

# Zaawansowane Programowanie Internetowe

## (studia niestacjonarne)

### Laboratorium 1

Zawartość instrukcji:

1. Przygotowanie środowiska.
2. Przygotowanie solucji.
3. Budowa modelu danych - zadania.
4. Budowa warstwy dostępu do danych - zadania.
5. Budowa oraz testowanie autonomicznych usług zawierających logikę biznesową rozwiązania - zadania.

dr inż. Rafał Grycuk

# 1. Przygotowanie środowiska (Environment setup)

W trakcie zajęć laboratoryjnych będziecie Państwo korzystać z poniższego oprogramowania proszę się upewnić że jest ono zainstalowane na Państwa komputerze.

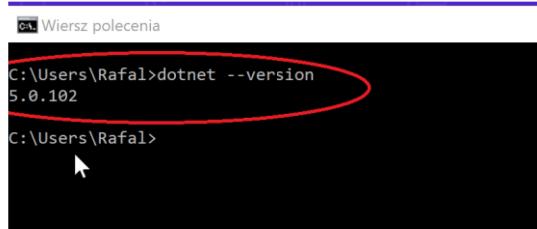
- [.NET SDK \(.NET 6.0 lub wyższy\)](#)
- [SQL Server LocalDB \(Windows\) lub SQL Server Express \(MacOS, Linux\)](#)
- [Visual Studio Code](#)
- [Git](#)

## 1.1. .NET SDK

Aby tworzyć aplikacje we frameworku .NET 5 lub wyższym, należy zainstalować środowisko uruchomieniowe .NET SDK. Aplikacje napisane na tej platformie mogą być uruchamiane zarówno na systemie Windows jak i Linux czy MacOS. Środowisko to wymaga dedykowanego SDK, dostępnego dla różnych platform, które należy [pobrać](#) i zainstalować. Niezwykle istotnym jest aby zainstalować wersję .NET 5.0 dla wybranej platformy. Ważne jest również aby był to pakiet SDK. Po poprawnej instalacji proszę uruchomić konsolę i wpisać:

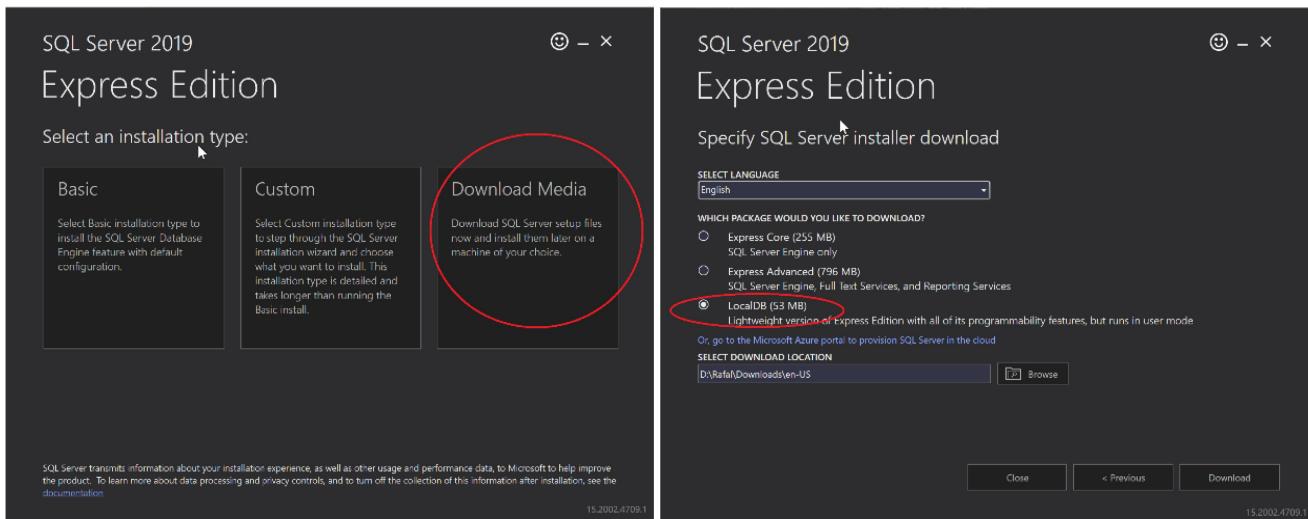
```
dotnet --version
```

Jeśli w odpowiedzi otrzymaliście Państwo wersję .NET SDK oznacza to że pakiet został zainstalowany poprawnie.



## 1.2. SQL Server LocalDB (Windows) lub SQL Server Express (MacOS, Linux)

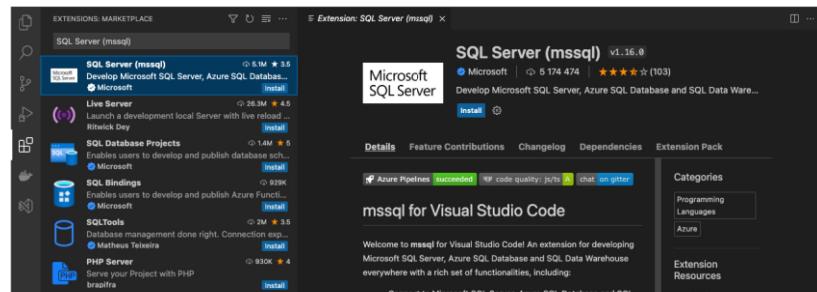
W celu przechowywania danych aplikacji oraz zachowania mechanizmu wielodostępności bazy danych zdecydowano się na SQL Server Express LocalDB. Narzędzie to pozwala na korzystania z silnika bazy danych SQL Server. Zawiera jedynie niezbędne składniki aby korzystać z bazy danych. W systemie Windows możliwe jest zainstalowanie jedynie składnika SQL Server LocalDB. Natomiast w przypadku systemu Linux lub MacOS należy zainstalować całą bazę SQL Server Express. Jeżeli jeszcze nie zostało ono zainstalowane, proszę [pobrać](#) i zainstalować to oprogramowanie. Aby zainstalować **LocalDb** proszę w instalatorze wybrać opcję **Download Media** a następnie opcję **LocalDb**.



W przypadku platform Linux lub MacOS proszę zainstalować wersję SQL Express proszę zrealizować to poprzez docker. Proszę wykorzystać poniższy obraz dockera.

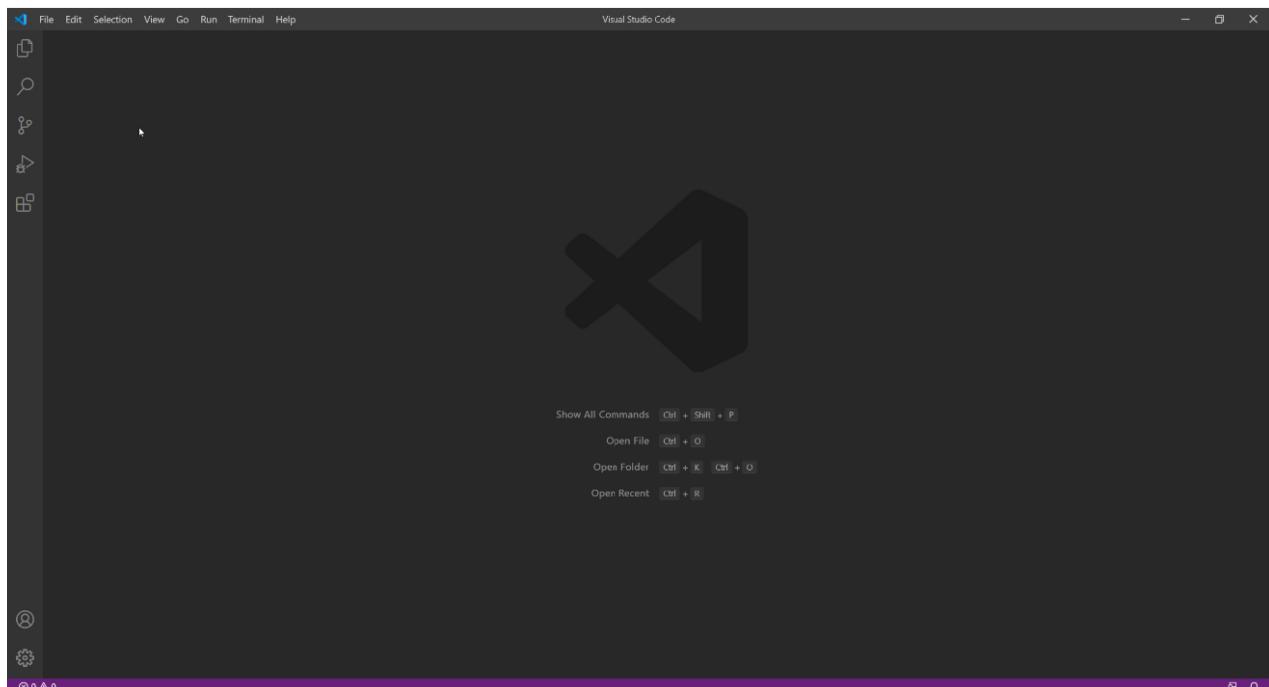
[mcr.microsoft.com/azure-sql-edge](https://mcr.microsoft.com/azure-sql-edge)

W celu połączenia się z bazą danych proszę zainstalować extension **SQL Server (mssql)** w VS Code.



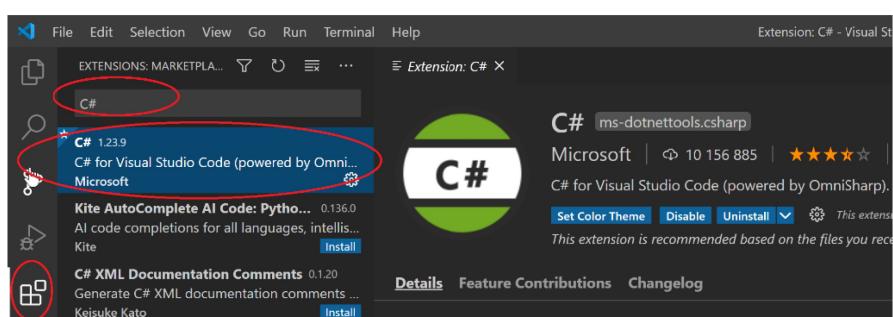
### 1.3. Visual Studio Code

Visual Studio Code to rozszerzony edytor kodu z obsługą operacji programistycznych, takich jak debugowanie, uruchamianie zadań oraz kontrola wersji. Ma na celu zapewnienie tylko niezbędnych narzędzi potrzebnych programistom do szybkiego cyklu kompilowania i debugowania kodu. Jest to rozwiązanie w pełni modułowe i poprzez wykorzystanie dodatków (extensions) możliwe jest rozbudowanie Visual Studio Code do pełnoprawnego IDE. Jednakże w takim przypadku może się okazać że lepszym rozwiązaniem jest zainstalowanie Visual Studio IDE. VS Code to rozwiązane dedykowane dla wielu języków oraz frameworków. W większości z nich samo narzędzie zaproponuje najbardziej odpowiedni dodatek do zainstalowania. Ogromną zaletą VS Code jest wieloplatformowość. Obsługuje zarówno system Windows, jak i Linux czy MacOS. Kolejną zaletą tego narzędzia jest stosunkowo niewielki rozmiar w szczególności w porównaniu do VS IDE. Jeżeli jeszcze nie zostało ono zainstalowane, proszę [pobrać](#) i zainstalować narzędzie VS Code. Po uruchomieniu powinniście Państwo uzyskać poniższy rezultat:



Aby tworzyć aplikacje w przy pomocy .NET oraz C# należy zainstalować dodatek umożliwiający korzystanie z IntelliSense oraz analizy języka. W tym celu proszę:

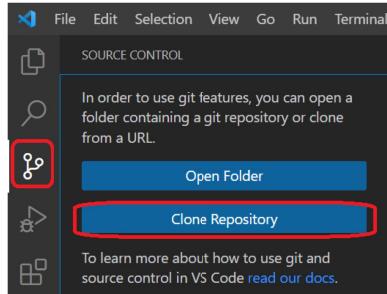
- 1) Przejść do menu Extensions:



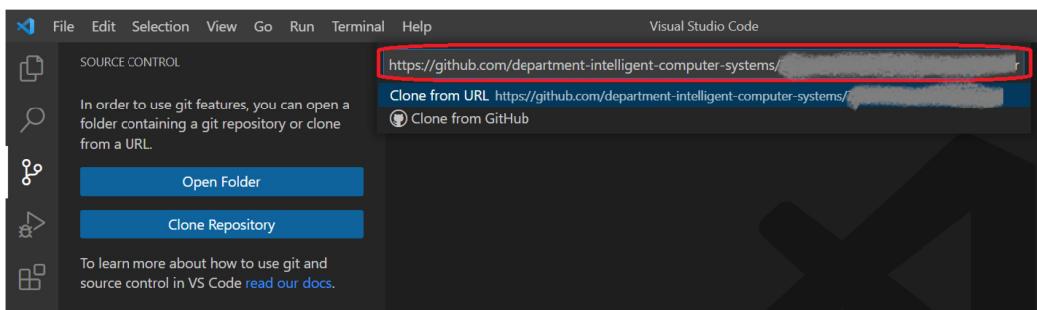
- 2) Następnie proszę wpisać C# w polu wyszukiwania i zainstalować powyższy dodatek.  
W ten sposób środowisko do tworzenia aplikacji webowych zostało przygotowane.

#### 1.4. Git

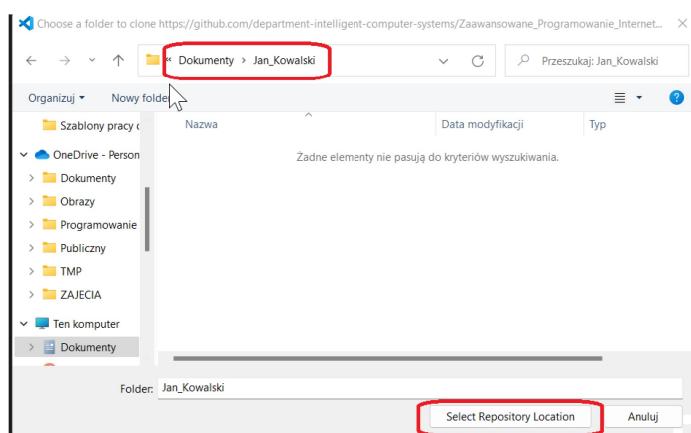
- 1) W kolejnym kroku uruchamiamy **Visual Studio Code** (zakładając ze już jest zainstalowane).
- 2) Następnie klikamy **Source Control**.
- 3) Klikamy **Clone Repository**.



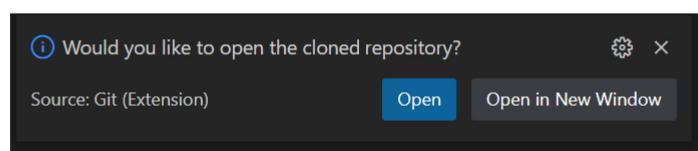
- 4) Następnie wklejamy poniższy do repozytorium i naciskamy Enter. Link ten powinniście Państwo otrzymać od prowadzącego.



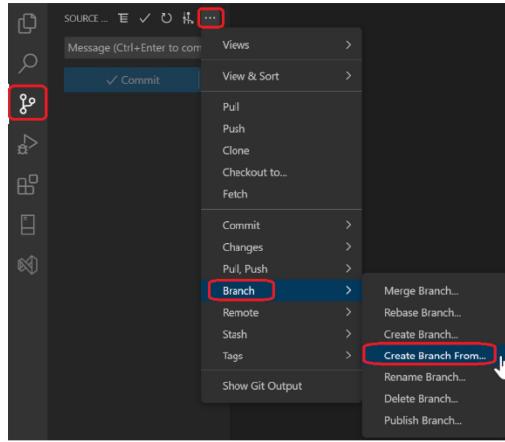
- 5) Następnie proszę w folderze **Dokumenty** stworzyć nowy folder o nazwie zawierającej imię i nazwisko studenta, np. Jan\_Kowalski.
- 6) W kolejnym kroku wskazujemy folder w którym będziemy przechowywać całe repozytorium.



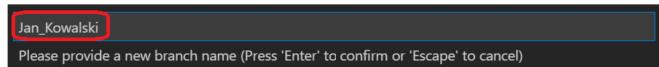
- 7) Na końcu klikamy **Select Repository Location**. W tym momencie klient VS Code pobierze całe repozytorium z GitHub.com oraz zaprośi się czy chcemy je otworzyć, klikamy **Open**.



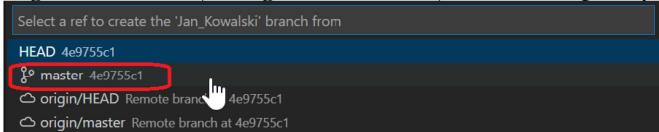
- 8) Następnie należy utworzyć nową gałąź (jedna gałąź = jeden student).
- 9) W VS Code w sekcji **Source Control** klikamy ... => **Branch => Create Branch From**.



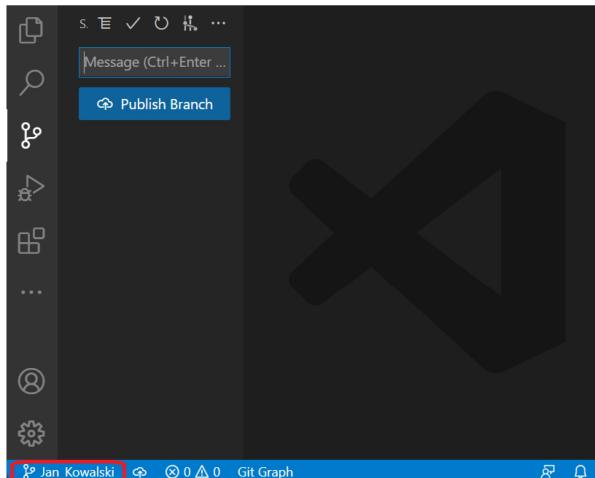
- 10) Następnie proszę wpisać swoje imię i nazwisko jako nazwę gałęzi, według poniższego wzoru: **imię\_nazwisko** oraz nacisnąć Enter.



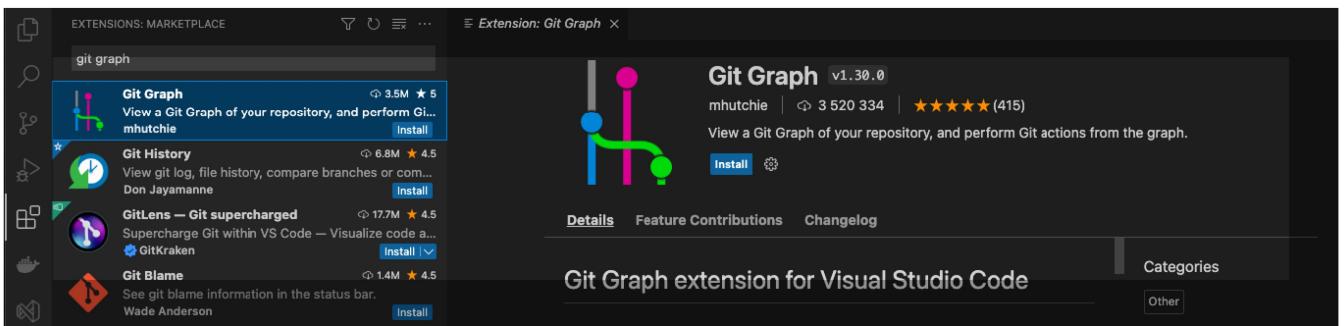
- 11) Kolejno proszę wybrać gałąź **master** jako gałąź od której będzie rozpoczynać się Państwa gałąź.



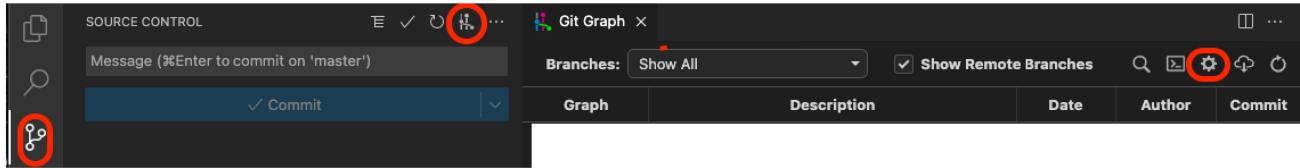
- 12) Podczas pracy nad projektem proszę zawsze sprawdzać, czy znajdujecie się Państwo na swojej gałęzi.



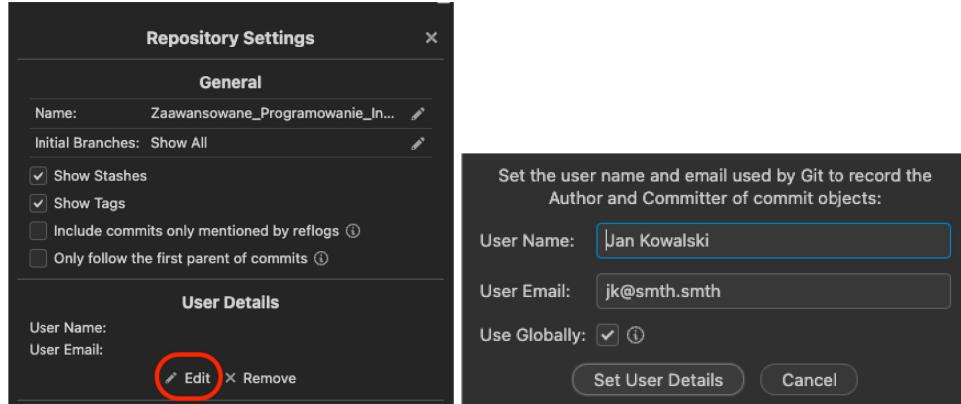
- 13) Proszę zainstalować rozszerzenie **Git Graph**, które umożliwia wizualizację wszystkich gałęzi w repozytorium.



- 14) W kolejnym kroku proszę przejść do ustawień repozytorium poprzez **Source Control => View Git Graph => Repository Settings**.

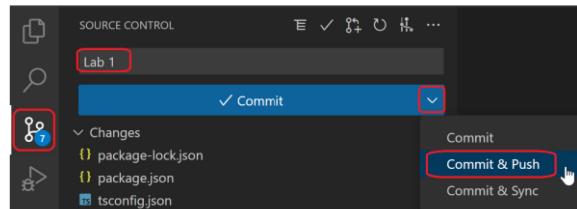


15) Następnie proszę edytować ustawienia **User Name** oraz **Email**. I podać **nieistniejący** email.



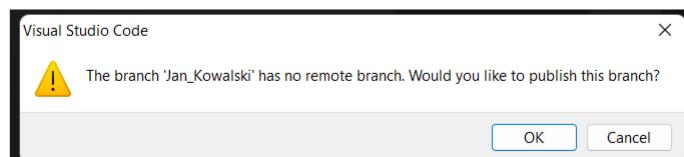
Następnie pracujemy nad kodem.... 😊 Tworzymy nowe pliki i zapisujemy zmiany. Git będzie śledził wszystkie zmiany wykonane w folderze repozytorium.

16) Po zakończonej pracy możemy zauważyc że kilka plików zostało zmienionych. W przykładzie poniżej zmieniono (lub utworzono) 7 plików.



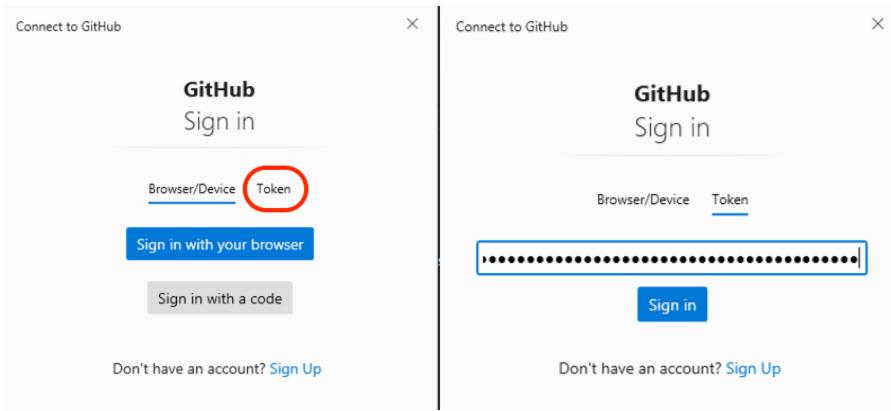
17) Aby zapisać zmiany w gałęzi lokalnej (polecenie **Commit**) oraz zdalnej (polecenie **Push**). Proszę wpisać komentarz (wiadomość) w polu **Message**. Następnie proszę rozwinąć przycisk **Commit** oraz wybrać opcję **Commit & Push**.

18) Następnie pojawi się poniższe okienko. Proszę wybrać ok.



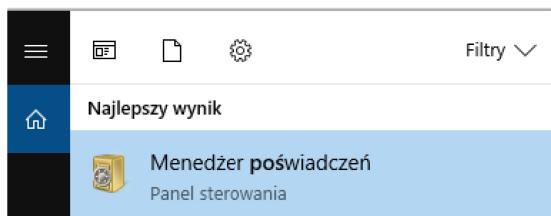
**WAŻNE:** Zalecany jest stosowanie **Commit & Push**, gdyż wszystkie zmiany będą umieszczone oraz dostępne na serwerze GitHub, w odpowiedniej gałęzi studenta.

W niektórych przypadkach może pojawić się poniższe okienko logowania. Występuje ono najczęściej, gdy nie posiadamy zapianych uprawnień do zapisu na serwerze GitHub. W takim przypadku proszę wybrać opcję **token** następnie wkleić podany przez prowadzącego **Personal Access Token**.

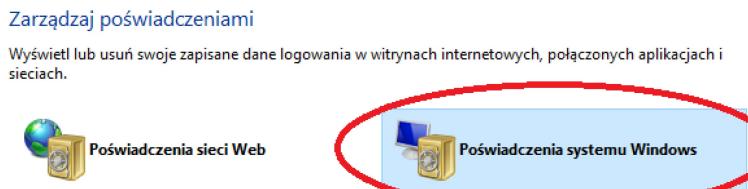


W przypadku, gdyby wystąpił błąd komunikujący brak dostępu do repozytorium oznacza to, że na komputerze istnieją zapisane inne poświadczenia do github.com nieposiadające dostępu do danego repozytorium.

- Należy wtedy w wyszukiwarce menu Start wyszukać **menedżera poświadczeń**.



- Następnie klikamy na:



- W kolejnym kroku usuwamy poświadczenia:



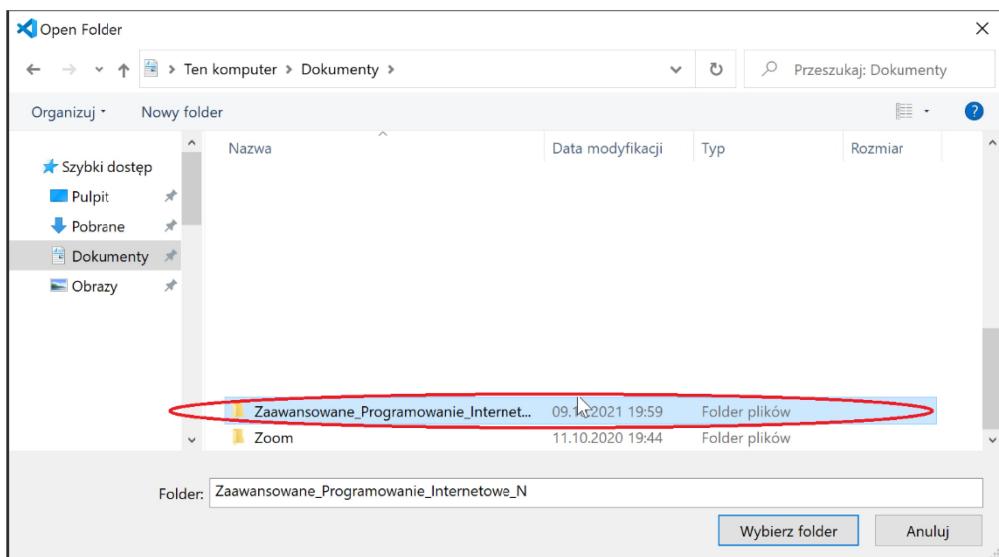
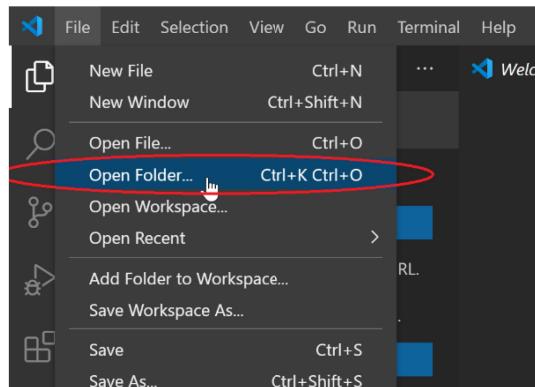
- Następnie próbujemy powtórzyć operację **push**, analogicznie jak w punkcie 17.

## 2. Przygotowanie solucji (Solution setup)

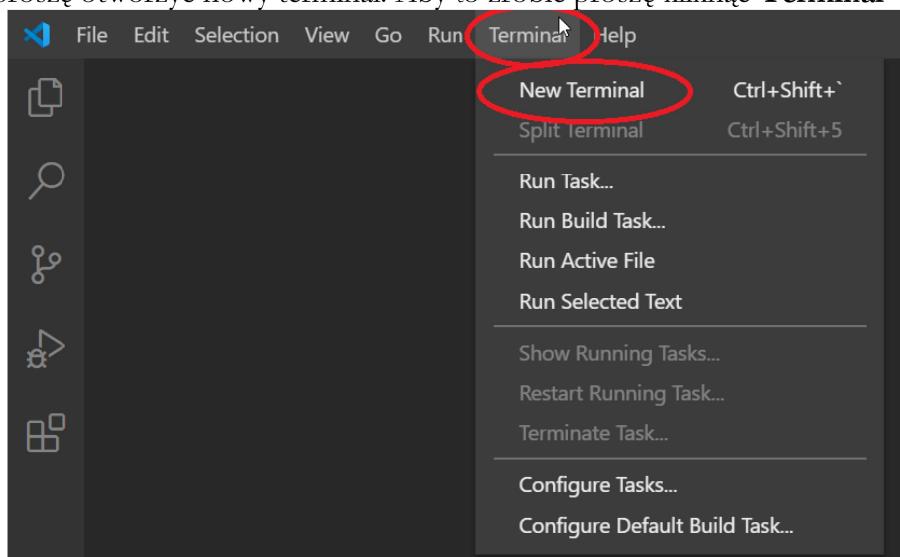
W celu stworzenia solucji z podziałem na poszczególne projekty należy postępować według poniższych kroków:

- W kolejnym kroku uruchamiamy Visual Studio Code.
- Proszę otworzyć poniższy folder (o ile jeszcze nie jest otwarty) **File => Open folder**. Następnie przechodzimy do folderu w którym zostało sklonowane repozytorium. Niezwykle ważne jest aby koniec ścieżki zawierał folder:

...\\Zaawansowane\_Programowanie\_Internetowe\_N



- 3) Następnie proszę otworzyć nowy terminal. Aby to zrobić proszę kliknąć **Terminal => New Terminal**



- 4) Typowa aplikacja wielowarstwowa składa się z poniższych warstw (projektów). Proszę w miejsce **NazwySolucji** wstawić wybraną nazwę solucji, np. **WebStore**.

<b>Nazwa projektu</b>	<b>Typ projektu</b>	<b>Parametr</b>
<b>NazwaSolucji.Model</b>	Class library	classlib
<b>NazwaSolucji.DAL</b>	Class library	classlib
<b>NazwaSolucji.Services</b>	Class library	classlib
<b>NazwaSolucji.ViewModels</b>	Class library	classlib
<b>NazwaSolucji.Web</b>	React App	react

- 5) W związku z tym aby stworzyć pierwszy projekt, proszę w terminalu wpisać następujące polecenie

```
dotnet new classlib -n WebStore.Model -f net6.0
```

Jak można zauważyć parametr `-n` oznacza nazwę projektu natomiast parametr `-f` oznacza wersję .NET.

- 6) Proszę powtórzyć powyższy punkt dla wszystkich projektów typu **Class library**. Czyli wszystkich pozostałych **oprócz WebStore.Web**
- 7) Wspomniany projekt **WebStore.Web** należy stworzyć przy pomocy poniższego polecenia:

```
dotnet new react -n WebStore.Web -f net6.0
```

Po stworzeniu wszystkich projektów powinniście Państwo uzyskać poniższy efekt, gdzie **NazwaSolucji** będzie zastąpiona przez **WebStore**:



- 8) Następnie należy stworzyć plik solucji: aby to zrobić proszę wykonać poniższe polecenie:

```
dotnet new sln -n WebStore
```

- 9) Kolejno należy dodać wszystkie wcześniej stworzone projekty do stworzonej przed chwilą solucji.

```
dotnet sln WebStore.sln add (ls -r **/*.csproj)
```

- 10) W kolejnym kroku należy dodać odpowiednie referencje do projektów. **Można to zrobić na dwa sposoby:**

- Poprzez wykonanie komendy

```
dotnet add {source project} reference {destination project}
```

Przykład dodania referencji do projektu **WebStore.Model** z projektu **WebStore.Web**, dzięki temu klasy stworzone w projekcie **WebStore.Model**, będą widoczne w projekcie **WebStore.Web**.

```
dotnet add WebStore.Web reference WebStore.Model
```

- Drugim sposobem na dodanie referencji jest dodanie wpisu w pliku **\*.csproj**. Przykład dodania referencji do projektu **WebStore.Model** z projektu **WebStore.DAL**. Poniżej zawartość pliku **WebStore.DAL.csproj**

```
<Project Sdk="Microsoft.NET.Sdk">

  <ItemGroup>
    <ProjectReference Include=".\\WebStore.Model\\WebStore.Model.csproj" />
  </ItemGroup>

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

11) Proszę dodać referencje do poszczególnych projektów w następujący sposób:

- Z projektu **WebStore.DAL** do:
  - ❖ **WebStore.Model**
- Z projektu **WebStore.ViewModels** do:
  - ❖ **WebStore.Model**
- Z projektu **WebStore.Services** do:
  - ❖ **WebStore.Model**
  - ❖ **WebStore.DAL**
  - ❖ **WebStore.ViewModels**
- Z projektu **WebStore.Web** do:
  - ❖ **WebStore.Model**
  - ❖ **WebStore.DAL**
  - ❖ **WebStore.ViewModels**
  - ❖ **WebStore.Services**

Poniżej znajdują się zawartości plików **\*.csproj** dla powyższych projektów:

➤ **WebStore.Model**

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Identity.Stores" Version="6.0.9" />
  </ItemGroup>
</Project>
```

➤ **WebStore.DAL**

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <ProjectReference Include=".\\WebStore.Model\\WebStore.Model.csproj" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="6.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.9">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Proxies" Version="6.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.9" />
  </ItemGroup>
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

➤ **WebStore.ViewModels**

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <ProjectReference Include=".\\WebStore.Model\\WebStore.Model.csproj" />
  </ItemGroup>
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

➤ **WebStore.Services**

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <ProjectReference Include=".\\WebStore.ViewModels\\WebStore.ViewModels.csproj" />
  </ItemGroup>
```

```

    <ProjectReference Include="..\WebStore.DAL\WebStore.DAL.csproj" />
    <ProjectReference Include="..\WebStore.Model\WebStore.Model.csproj" />
</ItemGroup>
<PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
</PropertyGroup>
</Project>

```

➤ **WebStore.Web**

```

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <TypeScriptCompileBlocked>true</TypeScriptCompileBlocked>
    <TypeScriptToolsVersion>Latest</TypeScriptToolsVersion>
    <IsPackable>false</IsPackable>
    <SpaRoot>ClientApp\</SpaRoot>
    <DefaultItemExcludes>$ (DefaultItemExcludes) ;$ (SpaRoot) node_modules\**</DefaultItemExcludes>
    <SpaProxyServerUrl>https://localhost:44433</SpaProxyServerUrl>
    <SpaProxyLaunchCommand>npm start</SpaProxyLaunchCommand>
    <ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
<ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.SpaProxy" Version="6.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.9" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.6.3" />
</ItemGroup>
<ItemGroup>
    <!-- Don't publish the SPA source files, but do show them in the project files list -->
    <Content Remove="$(SpaRoot)***" />
    <None Remove="$(SpaRoot)***" />
    <None Include="$(SpaRoot)***" Exclude="$(SpaRoot)node_modules\***" />
</ItemGroup>
<ItemGroup>
    <ProjectReference Include="..\WebStore.Model\WebStore.Model.csproj" />
    <ProjectReference Include="..\WebStore.DAL\WebStore.DAL.csproj" />
    <ProjectReference Include="..\WebStore.ViewModels\WebStore.ViewModels.csproj" />
    <ProjectReference Include="..\WebStore.Services\WebStore.Services.csproj" />
</ItemGroup>
<Target Name="DebugEnsureNodeEnv" BeforeTargets="Build" Condition=" '$(Configuration)' == 'Debug' And !Exists('$(SpaRoot)node_modules') ">
    <!-- Ensure Node.js is installed -->
    <Exec Command="node --version" ContinueOnError="true">
        <Output TaskParameter="ExitCode" PropertyName="ErrorCode" />
    </Exec>
    <Error Condition=" '$(ErrorCode)' != '0' " Text="Node.js is required to build and run this project. To continue, please install Node.js from https://nodejs.org/, and then restart your command prompt or IDE." />
    <Message Importance="high" Text="Restoring dependencies using 'npm'. This may take several minutes..." />
    <Exec WorkingDirectory="$(SpaRoot)" Command="npm install" />
</Target>

<Target Name="PublishRunWebpack" AfterTargets="ComputeFilesToPublish">
    <!-- As part of publishing, ensure the JS resources are freshly built in production mode -->
    <Exec WorkingDirectory="$(SpaRoot)" Command="npm install" />
    <Exec WorkingDirectory="$(SpaRoot)" Command="npm run build" />

    <!-- Include the newly-built files in the publish output -->
    <ItemGroup>
        <DistFiles Include="$(SpaRoot)build\***" />
        <ResolvedFileToPublish Include="@{(DistFiles->'%(FullPath)')}" Exclude="@(ResolvedFileToPublish)">
            <RelativePath>wwwroot\%(RecursiveDir)%(FileName)%(Extension)</RelativePath>
            <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
            <ExcludeFromSingleFile>true</ExcludeFromSingleFile>
        </ResolvedFileToPublish>
    </ItemGroup>
</Target>
</Project>

```

- 12) Proszę skompilować solucję, w tym celu należy w konsoli wykonać poniższe polecenie:

dotnet build

Jeżeli wszystkie kroki zostały wykonane poprawnie komplikacja powinna zakończyć się sukcesem.

Kompilacja powiodła się.

Ostrzeżenia: 0

Liczba błędów: 0

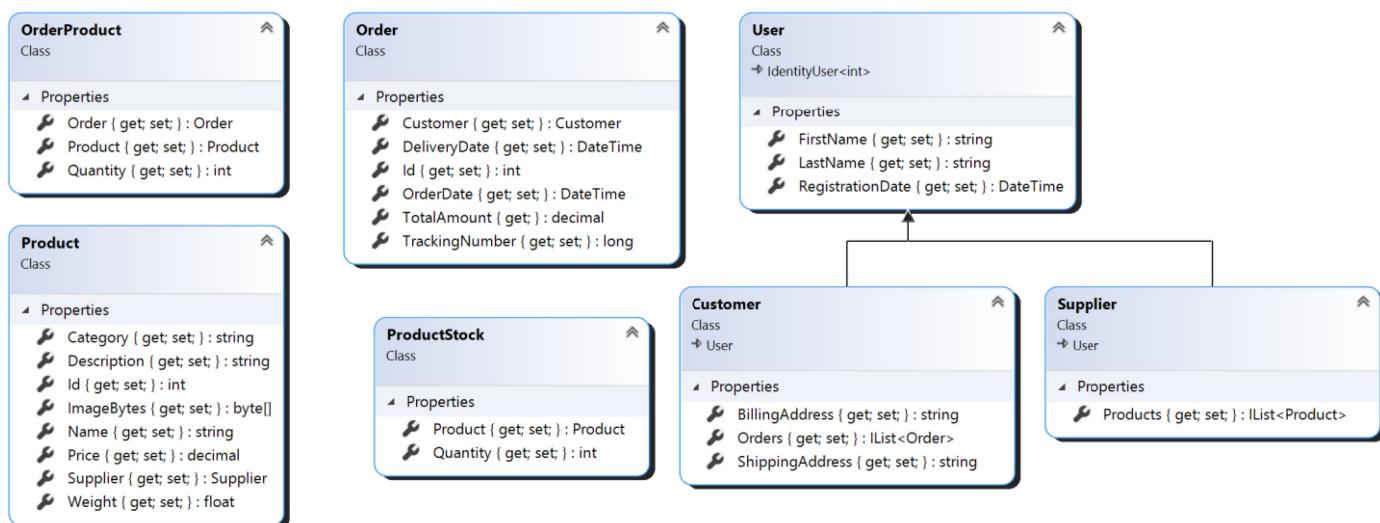
- 13) Aby uruchomić aplikację webową proszę wykonać poniższe polecenie:

```
dotnet run --project WebStore.Web
```

### 3. Budowa modelu danych - zadania

Fundamentem aplikacji webowych jest tzw. data model (lub domain model). Termin ten został on szczegółowo omówiony na kursie poprzedzającym niniejsze zajęcia stąd, proszę wykorzystać wiedzę oraz umiejętności zdobytые na poprzednim kursie w celu wykonania poniższych zdań.

- 1) Proszę stworzyć kompletną solucję o nazwie **WebStore** zawierającą 5 warstw (projektów), analogicznie jak w przypadku pokazanym w punkcie 2.
- 2) W utworzonej solucji proszę zbudować data model przy pomocy poniższego diagramu UML.



- 3) Proszę stworzyć klasę **Address** oraz modyfikować klasę **Customer** w taki sposób aby dane były przechowywane w postaci obiektu zamiast stringa.
- 4) Proszę zmodyfikować uzyskany data model o klasy: **StationaryStore**, **StationaryStoreEmployee**,
- 5) Proszę dodać klasę **Category** oraz odpowiednio zmodyfikować klasę **Product**.
- 6) Proszę dodać klasę **Invoice** oraz dokonać odpowiednich zmian w klasie **Order**.

### 4. Budowa warstwy dostępu do danych - zadania

Analogicznie jak w poprzednim punkcie, warstwa dostępu do danych oraz cały **Entity Framework Core** został omówiony na poprzednim kursie. W oparciu o uzyskaną tam wiedzę i umiejętności, proszę wykonać poniższe zadania. Proszę wykonać poniższe zadania, a następnie proszę przejść do etapu stworzenia oraz zastosowania migracji. Do wykonania poniższych zadań proszę wykorzystać kod stworzony w punkcie poprzednim.

- 1) Proszę zdefiniować relacje **one to many (1:M)** dla poniższych encji:

- **Customer – Order**
- **Supplier – Product**

- Product – ProductStock
  - Customer – Address
  - Category – Product
  - Invoice – Order
  - StationaryStore – Address
  - StationaryStore – StationaryStoreEmployee
- 2) Proszę zrealizować relację **many to many (N:M)** dla encji **Order – Product** wykorzystując encję asocjacyjną **OrderProduct**.

Proszę zrealizować relację dziedziczenia na EF przy pomocy metody **TPH** dla encji **User, Customer, Supplier, StationaryStoreEmployee**.

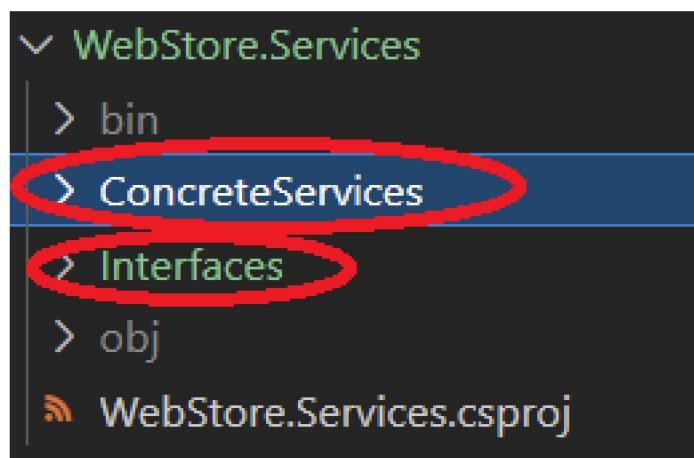
**Baza danych utworzona w ten sposób będzie podstawą pod dalsze elementy projektu. Proszę zwrócić szczególną uwagę na utworzoną migrację, poprawność relacji pomiędzy encjami, gdyż jest to jeden z głównych elementów, które mogą sprawić Państwu problemy.**

## 5. Budowa oraz testowanie autonomicznych usług zawierających logikę biznesową rozwiązań

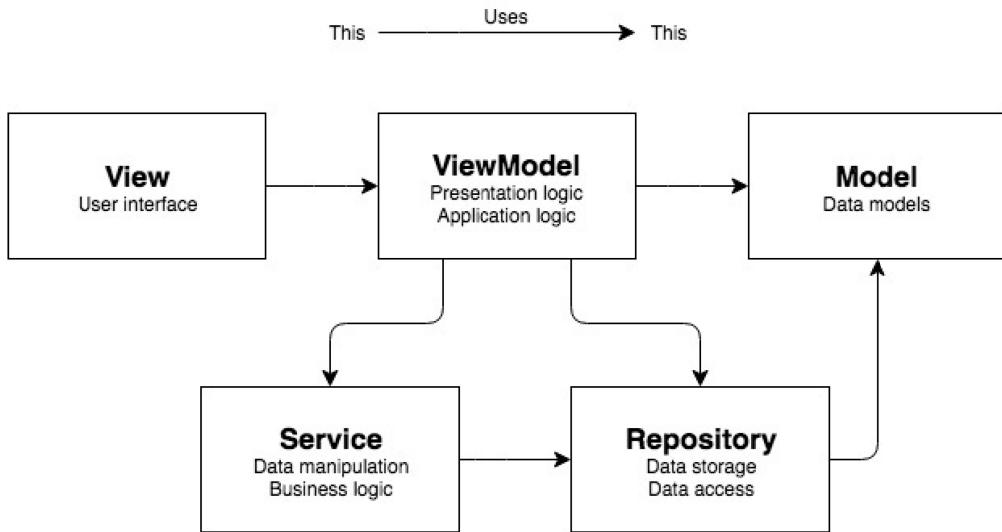
### 5.1. Budowanie autonomicznych usług

Aby przystąpić do tworzenia usług, należy wykonać poniższe kroki:

- 1) W projekcie **WebStore.Services** powinny się znajdować dwa foldery **Interfaces** oraz **ConcreteServices**. Jeżeli nie istnieją, należy je utworzyć.



- 2) Aby poprawnie zadeklarować implementację metod serwisów potrzebujemy wprowadzić pojęcie **view modelu**. View modele są to klasy odpowiadające za agregację, projekcję oraz dostarczanie danych między modelem a widokiem. Najczęściej klasy te „zbierają” dane z wielu modeli do jednej klasy lub redukują dane otrzymane z modelu. Takie podejście jest często stosowane we modelu programowania MVVM. Równie często view modele stosowane są do ukrywania niechcianych właściwości modelu w widoku. Takie podejście daje wiele korzyści, szczególnie na etapie testowania (testy jednostkowe). View modele najczęściej stosowane są przy pobieraniu i wyświetlaniu danych.



- 3) W projekcie **WebStore.ViewModels** mamy obecnie jeden folder **VM** (jeśli go jeszcze nie ma proszę go utworzyć). Wszystkie klasy (i pliki) view modeli będziemy umieszczać w folderze VM. Każda klasa znajdująca się w tym folderze powinna zawierać suffix **Vm**, np. **ProductVm**, **AddOrUpdateProductVm**.

```

using System.ComponentModel.DataAnnotations;
namespace WebStore.ViewModels
{
    public class AddOrUpdateProductVm
    {
        public int? Id { get; set; }

        [Required]
        public string Description { get; set; } = default!;

        [Required]
        public byte[] ImageBytes { get; set; } = default!;

        [Required]
        public string Name { get; set; } = default!;

        [Required]
        public decimal Price { get; set; }

        [Required]
        public float Weight { get; set; }

        [Required]
        public int CategoryId { get; set; }

        [Required]
        public int SupplierId { get; set; }
    }
}

namespace WebStore.ViewModels
{
    public class ProductVm
    {
        public string Description { get; set; } = default!;
        public byte[] ImageBytes { get; set; } = default!;
        public string Name { get; set; } = default!;
        public decimal Price { get; set; }
        public float Weight { get; set; }
        public int Quantity { get; set; }
    }
}

```

W klasie **AddOrUpdateProductVm** annotacja **[Required]** pozwoli na walidację view modelu przy pomocy **ModelState** na poziomie kontrolera. Oczywiście możemy dodawać inne validatory oraz tworzyć własne. Więcej informacji [tutaj](#).

- 4) W folderze **Interfaces** tworzymy nowy plik \*.cs, który będzie interfejsem o nazwie **IProductService.cs**. Należy również zachować odpowiednią konwencję nazewnictwą. Wszystkie interfejsy powinny się rozpoczynać od wielkiej litery „I”. Natomiast interfejsy będące abstrakcją usług powinny się kończyć sufiksem ”Service”. Utworzony interfejs powinien być publiczny oraz zawierać deklaracje metod, które będą później zaimplementowane przez klasę implementującą interfejs.

```
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using WebStore.Model.DataModels;
using WebStore.ViewModels;

namespace WebStore.Services.Interfaces {
    public interface IProductService {
        ProductVm AddOrUpdateProduct (AddOrUpdateProductVm addOrUpdateProductVm);
        ProductVm GetProduct (Expression<Func<Product, bool>> filterExpression);
        IEnumerable<ProductVm> GetProducts (Expression<Func<Product, bool>> ? filterExpression = null);
    }
}
```

- 5) Aby móc korzystać z **ApplicationDbContext**, oraz jednocześnie nie definiować ciągle zmiennych tego typu, należy stworzyć klasę bazową serwisu **BaseService**, po której będą dziedziczyć wszystkie serwisy. Klasę tę należy stworzyć w folderze **ConcreteServices** w projekcie **WebStore.Services**. Dzięki temu wszystkie klasy dziedziczące po klasie bazowej będą mogły korzystać ze składowych klasy bazowej, np. **ApplicationContext**.
- 6) Aby klasa **BaseService** działała poprawnie należy zainstalować dodatkowo z NuGet-a poniższe pakiety dla projektu **WebStore.Services**. O ile nie zostały wcześniej zainstalowane.

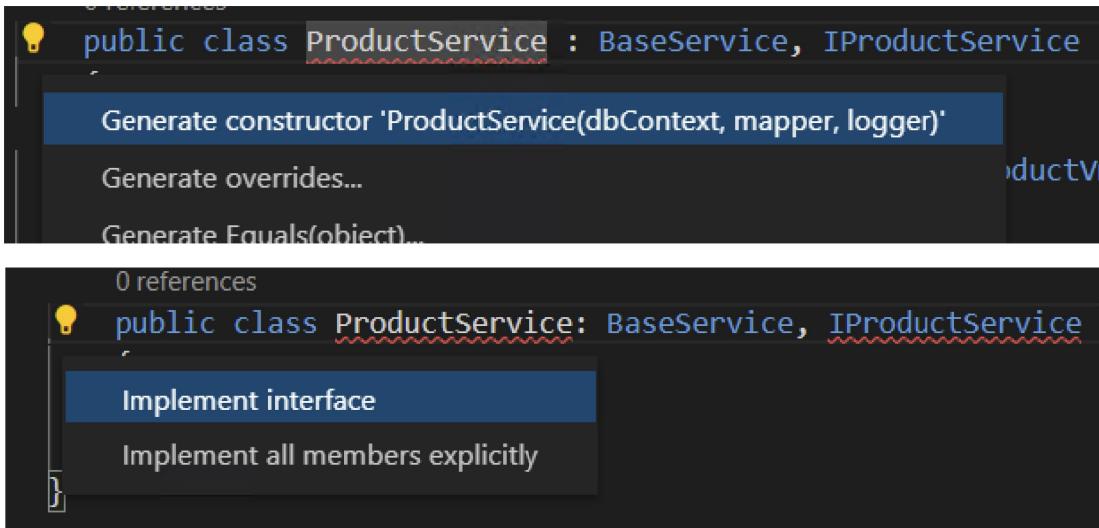
```
dotnet add WebStore.Services package Microsoft.Extensions.Logging.Abstractions --version 6.0.2
dotnet add WebStore.Services package AutoMapper --version 12.0.0
dotnet add WebStore.Services package AutoMapper.Extensions.Microsoft.DependencyInjection --version 12.0.0
```

- 7) Proszę wkleić poniższy kod do pliku **BaseService.cs**

```
using AutoMapper;
using Microsoft.Extensions.Logging;
using WebStore.DAL.EF;

namespace WebStore.Services.ConcreteServices {
    public abstract class BaseService {
        protected readonly ApplicationDbContext DbContext;
        protected readonly ILogger Logger;
        protected readonly IMapper Mapper;
        public BaseService (ApplicationDbContext dbContext,
                           IMapper mapper, ILogger logger) {
            DbContext = dbContext;
            Logger = logger;
            Mapper = mapper;
        }
    }
}
```

- 8) Następnie proszę stworzyć nowy plik o nazwie **ProductService.cs** (**w folderze ConcreteServices**) zawierający klasę o tej samej nazwie a następnie zaimplementować interfejs **IProductService** w postaci klasy. Po jej stworzeniu możemy stworzyć konstruktor oraz zaimplementować metody. W tym celu proszę kliknąć żółtą ikonę żarówki.



Stworzony konstruktor klasy umożliwia wstrzyknięcie obiektu typu **ApplicationDbContext**, **IMapper**, oraz **ILogger**. Następnie wywoływany jest konstruktor klasy bazowej (**BaseService**) do którego przekazujemy argumenty parametrów.

Dzięki zastosowaniu interfejsu możliwe jest zaimplementowanie klasy serwisu. Jak widać dziedziczymy po bazowym serwisie oraz implementujemy interfejs.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using AutoMapper;
using Microsoft.Extensions.Logging;
using WebStore.DAL.EF;
using WebStore.Model.DataModels;
using WebStore.Services.Interfaces;
using WebStore.ViewModels;

namespace WebStore.Services.ConcreteServices {
    public class ProductService : BaseService, IProductService {
        public ProductService (ApplicationDbContext dbContext, IMapper mapper, ILogger logger)
            : base (dbContext, mapper, logger) { }

        public ProductVm AddOrUpdateProduct (AddOrUpdateProductVm addOrUpdateProductVm) {
            try {
                if (addOrUpdateProductVm == null)
                    throw new ArgumentNullException ("View model parameter is null");
                var productEntity = Mapper.Map<Product> (addOrUpdateProductVm);
                if (addOrUpdateProductVm.Id.HasValue || addOrUpdateProductVm.Id == 0)
                    DbContext.Products.Update (productEntity);
                else
                    DbContext.Products.Add (productEntity);
                DbContext.SaveChanges ();
                var productVm = Mapper.Map<ProductVm> (productEntity);
                return productVm;
            } catch (Exception ex) {
                Logger.LogError (ex, ex.Message);
                throw;
            }
        }

        public ProductVm GetProduct (Expression<Func<Product, bool>> filterExpression) {
            try {
                if (filterExpression == null)
                    throw new ArgumentNullException ("Filter expression parameter is null");
                var productEntity = DbContext.Products.FirstOrDefault (filterExpression);
                var productVm = Mapper.Map<ProductVm> (productEntity);
                return productVm;
            } catch (Exception ex) {
                Logger.LogError (ex, ex.Message);
                throw;
            }
        }

        public IEnumerable<ProductVm> GetProducts (Expression<Func<Product, bool>> ? filterExpression = null)
    {
        try {

```

```
        var productsQuery = DbContext.Products.AsQueryable ();
        if (filterExpression != null)
            productsQuery = productsQuery.Where (filterExpression);
        var productVms = Mapper.Map<IEnumerable<ProductVm>> (productsQuery);
        return productVms;
    } catch (Exception ex) {
        Logger.LogError (ex, ex.Message);
        throw;
    }
}
}
```

Jak można zauważyć w metodzie **AddOrUpdateProduct** najpierw sprawdzamy czy argument parametru **addOrUpdateProductVm** jest równy null, zgłoszany jest wyjątek. Następnie mapujemy przy pomocy **AutoMappera** obiekt view modelu na encję typu **Product**. Kolejno w zależności czy właściwość **Id** jest pusta lub równa 0 encja jest dodawana (Add) w przeciwnym wypadku modyfikowana (Update). Jednakże na tym etapie dane zmieniane są tylko na poziomie aplikacji (oczywiście w pamięci serwera aplikacji). Dopiero metoda **SaveChanges** pozwala na zapisanie tych zmian na bazie danych przy pomocy EF.

W metodzie **GetProduct** przekazujemy wyrażenie lambda jako parametr. Dzięki temu będzie możliwe filtrowanie po dowolnej właściwości obiektu typu **Product**. Jeśli argument tego parametru jest równy null, zgłoszany jest wyjątek. Następnie pobieramy encję zgodną z predykatem podanym jako parametr. Następnie uzyskana encja **product** jest mapowana na view model – **ProductVm** a następnie zwracana.

Metoda **GetProducts** jest łudząco podobna do **GetProduct**, lecz jej zdaniem jest zwrócenie kolekcji obiektów. Stąd filtrowanie odbywa się przy pomocy **Where**. Jeśli predykat podany jako argument parametru jest pusty, zwracane są wszystkie encje (spełniające predykat) znajdujące się w bazie danych. Jeśli argument istnieje to jest przekazywany do metody **Where**, która zwraca wszystkie encje zgodne w predykatem podanym jaki wyrażenie lambda.

## 5.2. Testowanie oprogramowania

Testowanie oprogramowania jest oraz powinno być nieodłącznym elementem w trakcie procesu tworzenia aplikacji. Niestety w wielu przypadkach ten proces jest całkowicie pomijany, co często prowadzi do poważnych luk w oprogramowaniu. Często również w celu oszczędności w procesie developmentu stosuje się testowanie jedynie najbardziej witalnych modułów systemu. Jednakże profesjonalne systemy oraz projekty opierają się na wielu modelach testowania oprogramowania. Docelowo dążymy do pokrycia testami jak największej ilości kodu. Można wyróżnić następujące poziomy testów:

- Testy jednostkowe – pozwalają na testowanie pojedynczych elementów systemu tj. metody, obiekty.
  - Testy integracyjne – testy tego typu pozwalają na wykrycie luk w poszczególnych modułach lub elementach systemu jak również, umożliwiają wykrycie problemów pomiędzy poszczególnymi wersjami oprogramowania.
  - Testy GUI – testy tego typu pozwalają na sprawdzenie poprawności działania interfejsu użytkownika. Przykładem narzędzia do testów GUI, jest Selenium.

Obecnie, praktycznie każdy framework webowy korzysta z framework'a do testów. Nie inaczej jest w przypadku .NET. Dostępne są frameworki testowe:

- MSTest – framework do testów dostarczany przez Microsoft.
  - NUnit – jest framework zaczerpniętym (porting) z JUnit, dedykowanego do Javy.
  - xUnit – framework napisany przez twórcę NUnita, dedykowany do rozwiązań .NET, wspierający różne narzędzia, np. ReSharper.

Podczas laboratorium będziemy wykorzystywać xUnita. Jednakże możliwe jest wykorzystywanie pozostałych.

Poprzednio tworzone były usługi, których celem jest realizacja poszczególnych funkcjonalności w oparciu o dostarczone dane poprzez view modele. Jednakże do tej pory nie było możliwe sprawdzenie czy dana metoda/klasa działa poprawnie i zgodnie z założonymi wymaganiami. Po zastosowaniu testów jednostkowych będzie to możliwe. W celu konfiguracji projektu do testów jednostkowych proszę wykonać następujące kroki:

- 1) Proszę stworzyć nowy projekt o nazwie **WebStore.Tests**:

```
dotnet new xunit -n WebStore.Tests -f net5.0
```

- 2) Następnie proszę dodać projekt do solucji (poniższy kod działa tylko pod Windows, w przypadku innych platform proszę kolejno dodać wszystkie projekty do solucji):

```
dotnet sln WebStore.sln add (ls -r **/*.csproj)
```

- 3) W kolejnym kroku proszę dodać poniższe referencje:

```
dotnet add WebStore.Tests reference WebStore.Services
```

```
dotnet add WebStore.Tests reference WebStore.Model
```

```
dotnet add WebStore.Tests reference WebStore.DAL
```

- 4) Kolejno proszę zainstalować następujące biblioteki:

```
dotnet add WebStore.Tests package Xunit.DependencyInjection --version 8.5.0
```

```
dotnet add WebStore.Tests package Microsoft.EntityFrameworkCore.InMemory --version 6.0.9
```

```
dotnet add WebStore.Tests package Microsoft.AspNetCore.Identity.UI --version 6.0.9
```

- 5) Proszę w projekcie **WebStore.Tests** stworzyć plik **Extensions.cs** a następnie wkleić poniższą zawartość:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.DependencyInjection;
using WebStore.DAL.EF;
using WebStore.Model.DataModels;
namespace WebStore.Tests {

    public static class Extensions {
        // Create sample data
        public static async void SeedData (this IServiceCollection services) {
            var serviceProvider = services.BuildServiceProvider ();
            var dbContext = serviceProvider.GetRequiredService<ApplicationDbContext> ();
            var userManager = serviceProvider.GetRequiredService<UserManager<User>> ();
            var roleManager = serviceProvider
                .GetRequiredService<RoleManager<IdentityRole<int>>> ();

            // other seed data ...

            //Suppliers
            var supplier1 = new Supplier () {
                Id = 1,
                FirstName = "Adam",
                LastName = "Bednarski",
                UserName = "suppl1@eg.eg",
                Email = "suppl1@eg.eg",
                RegistrationDate = new DateTime (2010, 1, 1),
            };
            await userManager.CreateAsync (supplier1, "User1234");

            //Categories
            var category1 = new Category () {
                Id = 1,
                Name = "Computers",
                Tag = "#computer"
            };
        }
    }
}
```

```
    await dbContext.AddAsync (category1);

    //Products
    var p1 = new Product () {
        Id = 1,
        CategoryId = category1.Id,
        SupplierId = supplier1.Id,
        Description = "Bardzo praktyczny monitor 32 cale",
        ImageBytes = new byte[] { 0xff, 0xff, 0xff, 0x80 },
        Name = "Monitor Dell 32",
        Price = 1000,
        Weight = 20,
    };
    await dbContext.AddAsync (p1);

    var p2 = new Product () {
        Id = 2,
        CategoryId = category1.Id,
        SupplierId = supplier1.Id,
        Description = "Precyzyjna mysz do pracy",
        ImageBytes = new byte[] { 0xff, 0xff, 0xff, 0x70 },
        Name = "Mysz Logitech",
        Price = 500,
        Weight = 0.5f,
    };
    await dbContext.AddAsync (p2);

    // save changes
    await dbContext.SaveChangesAsync ();
}

}
```

- 6) W ostatnim kroku proszę stworzyć plik **Startup.cs** a następnie do niego wkleić poniższy kod:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using WebStore.DAL.EF;
using WebStore.Model.DataModels;
using WebStore.Services.ConcreteServices;
using WebStore.Services.Configuration.Profiles;
using WebStore.Services.Interfaces;

namespace WebStore.Tests {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddAutoMapper (typeof (MainProfile));
            services.AddEntityFrameworkInMemoryDatabase ()
                .AddDbContext<ApplicationDbContext> (options =>
                    options.UseInMemoryDatabase ("InMemoryDb")
                );

            services.AddIdentity<User, IdentityRole<int>> (options => {
                options.SignIn.RequireConfirmedAccount = false;
                options.Password.RequiredLength = 6;
                options.Password.RequiredUniqueChars = 0;
                options.Password.RequireNonAlphanumeric = false;
            })
                .AddRoleManager<RoleManager<IdentityRole<int>>> ()
                .AddUserManager<UserManager<User>> ()
                .AddEntityFrameworkStores<ApplicationDbContext> ();
            services.AddTransient (typeof (ILogger), typeof (Logger<Startup>));
            // service binding
            services.AddTransient<IPrductService, ProductService> ();
            // ... other bindings...
            services.SeedData ();
        }
    }
}
```

### 5.3. Tworzenie testów jednostkowych (Unit tests)

Aby stworzyć testy jednostkowe posłużymy się klasą bazową, dzięki której zaoszczędzimy ciągłego tworzenia obiektów typu **ApplicationDbContext**. Aby stworzyć testy jednostkowe proszę wykonać następujące kroki:

- 1) Proszę w projekcie **WebStoreTest** stworzyć nowy folder o nazwie **UnitTests**, a następnie w tym folderze stworzyć nowy plik o nazwie **BaseUnitTests.cs**, a następnie proszę wkleić poniższy kod:

```
using WebStore.DAL.EF;
namespace WebStore.Tests.UnitTests {
    public abstract class BaseUnitTests {
        protected readonly ApplicationDbContext DbContext;

        public BaseUnitTests (ApplicationDbContext dbContext) {
            DbContext = dbContext;
        }
    }
}
```

- 2) W kolejnym kroku proszę stworzyć nowy plik o nazwie **ProductServiceUnitTests.cs**. Następnie proszę wkleić poniższy kod. **W przypadku gdy nie posiadacie Państwo danej metody w swoich serwisach proszę odpowiednio zmodyfikować poniższy kod.**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using WebStore.DAL.EF;
using WebStore.Model.DataModels;
using WebStore.Services.Interfaces;
using WebStore.ViewModels.VM;
using Xunit;

namespace WebStore.Tests.UnitTests {
    public class ProductServiceUnitTests : BaseUnitTests {
        private readonly IProductService _productService;
        public ProductServiceUnitTests (ApplicationDbContext dbContext,
            IProductService productService) : base (dbContext) {
            _productService = productService;
        }

        [Fact]
        public void GetProductTest () {
            var product = _productService.GetProduct (p => p.Name == "Monitor Dell 32");
            Assert.NotNull (product);
        }

        [Fact]
        public void GetMultipleProductsTest () {
            var products = _productService.GetProducts (p => p.Id >= 1 && p.Id <= 2);
            Assert.NotNull (products);
            Assert.NotEmpty (products);
            Assert.Equal (2, products.Count ());
        }

        [Fact]
        public void GetAllProductsTest () {
            var products = _productService.GetProducts ();
            Assert.NotNull (products);
            Assert.NotEmpty (products);
            Assert.Equal (products.Count (), products.Count ());
        }

        [Fact]
        public void AddNewProductTest () {
            var newProductVm = new AddOrUpdateProductVm () {
                Name = "MacBook Pro",
                CategoryId = 1,
                SupplierId = 1,
                ImageBytes = new byte[] {
                    0xff,
                    0xff,
                    0xff,
                    0x80
                },
                Price = 6000,
                Weight = 1.1f,
                Description = "MacBook Pro z procesorem M1 8GB RAM, Dysk 256 GB"
            };
            var createdProduct = _productService.AddOrUpdateProduct (newProductVm);
            Assert.NotNull (createdProduct);
        }
    }
}
```

```

        Assert.Equal ("MacBook Pro", createdProduct.Name);
    }

    [Fact]
    public void UpdateProductTest () {
        var updateProductVm = new AddOrUpdateProductVm () {
            Id = 1,
            Description = "Bardzo praktyczny monitor 32 cale",
            ImageBytes = new byte[] { 0xff, 0xff, 0xff, 0x80 },
            Name = "Monitor Dell 32",
            Price = 2000,
            Weight = 20,
            CategoryId = 1,
            SupplierId = 1
        };
        var editedProductVm = _productService.AddOrUpdateProduct (updateProductVm);
        Assert.NotNull (editedProductVm);
        Assert.Equal ("Monitor Dell 32", editedProductVm.Name);
        Assert.Equal (2000, editedProductVm.Price);
    }
}

```

- 3) W celu uruchomienia wszystkich testów jednostkowych w całym projekcie proszę w terminalu wykonać poniższe polecenie.

**dotnet test**

```
Microsoft (R) Test Execution Command Line Tool Version 16.10.0
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: 5, Skipped: 0, Total: 5, Duration: 29 ms - WebStore.Tests.dll (net5.0)
```

Aby wykonać pojedynczy test należy wykorzystać parametr `-filter`

**dotnet test --filter {nazwa\_metody}**

Przykład:

**dotnet test --filter UpdateProductTest**

Aby wykonać wszystkie testy dla danej klasy testowej, należy podać nazwę klasy, np.

**dotnet test --filter ProductServiceUnitTests**

## 5.4. Zadania

Po zapoznaniu się z powyższą instrukcją, proszę zaprojektować oraz zaimplementować następujące usługi (services) oraz odpowiadające im testy (Unit Tests):

- OrderService
- InvoiceService
- StoreService
- AddressService

Każda usługa powinna składać się z dwóch elementów: interfejs oraz klasa go implementująca. Proszę pamiętać aby dziedziczyć również po klasie `BaseService`. Proszę również stworzyć odpowiednie klasy view modelu, analogicznie jak w przypadku wcześniej utworzonej usługi `ProductService`. Proszę przemyśleć funkcjonalność poszczególnych usług, być może możliwe jest wykorzystanie stworzonych wcześniej usług.

Po zakończonej pracy proszę użyć funkcji Commit & Push w Visual Studio Code aby zapisać zmiany w swojej gałęzi na serwerze.