Poniższy dokument jest suplementem do kursu video *C# .NET: Pytania z rozmów kwalifikacyjnych (junior/regular)*.

Znajdziesz tutaj listę pytań z kursu oraz strzeszczenie odpowiedzi na pytania.

Jeżeli masz jakieś pytania lub wątpliwości, to najlepszą formą kontaktu będzie dołączenie do grupy na discordzie poprzez link: https://discord.gg/UDHXQxhM4r lub kontakt przez wiadomość Udemy.

Zainteresowanych moimi pozostałymi kursami, zapraszam na stronę https://linktr.ee/fullstack_developer, gdzie znajdują się aktualne kupony rabatowe Udemy.

Pytania rekrutacyjne

Programowanie

Jakie są paradygmaty programowania obiektowego?

Paradygmaty to podstawowe założenia programowania obiektowego, wyróżniamy 4 z nich:

1. Hermetyzacja

Hermetyzacja polega na ukryciu szczegółów implementacji. Obiekty przedstawiają publiczny interfejs, który mogą wywoływać inne obiekty. Obiekt może zmieniać tylko swój stan. Nikt inny nie może tego robić.

2. Dziedziczenie

Mechanizm dający możliwość tworzenia wyspecjalizowanych obiektów na podstawie obiektów ogólniejszych. Pozwala to współdzielić funkcjonalności. Obiekt wyspecjalizowany może wykorzystywać część definicji obiektu ogólniejszego i rozszerzać go lub zmieniać.

3. Polimorfizm

Mechanizm umożliwiający traktowanie różnych obiektów w jeden sposób dziedziczących tę samą klasę.

4. Abstrakcja

Obiekty są modelami abstrakcyjnych bytów, które wykonują powierzone im zadania bez zdradzania, jak te działania są realizowane. Przedstawiają one uproszczenie danego zagadnienia.

Czym jest SOLID?

SOLID, to zbiór pięciu podstawowych założeń programowania obiektowego. Określają one dobre praktyki tworzenia zarówno klas, jak i tworzenia zależności między nimi. Te pięc zasad to:

1. Single Responsibility Principle
2. Open-close Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

Czym jest memory leak?

Memory leak, to stan naszego programu, w którym program z jakiegoś powodu alokuje coraz więcej pamięci bez jej zwalniania.

Taka sytuacja może doprowadzić do zcrashowania programu lub do zużycia większej ilości pamięci niż dany program miał dostępnej.

Memory leak może wystapić kiedy w odpowiedni sposób nie “czyścimy” obiektów, które korzystają z zasobów zewnętrznych: np. połączenie do bazy danych, czytanie plików z dysku.

Czym jest reguła DRY?

DRY (z ang. Don't Repeat Yourself) – reguła stosowana podczas wytwarzania oprogramowania, zalecająca unikanie różnego rodzaju powtórzeń wykonywanych przez programistów.

Przykładowo unikanie tych samych czynności podczas kompilowania, unikanie wklejania lub pisania tych samych (lub bardzo podobnych) fragmentów kodu w wielu miejscach. Reguła sugeruje też zastępowanie powtarzających się czynności przez jakiś program lub skrypt jeżeli daną czynność da się zautomatyzować.

W kontekście kodu źródłowego największe zastosowanie będą miały reużywalne metody oraz typy generyczne.

Czym jest reguła KISS?

Reguła KISS: (ang. Keep it simple, stupid) – dosłownie: “zrób to prosto, idioto”.

Mówi o na o tym, żeby niepotrzebnie nie komplikować rozwiązania.

Im prostrze będzie działanie programu, tym lepiej.

Czym jest reguła YAGNI?

YAGNI (z ang. You aren't gonna need it) – zasada w programowaniu, kładąca nacisk na zwrócenie uwagi na tworzenie kodu zanim będzie potrzebny.

Często zdarza się, że tworzony jest kod na wszelki wypadek, nawet gdy nie jest potrzebny w danym momencie.

Może się później okazać, że nasza ocena co do użyteczności kodu lub jego działania była błędna i funkcja będzie wymagała refaktoryzacji lub całkowitego usunięcia.

Dlatego zalecane jest stosowanie zasady YAGNI, czyli odkładanie pisania kodu na później, wtedy gdy będzie naprawdę potrzebne.

Czym jest klasa a czym obiekt?

Klasa to swojego rodzaju “wzór”, którego będziemy używać do tworzenia obiektów, czyli instancji danej klasy. Aby powstał obiekt, klasa musi być najpierw zdefiniowana.

Dla przykładu w ten sposób możemy zdefiniować klasę Person, a następnie utworzyć obiekt tej klasy.

```
public class Person
{

    public string FirstName { get; set; }

    public string LastName { get; set; }
}

Person person = new Person();
person.FirstName = "Jan";
```

Czym różni się JSON od XML?

Zarówno JSON jak i XML są formatami, w których możemy zapisać zserializowane obiekty.

JSON jest znacznie lżejszym formatem, przez co jest często preferowany w komunikacji między frontendem a backendem aplikacji webowych.

JSON nie używa tagów tak jak robi to XML, a domyślnie jest wspierany przez język JavaScript, przez co jest bardzo wygodnym rozwiązaniem dla aplikacji w tej technologii.

Przykładowy obiekt, przedstawiający te same dane prezentuje się następująco:

JSON:

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```

XML:

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

Czym jest serializacja?

Serializacja to proces konwertowania obiektu, który mamy w pamięci programu, np. na strumień bajtów w celu jego zapisu do bazy danych lub do pliku.

Jego głównym celem jest zapisanie stanu obiektu, aby można było go odtworzyć w razie potrzeby. Proces odwrotny jest nazywany deserializacją.

JSON i XML to najpopularniejsze formaty, do których serializujemy obiekty.

Aby zserializować obiekt w C# do JSON, możemy użyć popularnej paczki Nugetowej: *NewtonsoftJson*, lub skorzystać z wbudowanego serializatora: klasy *JsonSerializer*.

Czym są wzorce projektowe?

Wzorce projektowe to uniwersalne, sprawdzone w praktyce rozwiązania często pojawiających się problemów projektowych.

Każdy ze wzorców projektowych stanowi swojego rodzaju plan, który po odpowiednim zastosowaniu pomoże w konkretnym problemie który napotkamy przy tworzeniu oprogramowania.

Wzorce można sklasyfikować na podstawie ich przeznaczenia. Wyróżniamy 3 główne kategorie wzorców:

- Wzorce kreacyjne
- Wzorce strukturalne
- Wzorce behawioralne

Jak działa wzorzec singleton?

Singleton to kreacyjny wzorzec projektowy, dzięki któremu możliwość tworzenia konkretnego obiektu, będzie ograniczona do tylko jednej instancji. Ta instancja będzie globalnie dostępna w całym projekcie, udostępniając swoje właściwości do odczytu dla innych klas.

Singleton jest jednak często nadużywany przez co wiele osób uważa go za anty-pattern, jednak nadal ma on swoje zastosowanie, np. w dostarczaniu statycznej konfiguracji dla klas które potrzebują takiej informacji.

Jak działa wzorzec fabryka?

Fabryka jest kreacyjnym wzorcem projektowym, umożliwiającym tworzenie spokrewnionych ze sobą obiektów bez konieczności określania ich konkretnych klas.

Obiekty powiązane ze sobą na podstawie klasy bazowej, będą tworzone przez fabrykę w zależności od potrzeb.

Dzięki temu, zamiast tworzyć każdy obiekt osobno, uwzględniajac konkretne parametry, wystarczy że odwołamy się do fabryki, która dany obiekt zwróci.

Jaki jest cel wzorca strategia?

Strategia to behawioralny wzorzec projektowy umożliwiający zdefiniowanie powiązanych algorytmów, umieszczając je w osobnych klasach, które będą między sobą wymienalne dzięki wspólnemu interfejsowi.

Często w programach, które tworzymy, w zależności od jakiejś zmiennej, która np. może być pobrana od użytkownika, będziemy chcieli zastosować konkretny algorytm, dla przykładu do generacji dokumentu w różnych formatach: pdf, word czy txt.

Aplikując strategie do tego problemu będziemy w stanie wydzielić wspólny interfejs dla tych trzech metod generacji pliku i umieścić je w osobnych klasach.

Natomiast miejsce, w którym wołany będzie ten algorytm, będzie oparte właśnie o ten interfejs, dzięki czemu kod nie będzie zależny od konkretnej implementacji i w łatwy sposób taki kod można rozszerzać o dodawanie kolejnych implementacji algorytmu.

C# .NET

Jakie modyfikatory dostępu istnieją w C#?

Modyfikatory dostępu, to słowa kluczowe, które określają gdzie dany typ lub właściwość danego typu będzie widoczna. Dla typów są to: *private*, *public*, *internal*.

A dla właściwości typów (np. metod czy pól), są to:

- *Private*
- *Protected*
- *Internal*
- *Protected internal*
- *Public*

Jakie metody posiada każdy obiekt?

Każdy obiekt w języku C#, dziedzyczy po typie *System.Object*, który to udostępnia 4 publiczne metody:

- *ToString*
- *Equals*
- *GetHashCode*
- *GetType*

Po czym mozna dziedziczyć w c#?

W C# można dziedziczyć tylko po jednej klasie bazowej. Może to być zwykła klasa lub klasa abstrakcyjna.

Natomiast oprócz dziedziczenia można też wykorzystać mechanizm implementowania interfejsów, a tutaj nie ma już ograniczenia jak wiele możemy ich zaimplementować.

Czym jest sterta i stos?

Sterta (heap) i stos (stack), to miejsca w pamięci wirtualnej, która jest przydzielona do konkretnego programu.

Zarówno na stercie jak i na stosie są przechowywane wartości, których używamy w programie, a w zależności od tego czy dana wartość jest typem referencyjnym czy wartościowym, to znajdzie się albo na stercie, albo na stosie.

Zmienne typu wartościowego (np. int lub char) trafiają na stos, a dostęp do nich jest bezpośredni, co za tym idzie bardzo szybki.

Natomiast zmienne typów refercyjnych trafiają na sterte i to właśnie tą pamięć będą zajmować, a dodatkowo referencja do danego obiektu ze sterty, zostanie przechowana na stosie.

Ta referencja wskazuje na adres komórki ze sterty pod którą przechowywany jest dany obiekt.

Jaka jest różnica pomiędzy przesłanianiem a nadpisywaniem metod?

Jeżeli w klasie bazowej mamy metodę wirtualną i jakąś inną metodę, to w klasie dziedziczącej, będziemy w stanie nadpisać lub przesłonić te metody.

Aby nadpisać metodę wirtualną konieczne jest użycie słowa kluczowego *override*, a do przesłaniania możemy wykorzystać słowo kluczowe *new*.

Różnica jest taka, że mając jakaś metodę nadpisaną, to odwołując się do tej metody, poprzez obiekt typu bazowego, wywołana zostanie metoda nadpisania (w sposób polimorficzny).

Natomiast przy przesłanianiu, to odwołując się do metody przesłoniętej, przez obiekt typu bazowego, zostanie wywołana metoda z typu bazowego.

Czym są słowa kluczowe *ref* i *out*?

Słowa kluczowe *ref* i *out*, pozwalają nam odnieść się do referencji danej zmiennej. Dzięki temu będziemy mogli przekazać zmienną typu wartościowego do jakieś metody i przekazując ją ze słowem kluczowym *ref* lub *out* zmieniając wartość tej zmiennej w środku metody, to zmieni się ona też poza tą metodą.

Jeżeli przekazalibyśmy zmienną typu wartościowego do metody bez żadnego z tych dwóch słów kluczowych, to wtedy w metodzie operowalibyśmy na kopii tej zmiennej, przez co żadna jej modyfikacja nie miała by efektu.

Różnica między nimi jest taka, że używając *ref*, zmienna musi już być uprzednio zainicjalizowana, a z kolei używając *out* – wtedy zmienna nie musi być zainicjalizowana przed jej przekazaniem, ale za to musi być zainicjalizowana w ciele metody

Czym jest refleksja?

Refleksja to mechanizm, za pomocą którego można uzyskać dostęp do metadanych klas i obiektów.

Metadane to po prostu informacje opisujące pola, metody, właściwości i wszystkie inne elementy danego typu.

Wykorzystując refleksje możemy np. przeiterować po wszystkich właściwościach publicznych danego obiektu, filtrować je i zmieniać w dynamiczny sposób podczas działania programu.

Z refleksji korzystają różnego rodzaju biblioteki, np. do mapowania obiektów, serializacji lub Entity Framework.

Czym są typy generyczne?

Typy generyczne pozwalają na opóźnienie w dostarczeniu specyfikacji typu danych w elementach takich jak metody, właściwości lub klasy.

Inaczej mówiac, typy generyczne umożliwają na utworzenie klasy lub metody, która będzie działać z każdym typem danych.

Typy generyczne powstały ze względu na kolekcje dynamiczne, które przed typami generycznymi były mało efektywne, ze względu na boxing i unboxing elementów.

Czym są generyczne ograniczenia?

Generyczne ograniczenia to mechanizm, pozwalający nam ograniczyć typy, które są przekazywane jako generyczne parametry, do klas, metod czy właściwości.

Aby utworzyć ograniczenia generyczne, konieczne jest użycie słowa kluczowego *where*, za pomocą którego możemy określić, że dany typ będzie np. implementował konkretny interface, czy będzie miał jakąś bazową klasę, lub będzie miał konstruktor bezparametrowy.

Czym są generyczne delegaty?

Generyczne delegaty to predefiniowane typy delegat, do których możemy w generyczny sposób przekazać typy wejściowe i typ zwracany z danej delegaty.

Służą one np. do przekazywania funkcji anonimowych, jako parametrów metod przez co kod, który piszemy będzie bardziej reużywalny.

Typowym wykorzystaniem generycznych delegat jest przekazanie funkcji anonimowej, jako wyrażenia lambda, do dowolnej metody LINQ.

Trzy predefiniowane generyczne delegaty to:

- ***Action***: delegata przyjmująca do 15 parametrów, nie zwracająca żadnej wartości
- ***Predicate***: delegata przyjmująca do 15 parametrów, zwracająca zawsze wartość *Boolean*
- ***Func***: delegata przyjmująca do 15 parametrów, zwracająca dowolny typ

Jak działa wyrażenie *using*?

Wyrażenie *using* sprawia, że używając zasobów, które wymagają zwolnienia pamięci: np. połączenie do bazy danych lub zczytywanie pliku z dysku, zostaną poprawnie zamknięte nawet w przypadku wystąpienia wyjątku w programie.

Możemy wykorzystać go tylko dla obiektów typów, które implementują interface IDisposable.

Blok kodu, który definiujemy w using statement, to tak na prawdę blok *try*, który jest tworzony przez kompilator. Dodatkowo poza tym blokiem try, kompilator również tworzy blok *finally*, w którym to wywołuje metodę *Dispose*, dla konkretnego obiektu.

I właśnie dzięki temu mimo wystąpienia wyjątku w danym bloku kodu, mamy pewność, że metoda *Dispose* zostanie wywołana.

Jak działa *async* / *await*?

Async i *await* to słowa kluczowe, które pozwalają nam na pisanie kodu asynchronicznego. Kod asynchroniczny, to taki, którego wykonywanie nie blokuje głównego wątku. Dzięki temu nasz program będzie miał więcej dostępnych zasobów. Oprócz tego w aplikacji, która renderuje user interface, to użycie *async* i *await*, sprawi, że aplikacja nie zostanie zawieszona na czas wykonania np. zapytania HTTP.

Użycie słowa kluczowego *await* jest możliwe tylko w metodzie zadeklarowanej ze słowem kluczowym *async*. Dodatkowo taka metoda musi być typu:

- *Void*
- *Task*
- *Task<T>* (gdzie T to zwracany typ generyczny)
- *ValueTask*
- *ValueTask<T>*

Jak wykonać akcje równolegle?

Aby wykonać kilka akcji (np. zapytań do bazy danych) równolegle, możemy wykorzystać programowanie asynchroniczne, a konkretnie metody *Task.WhenAll* lub *Task.WhenAny*.

Do tych dwóch metod, możemy przekazać dowolną ilość metod asynchronicznych, które będą wykonywać się równolegle.

Różnica między *Task.WhenAll* a *Task.WhenAny* jest taka, że oczekując je (czyli dodając słowo kluczowe *await*), *Task.WhenAny* zakończy swoje działanie, kiedy jakakolwiek metoda asynchroniczna zakończy swoje działanie.

Z drugiej stony *Task.WhenAll* zakończy swoje działanie, kiedy wszystkie metody do niej przekazanie zakończą swoje działanie.

Czy z metody asynchronicznej można złapać wyjątek?

Wyjątki, które wystąpiły w metodach asynchronicznych są możliwe do złapania tylko jeśli metoda asynchroniczna nie była oznaczona typem void.

Także dla Task, Task<T>, ValueTask, ValueTask<T>, będziemy w stanie złapać wyjątek jeżeli wystąpił podczas procesowania takiej metody.

A przy *async* void, wyjątek spowoduje crash programu, ponieważ nie będziemy w stanie go obsłużyć w bloku *try catch*.

Czym są atrybuty?

Atrybuty to specjalne klasy, za pomocą których będziemy w stanie nałożyć dodatkowe metadane na konkretne typy, metody, parametry lub właściwości.

W połączeniu z refleksją, będziemy w stanie odczytać dodatkowe informacje o konkretnym typie bez zmieniania jego definicji.

Przykładowymi atrybutami, mogą być atrybuty walidacji, serializacji lub atrybuty z czasownikami metod HTTP we frameworku ASP.NET.

Można też definiować własne atrybuty, tworząc klase, która będzie dziedziczyć po klasie Attribute

Czym jest (un)boxing?

Boxing to process konwertowania zmiennej typu wartościowego na typ object lub jakikolwiek interface implementowany przez tą zmienną typu wartościowego.

Kiedy runtime wykonuje boxing dla typu wartościowego, "opakowuje" go w instacje typu **System.Object** i przechowuje na stercie. Unboxing to process odwrotny czyli zmiana z obiektu na typ wartościowy. Boxing jest niejawny a unboxing jawny.

Przykładowo tak wygląda boxing i unboxing wartości typu int.

```
int i = 123;
object o = i; // boxing
i = (int)o;   // unboxing
```

Czym są słowa kluczowe *const* i *readonly*?

Oba te słowa kluczowe służą do deklarowania zmiennych niemodyfikowalnych. *Const* jest wartością statyczna, znaną już podczas kompilacji programu i w żaden sposób nie jesteśmy w stanie jej ustawić podczas działania aplikacji. Jej wartość możemy ustawić tylko podczas deklaracji.

Natomiast *readonly* może, ale nie musi być wartością statyczną, jej wartość jest przetwarzana podczas działania programu (w runtimie).

Oprócz tego zmienną *readonly* możemy przypisać zarówno w jej deklaracji jak i w konstruktorze danej klasy (co przy *const* było nie możliwe)

Jak działa słowo kluczowe *yield*?

Słowo kluczowe *yield* służy to tworzenia iteratora dla konkretnych elementów. Za każdym razem, gdy w metodzie wywołamy wyrażenie *yield return <element>*, to ta wartość zostanie natychmiastowo zwrócona do miejsca wywołania.

Sama metoda, w której wywołaliśmy *yield return*, nie zakończy swojego procesowania, dopóki miejsce wywołania będzie chciało uzyskać następne elementy.

Deklaracja iteratora musi spełniać następujące wymagania:

1. Zwracany typ musi być jednym z następujących typów:

    - IAsyncEnumerable<T>

- IEnumerable<T>
- IEnumerable
- IEnumerator<T>
- IEnumerator

2. Deklaracja nie może mieć żadnych parametrów *ref* ani *out* w parametrach.

Jaka jest różnica miedzy metodą *First* a *Single*? (LINQ)

Jeżeli na jakiejś kolekcji wywołamy metodę *First* z predykatą, którą spełnia więcej niż jeden element, to rezultatem będzie pierwszy napotkany element.

Natomiast przy tej samej predykacie dla metody *Single*, zostanie wyrzucony wyjątek, jeżeli w danej kolekcji więcej niż jeden element spełnia tą predykatę.

Natomiast zarówo *First* jak i *Single* wyrzucą wyjątek, jeżeli w kolekcji nie ma żadnego elementu spełniającego predykatę.

Czym różni się metoda *First* od *FirstOrDefault*? (LINQ)

Jeżeli wywołując na jakieś kolekcji metodę *First*, przekażemy do niej delegatę, która nie zwróci żadnych rezultatów, to metoda *First* wyrzuci wyjątek informując o tym, że kolekcja nie zawiera pasujących elementów.

Natomiast wywołując metodę FirstOrDefault, zamiast wyrzucenia wyjątku, rezultatem będzie wartość domyślna.

Jaka jest różnica pomiędzy interfejsem a klasą abstrakcyjną?

Główna różnica między klasą abstrakcyjną a interfacem jest taka, że można implementować dowolną ilość interjesów, a dziedziczyć tylko po jednej klasie abstrakcyjnej. Poza tym, klasy abstrakcyjne mogą definiować konstuktory bazowe dla klas dziedziczących, co jest niemożliwe w intefejsach.

Klasy abstrakcyjne również mogą definiować właściwości jako *public*, *private* i *protected*, natomiast interfejs tylko jako *public*.

Interfejs nie jest w stanie zadeklarować pól ani przypisać wartości do właściwości.

W nowszych wersjach C#, interfejs jest w stanie zadeklarować statyczne właściwości oraz przypisać domyślną implementacje dla metod, dla których nie jest konieczna implementacja.

Czym jest metoda rozszerzająca?

Metoda rozszerzająca pozwala nam dopisać dodatkową funkcjonalność do istniejącego już typu poza jego definicją.

Dla przykładu dla typów, które istnieją w systemowych bibliotekach, jak np. typ string, możemy utworzyć metodę rozszerzającą, dzięki czemu będziemy w stanie ją wywołać bezpośrednio na zmiennej typu string.

Aby utworzyć metodę rozszerzającą musimy spełnić następujące wymagania:

- Musi znajdować się w statycznej klasie
- Musi to być publiczna, statyczna metoda
- Jej pierwszym parametrem, musi być zmienna typu, który chcemy rozszerzyć poprzedzona słowem kluczowym *this*

Czym różni się operator *is* od *as*?

Operator is służy do sprawdzania, czy typ konkretnego obiektu jest zgodny z danym typem, czy nie.

Natomiast operator as służy do wykonywania konwersji między typami referencyjnymi lub typami dopuszczającymi wartość null.

Operator is jest typu logicznego, podczas gdy operator as nie jest typu logicznego.

Operator is zwraca wartość true, jeśli dany obiekt jest tego samego typu, z którym go porównujemy, natomiast operator as zwraca obiekt, gdy są one zgodne z danym typem.

Operator is zwraca wartość false, jeśli dany obiekt nie jest tego samego typu, natomiast operator as zwraca wartość null, jeśli konwersja nie jest możliwa.

Operator is jest używany tylko do konwersji odwołań, boxingu i unboxingu, podczas gdy operator as jest używany tylko do konwersji z wartościami null.

Jakie są nowości w C#?

Wraz z premierą .NET 6, C# w wersji 10, pojawiło się kilka nowych feature'ów:

- File scoped namespaces
- Global usings
- Implicit usings
- Interpolated const strings
- Ulepszenia wyrażeń lambd
- Ulepszenia dekonstrukcji

Entity Framework

Co robi metoda *Include*?

Metoda *Include* użyta na obiekcie typu *IQuerable*, pozwala na załadowanie danych powiązanych.Dane są po prostu referencją do innej tabeli, związanych kluczem obcym na poziomie bazy danych. Include zostanie przekształcony na polecenie ***INNER JOIN*** dla konkretnej tabeli.

W metodzie *Include* musimy wskazać jaką właściwość chcemy załodować, a możemy to zrobić albo używając delegaty, albo wartości string, wskazującej daną tabele.

```
1   class User
2   {
3       public int Id { get; set; }
4       public string Name { get; set; }
5       public ICollection<Role> Roles { get; set; }
6
7   }
8
9   class Role
10  {
11      public int Id { get; set; }
12      public string Name { get; set; }
13      public int UserId { get; set; }
14  }
15
16  ..
17
18
19  var usersWithRoles = dbContext
20                          .Users
21                          .Include(u => u.Roles)
22                          // .Include("Roles")
23                          .ToList();
```

Czym jest lazy loading?

Lazy loading jest mechanizmem polegającym na opóźnieniu ładowania obiektu, aż do momentu w którym tych danych potrzebujemy.

Mówiąc inaczej, obiekt ładujemy na żądanie, zamiast niepotrzebnego, wcześniejszego ładowania danych.

Najprostszym sposobem korzystania z ładowania z opóźnieniem jest zainstalowanie pakietu *Microsoft.EntityFrameworkCore.Proxies* i włączenie go za pomocą odpowiedniego skonfigurowania.

Na przykład:

```
.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
          .UseSqlServer(myConnectionString));
```

Oprócz tej konfiguracji wymagane jest jeszcze aby właściwości nawigacji, które chcemy dołaczyć poprzez lazy loading, były oznaczone słowem kluczowym *virtual*.

Jak działają migracje?

Po dokonaniu zmian na klasie z kontekstem bazy danych, jesteśmy w stanie utworzyć migrację, wywołując polecenie '*add-migration <nazwa>*'.

Migracja ta zostanie utworzona na podstawie porównania obecnego stanu klasy *DbContext*, a snapshotu, który przechowuje 'zserializowaną' postać poprzedniej wersji *DbContext* (czyli aktualna postać rzeczywistej bazy danych).

Na podstawie różnic między tymi dwoma rzeczami EF wygeneruje klasę z migracją, zawierającą odpowiednie kroki, które po wykonaniu doprowadzą do zaktualizowania bazy danych do najnowszej wersji, po wykonaniu polecenia '*update-database*'.

EF automatycznie przechowuje informacje o wykonanych migracjach w tabeli *_EfMigrationsHistory*, a przez to, nie wykona tej samej migracji więcej niż raz przy wykonaniu polecenia *update-database*.

Czym jest tracking?

Tracking (śledzenie) to mechanizm określający czy Entity Framework będzie przechowywać informacje o zmianach jednostek w swoim monitorze zmian.

Jeżeli dana encja jest śledzona, to wszystkie zmiany wykryte w tej encji zostaną utrwalone w bazie danych podczas wywołania metody *SaveChanges*.

Domyślnie zapytania, które zwracają encje bazodanowe, wykorzystują tracking. Oznacza to, że można wprowadzać zmiany w tych wystąpieniach jednostek i utrwalać je przez *SaveChanges()*. Jednak tracking ma swój narzut zwiększonej ilości pamięci jak i CPU. Także jeżeli nie chcemy śledzić zmian danych jednostek, możemy wykorzystać zapytania bez śledzenia.

Możemy wyłączyć śledzenie na dwa sposoby:

- Globalnie na całym DbContext
- Na konkretnym zapytaniu

Przykład zapytania bez trackowania z wykorzystaniem metody AsNoTracking:

```
1  var blogs = context
2      .Blogs
3      .AsNoTracking()
4      .ToList();
5
```

Wyłączenie śledzenia na całym DbContext'cie:

```
7
8   context.ChangeTracker.QueryTrackingBehavior
9       = QueryTrackingBehavior.NoTracking;
10
11  var blogs = context.Blogs.ToList();
```

Jakie wyróżniamy stany encji?

Każda encja może być w jednym z pięci stanów:

- *Added*: encja jest śledzona przez kontekst, ale jeszcze nie istnieje w bazie danych
- *Unchanged*: encja jest śledzona przez kontekst i istnieje w bazie danych, a jej wartości właściwości nie zmieniły się od wartości w bazie danych
- *Modified*: encja jest śledzona przez kontekst i istnieje w bazie danych, a niektóre lub wszystkie jej wartości właściwości zostały zmodyfikowane
- *Deleted*: encja jest śledzona przez kontekst i istnieje w bazie danych, ale została oznaczona do usunięcia z bazy danych przy następnym wywołaniu SaveChanges
- *Detached*: encja nie jest śledzona przez kontekst

Jaka jest różnica pomiędzy IEnumerable a IQueryable?

Wszystkie operacje wykonywane na kolekcji przez interfejs *IEnumerable*, czyli np. filtrowanie czy sortowanie będą zawsze wykonywane w pamięci programu.

Natomiast *IQueryable* jest dedykowany dla bibliotek ORM, takich jak np. Entity Framework, dzięki czemu te same operacje (czyli np. filtrowanie czy sortowanie) będą wykonywane po stronie bazy danych.

Do programu trafią dane uprzednio przygotowane, przez co aplikacja będzie wydajniejsza i będzie zużywać mniej pamięci.

ASP.NET

Jak działa protokół HTTP?

HTTP to protokół, który definiuje zasady wymiany informacji urządzeń w internecie.
Klient (np. przeglądarka internetowa) wysyła zapytanie na konkretny adres serwera, wraz z opcjonalnymi: nagłówkami, parametrami lub ciałem zapytania.

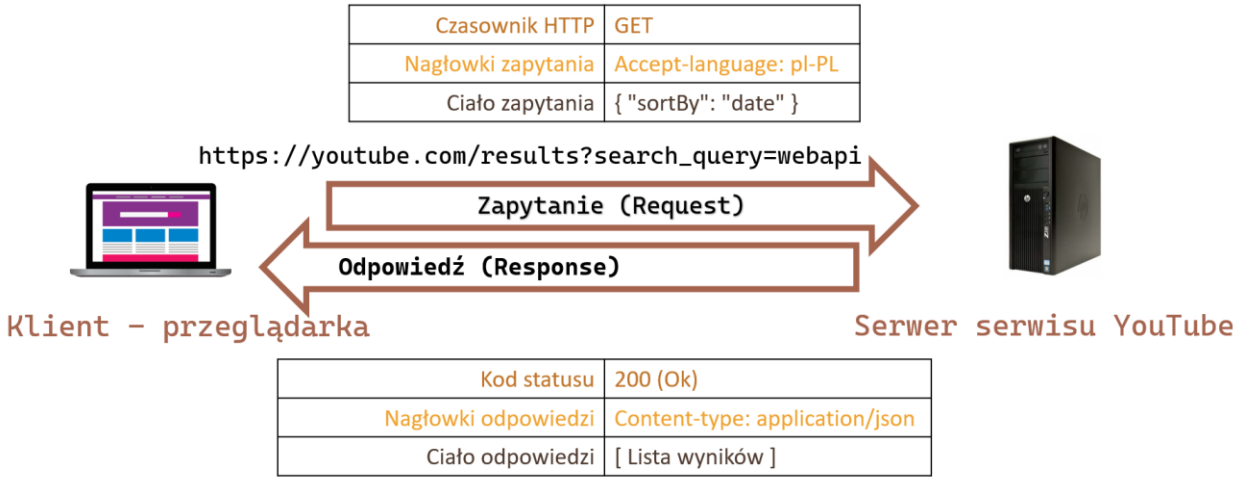
Serwer po otrzymaniu takiego zapytania, odpowiednio na podstawie wszystkich zebranych inforamcji od klienta przeprocesowuje takie zapytanie i zwraca do klienta odpowiedz zawierającą:

- kod statusu (informacje w postaci wartości liczbowej określającą status przebiegu procesowania)

- nagłówki odpowiedzi

- ciało odpowiedzi

Przykładowe zapytanie HTTP wysłane z przeglądarki do serwera serwisu YouTube:



Serwer nie zapisuje informacji o żadnej sesji klienta – jest on bezstanowy.

Czym różni się uwierzytelnianie od autoryzacji?

Uwierzytelnianie to potwierdzenie, że podmiot jest tym, za kogo się podaje, czyli inaczej ujmując, jest to potwierdzenie jego tożsamości.

Najczęściej możemy spotkać uwierzytelnianie podczas logowania do dowolnego systemu, gdzie: Login jest wartością, która określa, za kogo podajemy się w danym systemie.

Autoryzacja to proces określania uprawnień danego podmiotu. Inaczej mówiąc autoryzacja pozwala na stwierdzenie czy dany podmiot (np. osoba) posiada dostęp do danego zasobu.

W kontekscie programowania aplikacji webowych autoryzacje najczęściej implementuję się poprzez role użytkowników, które mają dostęp do konkretnych akcji na serwerze.

Poza tym dostęp może być też autoryzowany na poziomie konkretnego zasobu.

Jak działa wbudowany kontener DI?

Wbudowany kontener DI, pozwala na rejestracje typów jako abstrakcje pod postacią interfejsu lub jako konkretną klasę.

Taki typ po rejestracji, będzie możliwy do wstrzyknięcia przez konstruktor dowolnego, innego typu, zarejestrowanego w kontenerze DI.

Aby zarejestrować typ w wbudowanym kontenerze, możemy wykorzystać jedną z 3 metod na kolekcji serwisów, w metodzie ConfigureServices:

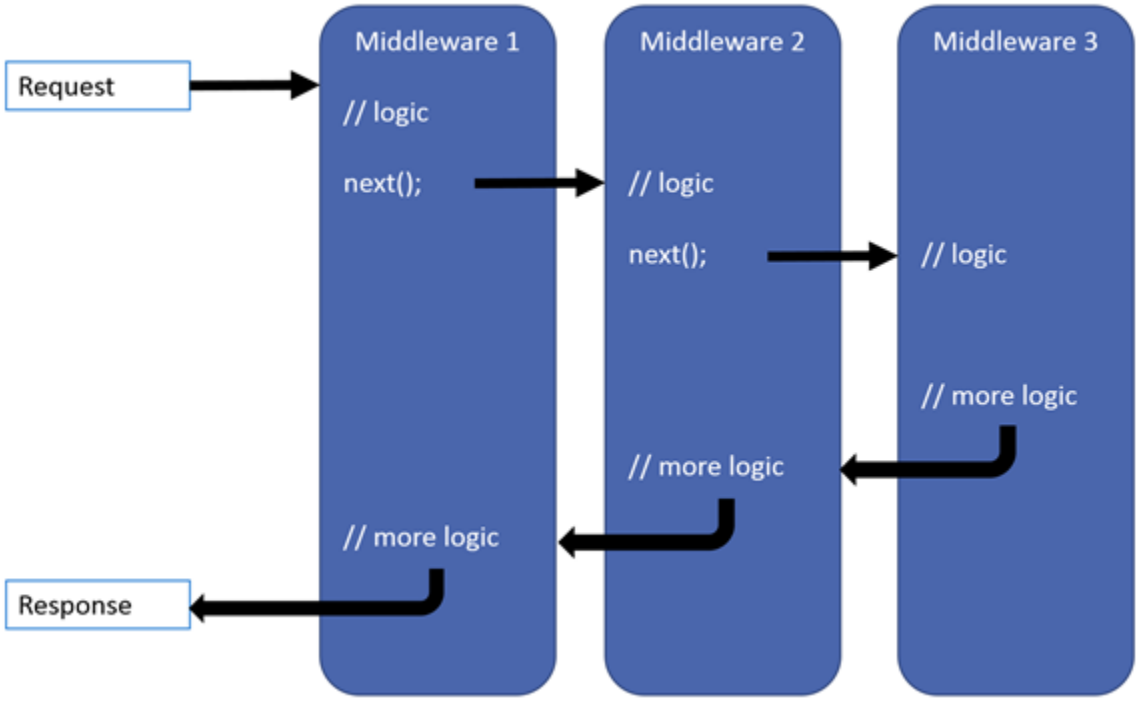
- AddSingleton
- AddScoped
- AddTransient

W zależności od wybranej metody, obiekt dostarczany przez dependency injection, będzie miał różne długości życia.

Czym jest Middleware?

Middleware, to specjalny kawałek kodu, który jest wykonywany podczas przetwarzania zapytania na serwerze.

Cała logika, która jest wykonywana po stronie serwera, składa się z wielu middleware. Każdy middleware określa, czy przekazywać żądanie do następnego middleware czy nie. Może też wykonywać swoją pracę przed następnym middleware i po nim.



Aby dodać middleware do procesowania zapytania, musimy go najpierw zarejestrować w metodzie *ConfigureServices*, a następnie użyć w konkretnym miejscu metody *Configure*.

Miejsce w którym użyjemy middleware jest istotnie, ponieważ w takiej kolejności będą one przetwarzane.

Jeżeli chcielibyśmy utworzyć customowy middleware, wystarczy, że do projektu dodamy klasę, implementującą interface *IMiddleware*, a następnie odpowiednio ją zarejestrujemy i użyjemy.

Jakie są kategorie kodów statusów HTTP?

Kody informacyjne: 1xx

kod	opis słowny	znaczenie/zwrócony zasób
100	Continue	Kontynuuj – prośba o dalsze wysyłanie zapytania
101	Switching Protocols	Zmiana protokołu
110	Connection Timed Out	Przekroczono czas połączenia. Serwer zbyt długo nie odpowiada.
111	Connection refused	Serwer odrzucił połączenie

Kody powodzenia: 2xx

kod	opis słowny	znaczenie/zwrócony zasób
200	OK	Zawartość żądanego dokumentu (najczęściej zwracany nagłówek odpowiedzi w komunikacji HTTP)
201	Created	Utworzono – wysłany dokument został zapisany na serwerze
202	Accepted	Przyjęto – zapytanie zostało przyjęte do obsłużenia, lecz jego zrealizowanie jeszcze się nie skończyło
203	Non-Authoritative Information	Informacja nieautorytatywna – zwrócona informacja nie odpowiada dokładnie odpowiedzi pierwotnego serwera, lecz została utworzona z lokalnych bądź zewnętrznych kopii
204	No content	Brak zawartości – serwer zrealizował zapytanie klienta i nie potrzebuje zwracać żadnej treści
205	Reset Content	Przywróć zawartość – serwer zrealizował zapytanie i klient powinien przywrócić pierwotny wygląd dokumentu
206	Partial Content	Część zawartości – serwer zrealizował tylko część zapytania typu GET, odpowiedź musi zawierać nagłówek Content-Range informujący o zakresie bajtowym zwróconego elementu

Kody przekierowania: 3xx

kod	opis słowny	znaczenie/zwrócony zasób
300	Multiple Choices	Wiele możliwości – istnieje więcej niż jeden sposób obsłużenia danego zapytania, serwer może podać adres zasobu, który pozwala na wybór jednoznacznego zapytania spośród możliwych
301	Moved Permanently	Trwale przeniesiony – żądany zasób zmienił swój URI i w przyszłości zasób powinien być szukany pod wskazanym nowym adresem
302	Found	Znaleziono – żądany zasób jest chwilowo dostępny pod innym adresem, a przyszłe odwołania do zasobu powinny być kierowane pod adres pierwotny
303	See Other	Zobacz inne – odpowiedź na żądanie znajduje się pod innym URI i tam klient powinien się skierować. To jest właściwy sposób przekierowywania w odpowiedzi na żądanie metodą POST.
304	Not Modified	Nie zmieniono – zawartość zasobu nie podległa zmianie według warunku przekazanego przez klienta
305	Use Proxy	Użyj serwera proxy – do żądanego zasobu trzeba odwołać się przez serwer proxy podany w nagłówku Location odpowiedzi

Kody błędu aplikacji: 4xx

kod	opis słowny	znaczenie/zwrócony zasób
400	Bad Request	Nieprawidłowe zapytanie – żądanie nie może być obsłużone przez serwer z powodu nieprawidłowości postrzeganej jako błąd użytkownika (np. błędna składnia zapytania)
401	Unauthorized	Nieautoryzowany dostęp – żądanie zasobu, który wymaga uwierzytelnienia

402	Payment Required	Wymagana opłata – odpowiedź zarezerwowana na przyszłość. Google Developers API korzysta z tego kodu, jeśli dany programista przekroczył dzienny limit zapytań.
403	Forbidden	Zabroniony – serwer zrozumiał zapytanie, lecz konfiguracja bezpieczeństwa zabrania mu zwrócić żądany zasób
404	Not Found	Nie znaleziono – serwer nie odnalazł zasobu według podanego URL ani niczego co by wskazywało na istnienie takiego zasobu w przeszłości
405	Method Not Allowed	Niedozwolona metoda – metoda zawarta w żądaniu nie jest dozwolona dla wskazanego zasobu, odpowiedź zawiera też listę dozwolonych metod

Kody błędu serwera: 5xx

kod	opis słowny	znaczenie/zwrócony zasób
500	Internal Server Error	Wewnętrzny błąd serwera – serwer napotkał niespodziewane trudności, które uniemożliwiły zrealizowanie żądania
501	Not Implemented	Nie zaimplementowano – serwer nie dysponuje funkcjonalnością wymaganą w zapytaniu; ten kod jest zwracany, gdy serwer otrzymał nieznany typ zapytania
502	Bad Gateway	Błąd bramy – serwer – spełniający rolę bramy lub pośrednika – otrzymał niepoprawną odpowiedź od serwera nadrzędnego i nie jest w stanie zrealizować żądania klienta
503	Service Unavailable	Usługa niedostępna – serwer nie jest w stanie w danej chwili zrealizować zapytania klienta ze względu na przeciążenie
504	Gateway Timeout	Przekroczony czas bramy – serwer – spełniający rolę bramy lub pośrednika – nie otrzymał w ustalonym czasie odpowiedzi od wskazanego serwera HTTP, FTP, LDAP itp. lub serwer DNS jest potrzebny do obsłużenia zapytania