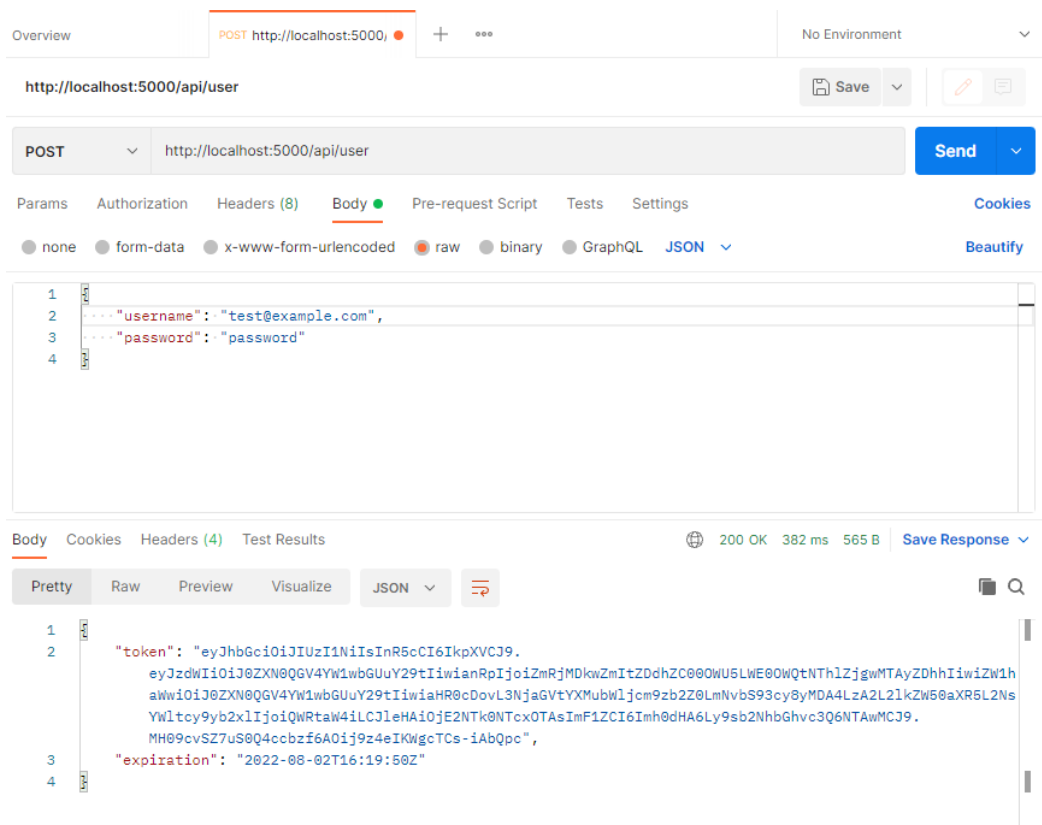
        foreach (var user in userManager.Users.Where(x =>
x.Email.EndsWith("@example.com")))
        {
            userManager.AddToRoleAsync(user, "Admin").Wait();
        }
    }
}
```

Spowoduje on, że wszyscy użytkownicy zapisani na e-mail kończący się na @example.com zostaną przeniesieni do roli „Admin”. W tym przykładzie wykorzystywany jest fragment kodu uruchamiany tuż przed właściwym uruchomieniem aplikacji – będzie to powodowało generowanie ostrzeżeń

Uruchom swoją aplikację i zarejestruj nowego użytkownika na adres e-mail kończący się w ten sposób, następnie zrestartuj ją, aby zadziałała migracja, i użyj narzędzia typu Postman lub Insomnia wysyłając żądanie:

- Typu POST, na adres <http://localhost:5010/api/user>,
- Z zawartością w postaci obiektu JSON: { "userName": "string", "password": "string" } (oczywiście zamiast „string” podaj nazwę użytkownika i hasło zarejestrowanego użytkownika),
- Z nagłówkiem Content-Type ustawionym na application/json.

Przykład w narzędziu Postman przedstawiono na rysunku 9.1.



Rys 9.1. Przykład żądania do pobrania tokenu JWT w narzędziu Postman.

Powinien zostać ci zwrócony obiekt, którego pierwsze pole, token, to długi ciąg znaków. Możesz wkleić ten ciąg znaków do aplikacji internetowej <https://jwt.io>, aby zobaczyć jego strukturę wewnętrzną:



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Algorithm

HS256

Encoded

PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXN0QGV4YW1wbGUuY29tIiwianRpIjoizmRjMDkwZmItZDdhZC00OWU5LWE0WQTNTh1ZjgwMTAyZDhhIiwiaWwiOiJ0ZXN0QGV4YW1wbGUuY29tIiwiaHR0cDovL3NjaGVtYXMuY29tIiwiaWF0IjoiMTY5ODU0MjY0In0=

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256",  "typ": "JWT"}
```

PAYLOAD: DATA

```
{  "sub": "test@example.com",  "jti": "fd090fb-d7ad-49e9-a49d-58ef80102d8a",  "email": "test@example.com",  "http://schemas.microsoft.com/ws/2008/06/identity/claims/role": "Admin",  "exp": 1659457190,  "aud": "http://localhost:5000"}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)
```

☐ secret base64 encoded

Zadanie 9.5. Uwierzytelnianie za pomocą tokena

W pliku `Program.cs` powyżej wywołania metody `builder.Build()` dodaj opcje uwierzytelnienia:

```
builder.Services
    .AddAuthentication()
    .AddCookie()
    .AddJwtBearer(
        JwtBearerDefaults.AuthenticationScheme,
        options =>
        {
```

```
options.TokenValidationParameters = new
TokenValidationParameters
{
    ValidateIssuer = true,
    ValidateAudience = true,
    ValidateLifetime = true,
    ValidateIssuerSigningKey = true,
    ValidIssuer =
builder.Configuration["Tokens:Issuer"],
    ValidAudience =
builder.Configuration["Tokens:Audience"],
    IssuerSigningKey = new SymmetricSecurityKey(
        Encoding.UTF8.GetBytes(builder.Configuration["Tokens:Key"]))
};
};
);

builder.Services.AddAuthorization();
```

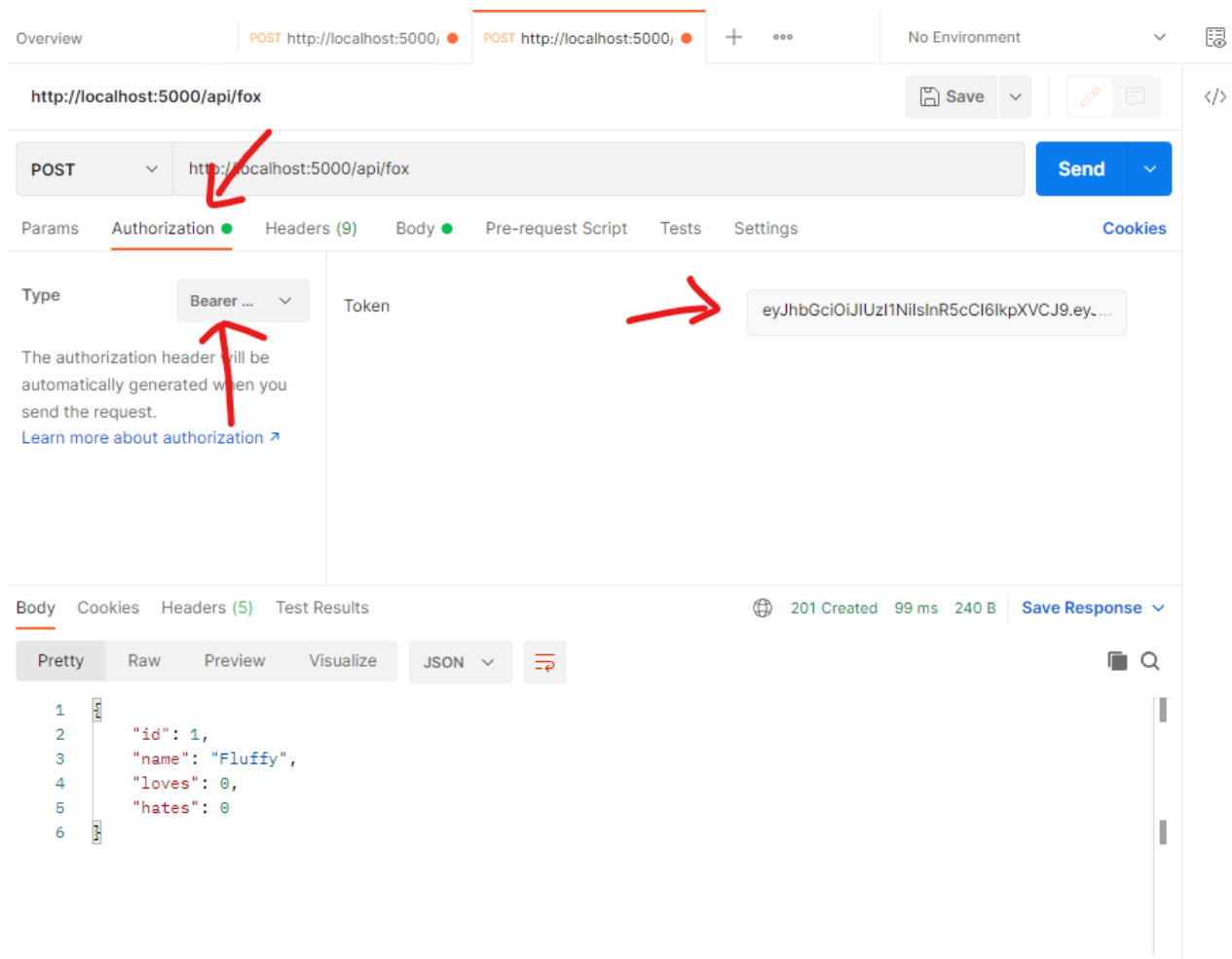
Definiowane tutaj jest, że token, aby mógł zostać zaakceptowany, musi być podpisany znanym kluczem, przeznaczony dla odpowiedniego odbiorcy, wystawiony przez znanego dostawcę i nie może być wygaśnięty. Wcześniej, aktywowany jest mechanizm *cookies* ze swoimi standardowymi parametrami.

Następnie, w metodzie akcji `Post()` kontrolera `FoxController` dodaj atrybut:

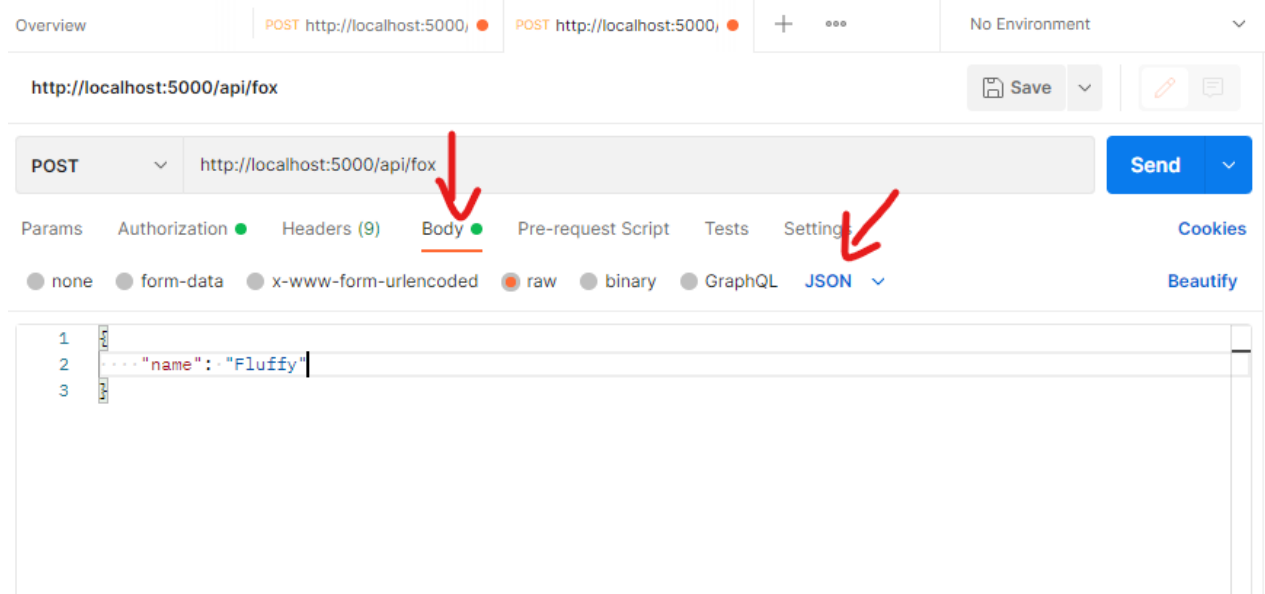
```
[Authorize(AuthenticationSchemes =
JwtBearerDefaults.AuthenticationScheme)]
```

Spowoduje to, że akcja ta będzie dostępna tylko dla użytkowników uwierzytelnionych za pomocą tokena.

Przetestuj działanie aplikacji, wysyłając żądanie uwierzytelnione tokenem, który zwrócił kontroler `UserController`, przykładowo w aplikacji Postman będzie to wyglądało następująco:



Rys 9.3. Ustawianie uwierzytelnienia w narzędziu Postman.



Rys 9.4. Ustawianie przesyłanych danych w narzędziu Postman



Zadanie 9.6. Dostosowywanie reguł bezpieczeństwa

Chcemy, aby akcje `Love()` i `Hate()` kontrolera `FoxController` były dostępne dla wszystkich użytkowników uwierzytelnionych za pomocą tokena, ale aby akcja `Post()` była dostępna tylko dla użytkowników będących w roli administratora.

Dodaj nagłówek uwierzytelnienia do akcji `Love()` i `Hate()`:

```
[Authorize(AuthenticationSchemes =  
JwtBearerDefaults.AuthenticationScheme)]
```

Następnie, w pliku `Program.cs` zmień wywołanie metody `AddAuthorization()`, aby było wykorzystywane wymaganie roli:

```
builder.Services.AddAuthorization(options =>  
{  
    options.AddPolicy(  
        "IsAdminJwt",  
        policy =>  
            policy  
                .RequireRole("Admin")  
                .AddAuthenticationSchemes(JwtBearerDefaults.Au  
thenticationScheme)  
    );  
});
```

A w nagłówku akcji `Post()` kontrolera `FoxController` użyj tych reguł bezpieczeństwa zmieniając atrybut `[Authorize]` na:

```
[Authorize("IsAdminJwt")]
```

Rola użytkownika musi być również zapisywana w tokenie, aby było możliwe jej sprawdzenie. Zmodyfikuj akcję `Token()` kontrolera `User` dodając wszystkie role do *claims* tworzonego tokena:

```
foreach (var role in await _user.GetRolesAsync(user))  
{  
    claims.Add(new Claim(ClaimTypes.Role, role));  
}
```

(po utworzeniu listy o nazwie `claims`)

Przetestuj, czy faktycznie dodawać „polubienia” mogą użytkownicy którzy uzyskali token. Nie powinni móc dodawać nowych obiektów żądaniem `POST` do <http://localhost:5010/api/fox>.

Z kolei administratorzy (użytkownicy należący do roli administratora) ze swoim tokenem powinni móc robić obydwie czynności.

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

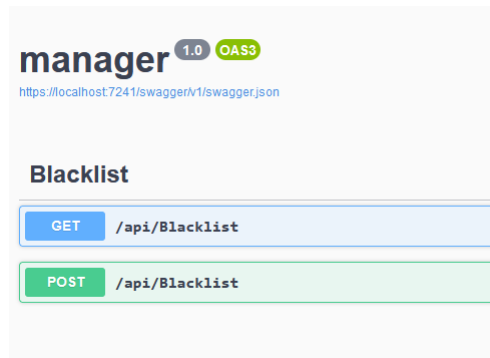
Pobierz z Moodle przykładowy projekt „Lab10”. Składa się on z 3 projektów: „api” to podsystem, który realizuje mechanizm „cenzury” – po wysłaniu tekstu powinien on zwracać ocenzurowany tekst, na podstawie listy słów „zakazanych” – słowa znajdujące się na liście słów zakazanych mają zostać zamienione 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 Poleceń i przejdź do folderu projektu „api”. Następnie spróbuj wydać komendę:

```
dotnet publish
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



W folderze `/bin/Debug/net6.0/publish` znajdują 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 linię:

```
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	sensorapi: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	sensormanager:latest	"dotnet manager.dll"	3 minutes ago	Up 2 minutes	80/tcp, 443/tcp
	lab10-manager-1				

Rys 10.2. Raportowanie stanu zdrowia aplikacji.





Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego