Szoftverfejlesztés .NET platformra gyakorlatok

v1.7 - hetedik kiadás, 2024. június

Tóth Tibor <toth.tibor@vik.bme.hu>

Simon Gábor <simon.gabor@vik.bme.hu>

Szabó Gábor <szabo.gabor@podnet.hu>

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

1117 Budapest, Magyar tudósok körútja 2. (Q épület)

https://www.bme.hu
https://www.vik.bme.hu
https://www.aut.bme.hu

Tóth Tibor <toth.tibor@vik.bme.hu>

Simon Gábor <simon.gabor@vik.bme.hu>

Szabó Gábor <szabo.gabor@podnet.hu>

Hibajelentések, közreműködők: https://github.com/bmeviauav23/aspnetcorebook

ISBN 978-963-421-878-4

Tartalomjegyzék

1. Tudnivalók

1.1 Szoftverfejlesztés .NET platformra

Jegyzetek, gyakorlati anyagok és házi feladatok a Szoftverfejlesztés .NET platformra c. tárgyhoz.

Javítás az anyagban

A tárgy hallgatóinak a jegyzet anyagában történő javításért, kiegészítésért plusz pontot adunk! Ha hibát találsz a jegyzet bármely részében, vagy kiegészítenéd azt, nyiss egy *pull request*-et! A repository linkjét a jobb felső sarokban találod.

1.2 Tudnivalók

1.2.1 A jegyzet célja és célközönsége

Ezen jegyzet elsődlegesen a BME Villamosmérnöki és Informatikai Karán oktatott Szoftverfejlesztés .NET platformra című tárgyhoz készült, célja, hogy segítséget nyújtson egyrészt a gyakorlatvezetőnek a gyakorlat megtartásában, másrészt a kurzus hallgatóinak a gyakorlat otthoni utólagos megismétléséhez, a tanult ismeretek átismétléséhez.

Ebből kifolyólag nem tekinthető egy teljesen kezdő szintű bevezető C# tankönyvnek, hiszen erőteljesen épít más kari tárgyak (pl. Szoftvertechnikák, Adatbázisok) által lefedett ismeretekre, de még inkább a Szoftverfejlesztés .NET platformra című tárgy előadásaira.

A feltételezett előismeretek:

- C# és objektumorientált nyelvi alapok

  operátorok, változók, tömbök, struktúrák, függvények fogalma

  operátor felüldefiniálás és függvényváltozatok

  alapvető memóriakezelés (heap, stack), mutatók fogalma, érték és referencia típusok

  alapvető vezérlési szerkezetek (ciklus, elágazás, stb.), érték- és referencia szerinti paraméterátadás, rekurzió

  osztály, osztálypéldány fogalma, static, `new` operátor, osztály szintű változók, generikus típusok

  leszármazás, virtuális tagfüggvények

  C# esemény, delegate típusok és delegate példányok

  Visual Studio használatának alapjai

  operációs rendszer kapcsolatok, folyamatok, szálak, parancssor, parancssori argumentumok, környezeti változók
- SQL nyelvi alapok (SELECT, UPDATE, INSERT, DELETE utasítások), valamint alapvető relációs adatmodell ismeretek (táblák, elsődleges- és idegen kulcsok)

A fentiek elsajátításához segítséget nyújthatnak Reiter István ingyenesen letölthető könyvei.

A szövegben megtalálhatók a gyakorlatvezetőknek szóló kitételek („Röviden mondjuk el…", „Mutassuk meg…", stb.). Ezeket mezei olvasóként érdemes figyelmen kívül hagyni, illetve szükség esetén a kapcsolódó elméleti ismereteket az előadásanyagból átismételni.

1.2.2 A jegyzet naprakészsége

Az anyag gerincét adó .NET Core / .NET 5,6 platform jelenleg igen gyors ütemben fejlődik. A .NET Core 1.0-s verzió óta a készítők törekednek a visszafelé kompatibilitásra, azonban az eszközkészlet és a korszerűnek és ajánlottnak tekinthető módszerek folyamatosan változnak, finomodnak.

A jegyzet elsődlegesen az alábbi technológiai verziókhoz készült:

- C# **12**
- .NET **8**
- ASP.NET Core **8**
- Visual Studio **2022**

Ahogyan a fenti verziók változnak, úgy avulhatnak el a jegyzetben mutatott eljárások.

1.2.3 Szoftverkörnyezet

A gyakorlatok az alábbi szoftverekből álló környezethez készültek:

- Windows 11 operációs rendszer
- Visual Studio 2022 (az ingyenes Community verzió elég) az alábbi workloadokkal:
  - .NET desktop development
  - Data storage and processing
  - ASP.NET and web development
  - Azure Development
- Telerik Fiddler Classic
- Postman

A .NET (korábban .NET Core) széleskörű platformtámogatása miatt bizonyos nem Windows platformokon is elvégezhetők a gyakorlatok Visual Studio helyett Visual Studio Code használatával - azonban a gyakorlatok szövege a Visual Studio használatát feltételezi.

1.2.4 Kódrészletek változáskövetése

Az egyes gyakorlatok során gyakori eset, hogy a C# kód egy részét továbbfejlesztjük, megváltoztatjuk. Ilyen esetben a változó sorokat a jegyzetben kiemelt háttérrel rendelkeznek. A törölt kódrészleteket (amennyiben van segíti a megértést) kommentezéssel jelezzük. Jelöljük még a meglévő, de a jegyzetben nem megjelenített kódrészleteket komment és ... (`//...`) jellel.

```
using System; //ez egy korábban meglévő kódsor, változatlan
using static System.Console; //ez új kódsor

//... meglévő kódrészlet az előző feladatokból

foreach (var dog in dogs)   //ez egy korábban meglévő kódsor, változatlan
  /* Console.*/WriteLine(dog); //ez a sor megváltozott, az elejéről kód törlődött

/* Console.*/ReadLine();    //ez a sor megváltozott, az elejéről kód törlődött
```

> **JSON kommentek**
>
> A JSON formátum alapértelmezésben (RFC szerint) nem támogatja a kommenteket, így ha JSON kódrészletet másolunk, győződjünk meg arról, hogy nem maradt-e a beillesztett kódban komment, mert problémát okozhat.

1.3 Hozzájárulás az anyaghoz

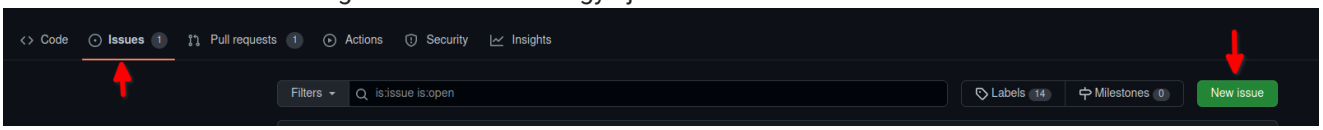
Az anyag terjedelméből adandóan apróbb hibák esetenként hiányosságok jelentkezhetnek a laborokban. Ha egy ilyennel találkozol és úgy döntesz szeretnél segíteni hallgatótársaidnak, azt a következőkben leírtak alapján tudod megtenni.

> **Plusz pont jegyzet javításért**
>
> Más tantárgyak mintájára itt is szeretnénk plusz pontot adni a jegyzet open-source hozzájárulásaiért. Akik a tárgyat jelenleg hallgatják, pontokat kaphatnak hozzájárulásaikérrt.
>
> A félév során max 3 db plusz pontot lehet szerezni fejenként olyan javításokért, amik a triviális 1-2 betű elgépelésen túl érdemben javítanak a githubon található labor jegyzetek minőségén. Pl.: jelentős mennyiségű elgépelés javítása, egyértelműsítések, illusztrációk kiegészítések készítése vagy akár egy teljes kiegészítő jegyzet írása (természetesen nem azonos pontértékkel).
>
> Persze a pont nélkül az 1-1 betűs elgépeléseket is szívesen fogadjuk, ami bemelegítésnek is tökéletes.

1.3.1 Hibák jelzése

Amennyiben hibát találsz az anyagban, vagy szeretnéd bővíteni, de nem áll módodban javítani, nyithatsz egy issue-t amiben leírod a hibát.

1. Nézd meg, hogy valaki nem jelezte-e, amit szeretnél. Gyakran már létező problémákat találnak, amire már van pull request, így mielőtt bármit tennél nézd meg valaki nem előzött-e meg
2. Az issues tabon a new issue gombbal hozz létre egy új issue-t.



3. Lásd el a megfelelő címkékkel
    a. A labor típusa (`android` az androidos laboroknál és `web` a webes laboroknál)
    b. A hiba típusa (`clarification` , `typo` , `illustration` vagy `notes`)
4. Írd le, hogy mit kéne tartalmaznia a javításnak

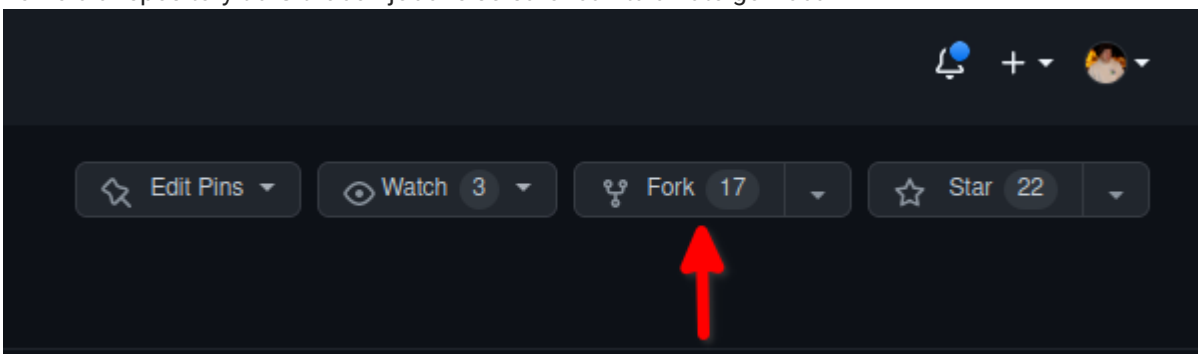
> **Tip**
>
> Az címe legyen rövid és lényegretörő, pl.: `Megfogalmazás pontosítása a 4. laborban` vagy `A 6. laborban a leírt kód hibásan működik Android 12-n`
>
> A issue descriptionjében pedig fejtsd ki, hol található a hiányosság, illetve ha van rá ötleted, hogy lehetne orvosolni ezt. Ha ezeken túl még screenshotot is tudsz mellékelni, az nagyban megsegíti a probléma mihamarabbi javítását.

> **Warning**
>
> A github issues nem a laborfeladatok megoldásával kapcsolatos problémák helye, így a "Nem tudom megoldani hogy az értesítés megérkezzen" jellegű problémákat ne itt jelezzétek, erre vannak a laboralkalmak.

1.3.2 Változtatások javaslása

Amennyiben a hozzájárulásod meg tudod valósítani indíts pull requestet

1. Forkold a repository-t a Githubon jobb felső sarokban található gombbal



2. Végezd el a változtatásokat.
    a. Hozz létre egy branchet a saját forkodon, amin a változtatásokat el fogod végezni.
    b. Ezen a branchen készítsd el a javításokat
    c. Ellenőrizd, hogy ne kerüljön bele a commitba olyan file, amit az editor generált (pl.: `.idea` mappa) illetve olyan file aminek nem kéne kikerülnie, pl.: Github Private Access Token
    d. Ha kész vagy indíts egy pull requestet a jegyzeteket tartalmazó repó fő ágára.
    e. A leírásban részletezd változtatások okát. Ne felejtsd el beleírni a NEPTUN kódod a leírásba, mert így fogjuk tudni megadni a pontokat.
3. Valaki, akinek hozzáférése van a repositoryhoz, ellenőrzi a változtatások szükségességét, és elbírálja, hogy valóban bekerülhet az anyagba.
4. A változtatásokra review-t indítunk és ha kell módosításokat fogunk kérni.
5. Ha minden kért változtatás megtörtént, a hozzájárulásod belekerül az anyagba.

1.3.3 Code style

- Markdown: Mivel az alap spec nem mindig a legtisztábban érthető, a markdownlint szabályai alapján, az néhány kivétellel. Ezeket a `.markdownlint.yaml` -ben találod, ha VSCode-ot használsz automatikusan alkalmazza őket az editor és jelzi ha nem megfelelő amit írsz.

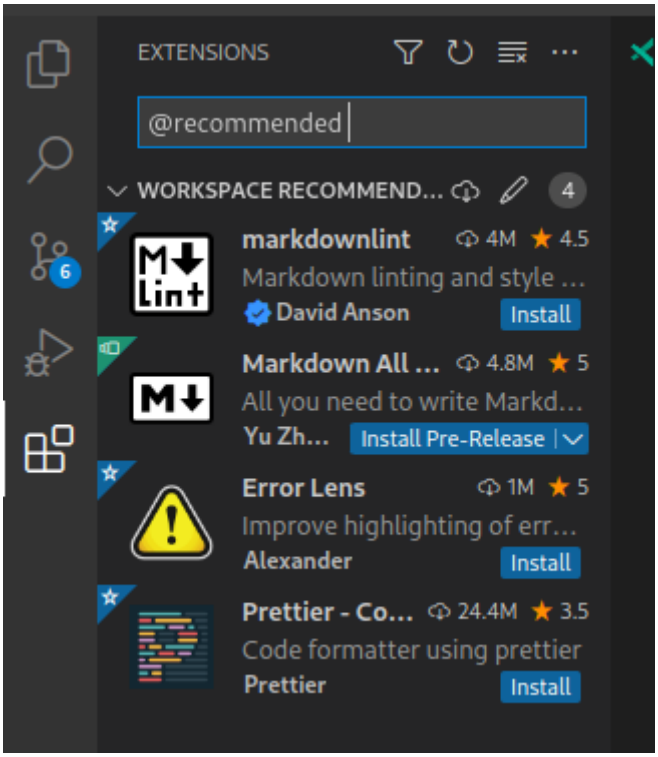
Ezek a stílusok a tárgyban ajánlott editorokban könnyen beállíthatóak.

VSCode

Ajánlott extensionök:

- `yzhang.markdown-all-in-one` : MD szinkronizált live preview
- `DavidAnson.vscode-markdownlint` : MD formázás, szabályok stb.
- Prettier: HTML+CSS formázó
- Error Lens: Kiemeli a hibákat hogy gyorsabben megtaláljuk őket
- Paste Image: egyszerűsíti a képek beillesztését markdownba

Az editor beállításához nyisd meg a repo-t a gyökerében VSCode-al. A VSCode fel fogja ajánlani a két markdown extension-t.



Ha ez megtörtént, nyiss meg egy markdown dokumentumot, és használd a Ctrl + Shift + P shortcutot, a command palette megnyitásához.

> **Tip**
>
> A command palette a VSCode parancsaihoz nyújt hozzáférést, autocompleteeli a parancsokat és egy minimális GUI-t is biztosít.

A command palette-be keressük meg a `Format Document With...` menüpontot és válasszuk ki. Ekkor egy almenübe dob az editor és kiválaszthatjuk hogy melyik formázóval formázzuk a MD dokumentumokat. Legalul lesz egy `Configure Default Formatter`, válasszuk ezt. Ezután válasszuk a `markdownlint` extensiont, és készen vagyunk.

> **Megfelelő formatter kiválasztása**
>
> Ne válaszd ki a prettiert formatterként, mert eltöri a szövegbuborékokat.

Ezen felül érdemes lehet bekapcsolni a mentés előtti formázást.

A Ctrl + , shortcuttal megnyitjuk a beállításokat, és rákeresünk arra, hogy format on save. Itt kipipáljuk a checkboxot és készen vagyunk.

Ha ehhez nem lenne türelmed, itt a json amit a `settings.json` -ba illesztve beállítódik minden.

```
{
  "[markdown]": {
    "editor.defaultFormatter": "DavidAnson.vscode-markdownlint",
    "editor.formatOnSave": true
  }
}
```

2. Laborok

2.1 C# alapok, szintaxis

Célunk, hogy a hallgatók legalább részben megértsék és ráérezzenek a C# szintaktikájára, megismerkedjenek alapvető nyelvi elemekkel és konstrukciókkal.

2.1.1 Hello C#!

A Visual Studio indítóablakában válasszuk a `Create a new project` opciót. Magyarázzuk el, hogy van lehetőségünk előre gyártott sablonokból létrehozni projekteket, illetve hogy

- egy C# projekt egy szerelvénnyé fordul (.dll, .exe).
- a *Solution* dolga, hogy logikailag összefogja a *Project*-eket (több-többes kapcsolatban vannak).
- a projektek között referenciákat adhatunk másik projektekre úgy, hogy a fordítási mechanizmus figyelembe veszi a referenciákat és szükség esetén újrafordítja a szerelvényeket.
- a projektek hivatkozhatnak külső forrásból származó szerelvényekre is NuGet csomagok formájában. A NuGet egy egységes módszer szerelvényeink terjesztésére.

Hozzunk létre egy új C# Console Application-t! Ehhez keressük ki a sablonok közül a Console App nevűt (ne a .NET Framework-öset). A neve legyen *HelloCSharp*.

> **New Project dialogusablak kereső**
>
> A kikereséshez használhatjuk felül a szövegdobozos szűrőt, illetve a legördülő listás szűrőket is (Nyelv: C#, Platform: Windows, Projekttípus: Console)

A sablon konfigurációjánál adjunk meg egy olyan helyet, ahová van írási jogunk. A *Place solution and project in the same directory* opciót kapcsoljuk be, így nem fog létrejönni egy felesleges mappa a könyvtárszerkezetben. A .NET verziót állítsuk .NET 8-ra.

Észrevehetjük, hogy az alkalmazás sablonok között sima és (.NET Framework) jelölésűek is vannak. A simák alapvetően a modernebb .NET Core/.NET 5-8 platformot célozzák, a *.NET Framework* ezekhez képest egy régebbi platform.

- **.NET Core**: a .NET Framework modularizált, modernizált, cross-platform és nyílt forráskódú megvalósítása. Kisebb NuGet csomagokban érhető el a teljes .NET Framework funkcionalitása (Collections, Reflection, XML feldolgozás, stb.).
- **.NET Framework**: a „klasszikus”, teljesértékű .NET keretrendszer, out-of-the-box támogatja a legelterjedtebb alkalmazásfejlesztési lehetőségeket. A .NET Core megjelenését követően is támogatott, enterprise környezetekben használatos, ugyanis néhány enterprise technológia elsődlegesen csak ebben támogatott (pl. szerver oldali WCF). Csak Windows-ra telepíthető.
- **.NET 5 és fölötte**: A .NET Core 3.1 utáni fő verziói. Már elnevezésében is jelzi, hogy ez egyben a korábbi .NET Core és .NET Framework verzióknak is utódja.

Az alábbi elemeket ismertethetjük, mielőtt a kódírásba belekezdünk:

- Rövid áttekintés az IDE-ről: menüsáv, Solution Explorer, Properties, Output, Error List ablakok, ablakozórendszer. Mutassuk meg, hogy drag-n-drop műveletekkel testreszabható a felület, pl. helyezzük a Solution Explorert a képernyő bal oldalára. Ha valaki véletlenül átrendezi az alapértelmezett elrendezést, a menu:Window[Reset Window Layout] lehetőséggel visszaállíthatja.
- A projekt tulajdonságok (menu:jobb klikk[Properties]) oldalán az Application fülön megnézhetjük, hogy az Output type értéke határozza meg, hogy milyen jellegű (konzolos, Windows, osztálykönyvtár) alkalmazást készítünk.
- Mutassuk meg, hogy milyen alapvető szerelvényekre adunk referenciát a projektben!
- Nézzük meg a `Program.cs` fájl tartalmát és fussuk át a látható elemeket!
- Magyarázzuk el a `using` és `namespace` kulcsszavak jelentését, egymáshoz képesti viszonyukat! A névtér értéke egy újonnan létrehozott fájlnál alapértelmezetten Projektnév.Mappaszerkezet alakú, érdemes konvencionálisan ezt követni. Sok hallgatónál nem tiszta, hogy hogyan viszonyul egymáshoz a névtér és a szerelvény fogalma, ezért próbáljuk meg ezt tisztázni!
- Utaljunk arra, hogy alapvetően kizárólag objektumorientáltan tudunk kódot írni, így a `Program` egy osztály, a `Main` belépési pont pedig egy statikus metódus.
- Beszéljünk röviden a C# elnevezési konvenciókról! A publikus elemeket (pl. Java-val és JavaScripttel ellentétben) és minden metódust ökölszabályként PascalCasing elnevezési konvenció követ, a nem publikus elemeknél camelCasing (ezek közül vannak kivételek és más konvenciók, de ez egy gyakori megközelítés).

Egészítsük ki a `Main` metódust az alábbi kódrészlettel, közben hívjuk fel a figyelmet az IntelliSense használatára:

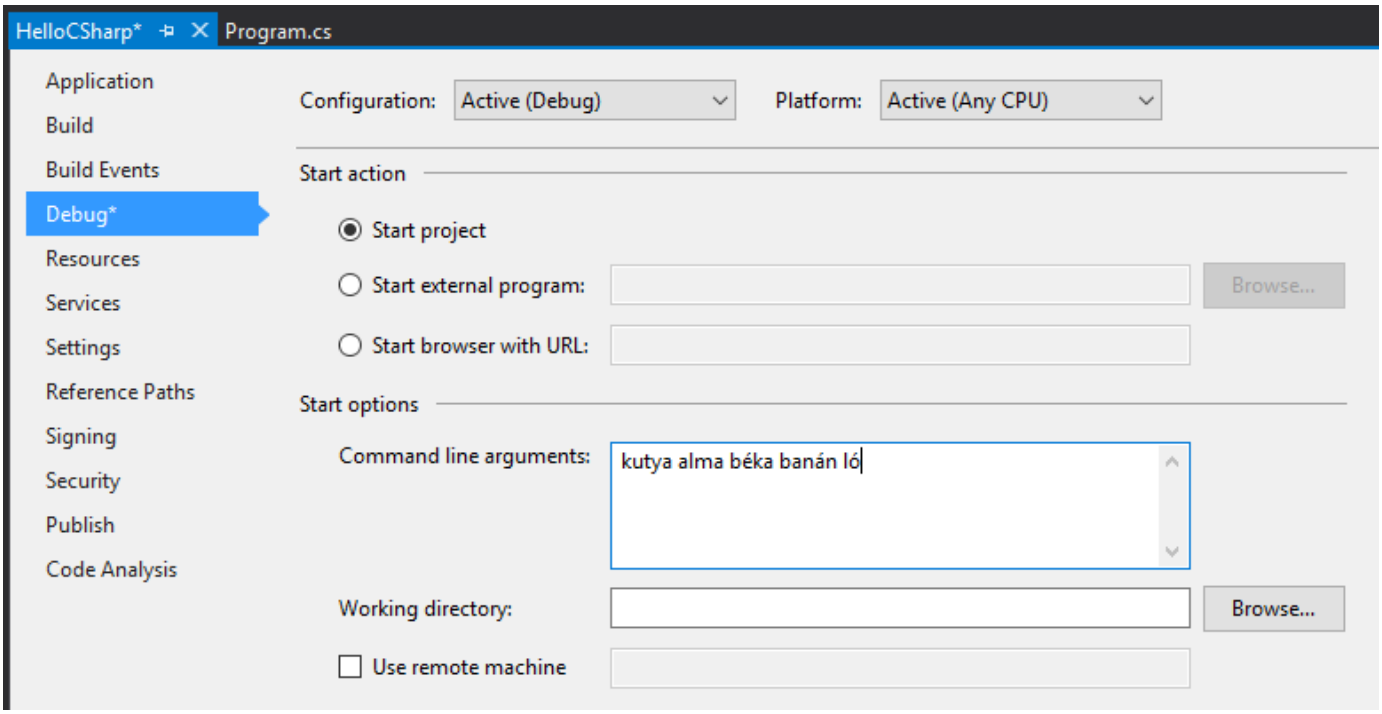
```
int a = 5;
int b = 7;
Console.WriteLine(a + b);
Console.ReadLine();
```

Az IntelliSense-t demonstrálhatjuk az alábbi módon:

- A kódban bármely logikus helyen használható az IntelliSense a Ctrl + Space billentyűkombinációval, ezen kívül alapértelmezetten felugrik kódírás közben is.
- Írjuk be a `Console` és a `.WriteLine()` elemeket úgy, hogy gépelés közben az IntelliSense legördülőből válasszuk ki az elemet, majd Tab billentyűvel véglegesítsük a választást.
- Használjuk a `cw` code snippetet, amit az IntelliSense is jelez, azaz írjuk be: `cw` majd nyomjunk kétszer Tab-ot.
- Ha a `Console.ReadLine()` helyett `Console.Readline()` -t írunk, elsőként az IDE azonnal javítja a hibát. Ha ezt a javítást visszavonjuk (Ctrl + Z), lehetőségünk van a javításra a Ctrl + . használatával: a fejlesztőeszköz észreveszi, hogy hibát vétettünk, és felkínálja a gyakori megoldásokat.
- *Overload*-ok: jelöljük ki a `WriteLine` hívás nyitó zárójelét, és írjuk be ismét a nyitó zárójelet. Így előjön az overload-ok listája, amik közül a megfelelőt a föl/le iránybillentyűkkel választhatjuk ki. Az overload listát megnyithatjuk úgy is, hogy a zárójelben bárhova írunk egy vessző karaktert. Az overload azt jelenti, hogy ugyanazzal a függvénynévvel több, különböző szignatúrájú metódust is felvehetünk, a megfelelő függvény kiválasztása a megadott paraméterek száma és típusa alapján történik.

Indítsuk el az alkalmazást! Ehhez a fent található Start lehetőséget használhatjuk, de mondjuk el, hogy ez a menü menu:Debug[Start Debugging] (F5) lehetőséggel ekvivalens.

Mutassuk be a `for` és `foreach` vezérlési szerkezeteket! A projekt *Properties* oldalán (kbd:[Alt+Enter] a projekt kijelölése után) adjunk meg a *Debug* fülön a *Start Options* blokknál legalább öt tetszőleges parancssori argumentumot (szóközzel elválasztva), pl. kutya alma béka banán ló.



.NET projektbeállítások

```
for (int i = 0; i < args.Length; i++)
    Console.WriteLine(args[i]);

foreach (string arg in args)
    Console.WriteLine(arg);

Console.ReadLine();
```

Indítsuk el, és gyönyörködjünk.

2.1.2 Debug

Rakjunk egy breakpointot (F9), vagy klikkeljünk baloldalon a függőleges sávon a kód sorszáma mellett) a `Console.WriteLine(args[i]);` sorra, majd indítsuk újra az alkalmazást! Amikor a breakpointon megáll az alkalmazás futása, a sor sárga színű lesz. Ekkor vigyük az egeret az `i`, az `args` és az `args.Length` elemek felé, és mutassuk meg, hogy láthatjuk az aktuális értékeiket, komplexebb objektumok esetén be tudjuk járni az objektumgráfot. A *Watch* ablakba is írhatunk kifejezéseket, és megmutathatjuk a *Locals* ablakot is. F10-zel (vagy a menüsoron a *Step Over* elemmel) lépjünk tovább, nézzük meg, milyen sorrendben értékelődik ki a `for` ciklus. Az F5-tel továbbengedhetjük az alkalmazás futását, majd zárjuk is be.

Mutassuk meg a Conditional Breakpoint használatát is. Tegyünk még egy breakpointot a másik `Console.WriteLine`-ra is. menu:Jobb egér gomb az első breakpointon[Conditions...], majd adjuk meg az alábbiakat: *Conditional Expression Is true* `(i == 3)`. A másik breakpointon is adjunk meg feltételt: *Hit Count = 4*. Mindkét alkalommal a 4. elemen (banán) állunk meg. Megjegyezhetjük, hogy a Conditional Breakpoint használatával nem érdemes mellékhatást okozó műveleteket megadni, illetve hogy jelentősen le tudja csökkenteni a debuggolás sebességét.

2.1.3 Tulajdonságok (Property-k)

Hozzuk létre a `Person` adatosztályt! Ehhez menu: jobb katt a projekten[Add > Class], a fájl neve legyen `Person` (a kiterjesztést automatikusan hozzábiggyeszti a Visual Studio, ha nem adjuk meg). .NET-ben nincs megkötés arra,

hogy a kódokat tartalmazó fájlok és az egyes típusok számossága hogyan viszonyul egymáshoz: lehetséges egy kódfájlba is írnunk a teljes alkalmazás-kódot, illetve egy osztályt is szétdarabolhatunk több fájlra (ehhez a `partial` kulcsszót használjuk).

A C# tulajdonság (*property*) egy szintaktikai édesítőszer, amely egy objektumpéldány (vagy osztály) egy explicit (memóriabeli) vagy implicit (származtatott vagy indirekt) jellemzőjét írja le. Egy tulajdonsággal két művelet végezhető: lekérdezés (*get*) és értékadás (*set*); ezeknek megadható külön a láthatósága, és a kettő közül elegendő egy implementálása. A legtöbb C# szintaktikai édesítőszer a boilerplate kódok írásának elkerülése végett készült, így kevesebb kódolással érjük el ugyanazt az eredményt (sokszor az IL kód nem is változik, gyakorlatilag hasonló a kódgeneráláshoz).

A `Person` osztályban hozzuk létre a `string Name` property-t, `name` osztályváltozóval (field). Ehhez használjuk a `propfull` code snippetet (`propf` , majd `Tab` + `Tab` , ezután `Tab` -bal lehet lépkedni a módosítandó elemek között):

```
public class Person
{
    private string name;
    public string Name
    {
        get { return name; }
        private set { name = value; }
    }

    public Person(string name)
    {
        this.name = name;
    }
}
```

> **Láthatóság**
>
> Figyeljünk az osztály láthatóságára is, alapból nem publikusként generálódik!

Igazából csak két további (kódban nem látható) metódust hozunk létre, mintha egy-egy `GetName` és `SetName` metódust készítenénk, viszont használat szempontjából ugyanolyannak tűnik, mintha egy sima mező lenne. A settert privát láthatóságúra tesszük, ezért csak egy `Person` példányon belülről tudjuk állítani a `Name` property értékét. Jegyezzük meg, hogy a getterben és setterben teljesen más jellegű műveleteket is végezhetünk (pl. elsüthetünk egy eseményt, hogy megváltozott a felhasználó neve, naplózhatjuk, hányszor kérték le a nevét, stb.). A property egyik nagy erénye, hogy osztályon kívülről az osztályváltozóknál megszokott szintaxissal használhatjuk.

A `Main` függvénybe írhatjuk például:

Program.cs

```
Person p = new Person("Eric Lippert");
p.Name = "Mads Torgersen";
Console.WriteLine(p.Name);
```

Debuggerrel figyeljük meg, hogy az első sor a konstruktort, míg a második a property setterét, végül a harmadik sor ugyanazon property getterét hívja.

Mivel a backing field állításán kívül nem csinálunk semmit a property kódban, ezért használhatjuk a `propg` code snippetet is:

Person.cs

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

Ez az ún. auto-implementált property szintaxis. A property által lekérdezhető-beállítható field generálódik, arra a kódban nem is tudunk hivatkozni - ez az egységbe zárás miatt előnyös.

A láthatóság miatt a `Main` függvényünkben a setter hívás már nem fordul, kommentezzük ki.

Person.cs

```
//p.Name = "Mads Torgersen";
```

Létezik még a `prop` code snippet is, ami mindkét módosítószót publikusan hagyja. Láthatósági módosítószót a `get` és `set` közül csak az egyik elé tehetünk ki, és az is csak szigoríthat a külső láthatóságon (ekkor a másik a külsőt kapja meg).

Ez a megoldás az előzővel teljes mértékben ekvivalens (csak nem látjuk a generált backing fieldet, de valójában ott van). Ha van időnk, akkor vizsgáljuk meg decompilerben (pl. Telerik JustDecompile), hogy valóban így van.

Az előzőhöz hasonlóan vegyük fel a születési dátumot is. A születési dátum nem változhat, gyakorlatilag `readonly` mezőről van szó. Ha egy tulajdonság értékét az objektum is csak a konstruktorban tudja megadni, akkor a setter teljes mértékben elhagyható:

Person.cs

```
public DateTime DateOfBirth { get; }

public Person(string name, DateTime dateOfBirth)
{
    Name = name;
    DateOfBirth = dateOfBirth;
}
```

Ez a szintaktika megegyezik azzal, mintha egy `readonly` mezőt használnánk, azaz a mező értéke legkésőbb a konstruktorban inicializálandó.

Vegyünk fel neki egy azonosítót, ami egy `Guid` struktúra típusú legyen:

Person.cs

```
public Guid Id { get; } = Guid.NewGuid();
```

Ez egy csak lekérdezhető tulajdonság, ami konstruáláskor inicializálódik egy új véletlenszerű azonosító értékre.

Megadhatjuk a kort, mint implicit/számított tulajdonságot:

Person.cs

```
public int Age { get { return DateTime.Now.Subtract(DateOfBirth).Days / 365; } }
```

Mivel a függvényünk törzse egyetlen kifejezéssel megadható, ezért elhagyva a sallangot (`return` , kapcsos zárójelek, stb.) *expression bodied property* szintaxissal is írhatjuk:

Person.cs

```
public int Age => DateTime.Now.Subtract(DateOfBirth).Days / 365;
```

> **Tip**
>
> Alkalmazások fejlesztésekor a legfontosabb első lépések egyike, hogy az objektummodellünk átlátható, karbantartható és egyértelmű legyen. A C# változatos szintaxisa nagyon sokat segít ezen célok elérésében.

2.1.4 Generikus kollekció

A `Main` metódusban vegyünk fel néhány `Person` objektumot, és listázzuk ki a releváns tulajdonságaikat! Ehhez egy `Person` listában tároljuk a személyeket. A `List` generikus kollekció, azaz típusparamétert vár, típusokkal paraméterezhető. A `List` típusparamétere jelzi, hogy milyen típusú objektumokat tárol. Metódusok, tulajdonságok, típusok lehetnek generikusak. A genericitás fontos a kódunk újrafelhasználhatósága és karbantarthatósága érdekében.

```
static void Main(string[] args)
{
   List<Person> people = new List<Person>();
   people.Add(new Person("Horváth Aladár", new DateTime(1991, 06, 10)));
   people.Add(new Person("Kovács István", new DateTime(1994, 04, 22)));
   people.Add(new Person("Kovács Géza", new DateTime(1998, 03, 16)));

   foreach (Person person in people)
      Console.WriteLine(person);

   Console.ReadLine();
}
```

Indítsuk el az alkalmazást, és nézzük meg, mi történik! Annyiszor íródik ki a `Person` osztályunk teljes neve (*fully qualified type name*), ahány elem van a listában.

2.1.5 Leszármazás, string interpoláció

Ha a `WriteLine` fölé visszük az egeret, látható, hogy az overload-ok közül az hívódik meg, amelyik objektumot vár paraméterül. Ebben az esetben a paraméter `ToString` metódusát hívja meg a `WriteLine`, ami alapértelmezés szerint az objektum típusának teljes nevét adják vissza. Tegyük szebbé a kiírást, definiáljuk felül az alapértelmezett `ToString` implementációt a `Person` osztályban:

Person.cs

```
public override string ToString()
{
   return string.Format("{0} ({1}) [ID: {2}]", Name, Age, Id);
}
```

A `Person` osztálynak nincs explicit megadva ősosztálya, mégis van felüldefiniálható függvénye. Ezeket az `Object` osztály definiálja. Ha egy referencia típusnak nincs megadva ősosztálya, akkor az `Object` lesz az.

A `ToString` implementációjára más szintaktikai édesítőszereket is használhatunk:

Person.cs

```
public override string ToString() => $"{Name} ({Age}) [ID: {Id}]";
```

A két implementáció ekvivalens, a második implementáció az ún. *expression bodied method* és a *string interpoláció* kombinálásából adódik.

Próbáljuk ki az alkalmazást!

Hozzuk létre a `Student` osztályt, ami származik a `Person` osztályból!

```
public class Student : Person
{
    public string Neptun { get; set; }

    public string Major { get; set; }

    public Student(string name, DateTime dateOfBirth)
        : base(name, dateOfBirth)
    {
    }

    public override string ToString() => $"{base.ToString()} Neptun: {Neptun} Major: {Major}";
}
```

Ez az osztály más megközelítéssel készült, mint a szülője, az állapota nem a konstruktor meghívásakor töltődik fel, utólag lehet megadni setter hívásokkal. Ez egyrészt kényelmes, mert nem kell sokparaméteres konstruktorokkal küzdeni, másrészt fel kell készülnünk arra, hogy bizonyos adatokat nem töltenek ki.

Ős konstruktora

Ha az ősosztálynak nincs paraméter nélküli konstruktora (a `Person` osztálynak nincs), akkor kötelesek vagyunk a gyerek konstruktorban az ősosztály valamelyik konstruktorát meghívni a `base` kulcsszóval.

2.1.6 Objektum inicializálók

Az *object initializer* segítségével az objektum létrehozását (konstruktor hívás) és a property setterek meghívásával történő inicializálását intézhetjük egy füst alatt. Az objektum inicializáló csak konstruktorhívás esetén használható, így pl. factory metódus által gyártott objektumpéldány esetén nem.

A `Main` metódusban írhatjuk az alábbi példát:

```
static void Main(string[] args)
{
    /*...*/
    people.Add(new Person("Kovács Géza", new DateTime(1998, 03, 16)));
    Student elek = new Student("Fel Elek", new DateTime(2002, 06, 10))
    {
        Neptun = "ABC123",
        Major = "Info BSc"
    };
    /*...*/
}
```

Konstruktor szintaktikák

Objektum inicializálás során, paraméter nélküli konstruktor esetén a `()` is elhagyható.

Több sorba tördelés

Általában 1-2 tulajdonság esetén lehet egy sorba is írni az inicializációt, több esetén viszont általában több sorba érdemes tördelni az olvashatóság érdekében.

Láthatjuk, hogy csak az aktuális kontextusban egyébként is látható és beállítható tulajdonságokat állíthatjuk be, egyik így beállított tulajdonság sem kötelező jellegű.

Az object initializer valóban csak az egyes tulajdonságokat állítja be, tehát csak szintaktikailag különbözik az első definíció az alábbitól:

Note

Nem kell beírni, csak szemléltetés.

```
Student _elek = new Student("Fel Elek", new DateTime(2002, 06, 10));
_elek.Neptun = "ABC123";
_elek.Major = "Info BSc";
Student elek = _elek;
```

A háttérben tényleg egy (számunkra nem látható) temporális változóban fog történni az inicializáció, ugyanis, ha az object initializer kivételt dob (az egyik setter által), az objektumunk nem veszi fel a kívánt értéket.

Objektum inicializáló haszna

Ebből látszik az objektum inicializáló elsődleges haszna, mégpedig, hogy nem kell állandóan kiírogatni, hogy melyik példányra gondolunk (így elrontani sem tudjuk).

2.1.7 Kollekció inicializáció

Az egyszerűsített kollekció inicializáció szintaxissal a lista teljes feltöltése jóval kevesebb kóddal és jóval olvashatóbban megadható. Ráadásul a kollekció elemeit létrehozhatjuk az objektum inicializációs szintaxissal is. A teljes lista létrehozást és -feltöltés részt cseréljük le az alábbira.

```
List<Person> people = new List<Person>
{
   new Person("Horváth Aladár", new DateTime(1991, 06, 10)),
   new Person("Kovács István", new DateTime(1994, 04, 22)),
   new Person("Kovács Géza", new DateTime(1998, 03, 16)),
   new Student("Fel Elek", new DateTime(2002, 06, 10))
   {
      Neptun = "ABC123",
      Major="Info BSc"
   },
   new Student("Hiány Áron", new DateTime(2000, 02, 13))
};
```

Nem kell az `Add` függvényhívást és a lista referenciát kiírni, egyértelmű, hogy melyik listához adunk hozzá.

Add

Ez a forma is ugyanolyan `Add` függvényhívásokra fordul, mint az eredeti változatban.

Próbáljuk ki az alkalmazást! Láthatjuk, hogy a konstruktoron keresztül teljesen inicializálható `Person` példányok esetében a kiírás teljes, viszont vannak olyan `Student` példányok, ahol a kiírás üres értékeket talál. Ezzel a jelenséggel a következő gyakorlatokon tovább foglalkozunk.

2.2 C# alapok II.

2.2.1 Előkészítés

Első lépésként hozzunk létre egy .NET C# konzolalkalmazást: a projektsablon szűrőben válasszuk a C# nyelv - Windows platform - *Console* projekttípust. A szűrt listában válasszuk a *Console App* sablont (**ne** a .NET Framework-ös legyen). A neve legyen *HelloCSharp2*. A solutiont ne tegyük külön mappába (*Place solution and project in the same directory* legyen bekapcsolva). A megcélzott framework verzió legyen .NET 8.

Legfelső szintű utasítások, implicit globális névtér-hivatkozások

Csodálkozzunk rá, hogy a generált projekt mindössze egyetlen érdemi sort tartalmaz.

```
Console.WriteLine("Hello, World!");
```

C# 10 óta a program belépési pontját adó forrásfájlt jelentősen lerövidíthetjük:

- a fájl tetején lévő using-okat elhagyhatjuk, ha azok implicit hivatkozva vannak. Az implicit hivatkozott using-ok projekttípustól függenek és a dokumentációból olvashatjuk ki
- a `Main` függvényt tartalmazó osztály deklarációját (`namespace` blokk, `class` blokk) elhagyhatjuk, ezt a fordító generálja nekünk
- a `Main` függvény deklarációját szintén generálja a fordító. A metódus neve nem definiált, nem (biztos, hogy) `Main`. A metódus szignatúrája attól függ, milyen utasításokat adunk meg a forrásfájlban. Például, ha nincs return, akkor `void` visszatérési értékű. A paramétere viszont mindig `string[] args`.
- a függvény blokkba nem foglalt kód a generált belépési pont függvény belsejébe kerül. Függvényt is írhatunk, az a belépési pontot tartalmazó generált osztály tagfüggvénye lesz.
- típusokat, osztályokat is definiálhatunk, de csak a legfelső szintű kódot követően

> **Warning**
>
> Fontos észrevétel a fentiekből: ezen képesség nem változtatja meg a C# semmilyen alapvető jellemzőjét, például ugyanúgy minden függvénynek osztályon belül kell lennie. A fordítás során a legfelső szintű utasítások kódja úgy egészül ki, ami már minden szabálynak megfelel.

> **Láthatóság**
>
> A legfelső szintű kód olyan, amit a program más részéről nem tudunk hívni, hiszen nem is ismerjük a burkoló osztály nevét. Emiatt nincs értelme legfelső szintű kódban láthatósági beállításnak (`private`, `protected` stb.) vagy propertynek.

Akadályozzuk meg a program azonnali lefutását egy blokkoló hívással.

```
Console.WriteLine("Hello, World!");
Console.ReadLine();
```

Próbáljuk ki a generált projektet mindenféle egyéb változtatás nélkül, fordítás (menu:projekten jobbklikk[Build]) után. Nézzünk bele a kimeneti könyvtárba (menu:projekten jobbklikk[Open Folder in File Explorer], majd menu:bin[Debug > net8.0]): látható, hogy az alkalmazásunkból a fordítás során egy cross-platform bináris (\<projektnév>.dll) és .NET Core v3 óta egy platform specifikus futtatható állomány (Windows esetén \<projektnév>.exe) is generálódik. Kipróbálhatjuk, hogy az exe a szokott módon indítható (pl. duplaklikkel), míg a dll a `dotnet` paranccsal.

```
dotnet <projektnév.dll>
```

Parancssor aktuális mappája

A dotnet parancshoz a dll könyvtárában kell lennünk. Ehhez a legegyszerűbb, ha a Windows fájlkezelőben a megfelelő könyvtárban állva az elérési útvonal mezőt átírjuk a `cmd` szövegre, majd kbd:[ENTER]-t nyomunk.

Adjunk a létrejövő projekthez egy `Dog` osztályt *Dog.cs* néven, ez lesz az adatmodellünk:

```
public class Dog
{
    public string Name { get; set; }
    public Guid Id { get; } = Guid.NewGuid();
    public DateTime DateOfBirth { get; set; }
    private int AgeInDays => DateTime.Now.Subtract(DateOfBirth).Days;
    public int Age => AgeInDays / 365;
    public int AgeInDogYears => AgeInDays * 7 / 365;
    public override string ToString() =>
        $"{Name} ({Age} | {AgeInDogYears}) [ID: {Id}]";
}
```

Az adatmodell az előző órán létrehozotthoz nagyon hasonlít, ennek viszont nincsen explicit konstruktora és a `Name` és `DateOfBirth` tulajdonságok publikusan is állíthatók.

Hozzunk létre egy `Dog` példányt objektum inicializációs szintaxissal, majd írjuk ki ezt a példányt a kezdeti köszöntő szöveg helyett:

```
Dog banan = new Dog
{
    Name = "Banán",
    DateOfBirth = new DateTime(2014, 06, 10)
};
Console.WriteLine(banan);
```

Ezzel kész a kiinduló projektünk.

2.2.2 Implicit típusdeklaráció

A `var` kulcsszó jelentősége: ha a fordító ki tudja találni a kontextusból az értékadás jobb oldalán álló érték típusát, nem szükséges a típus nevét explicit megadnunk, az implicit következik a kódból. Ebben az esetben a típus egyértelműen `Dog`. Ha csak deklarálni szeretnénk egy változót (nem adunk értékül a változónak semmit), akkor nem használhatjuk a `var` kulcsszót, ugyanis nem következik a kódból a változó típusa. Ekkor explicit meg kell adnunk a típust.

```
Dog banan = new Dog
{
    Name = "Banán",
    DateOfBirth = new DateTime(2014, 06, 10)
};
```

```
var watson = new Dog { Name = "Watson" };

var unnamed = new Dog { DateOfBirth = new DateTime(2017, 02, 10) };
var unknown = new Dog { };
//watson = 3; //
//var error;  //

Console.WriteLine(banan);
Console.ReadLine();
```

- Fordítási hiba: a `watson` deklarációjakor eldőlt, hogy ő `Dog` típus, utólag nem lehet megváltoztatni és például számértéket értékül adni. Ez nem JavaScript.
- Fordítási hiba: implicit típust csak úgy lehet deklarálni, ha egyúttal inicializáljuk is. Az inicializációs kifejezés alapján dől el (implicit) a példány típusa.

Próbáljuk ki a nem forduló sorokat, nézzük meg a fordító hibaüzeneteit!

> **Erőss típusoság**
>
> A `var` nem a gyenge típusosság jele a C#-ban, nem úgy, mint pl. JavaScript-ben. Az inicializációs sor után a típus egyértelműen eldől, utána már csak ennek a típusnak megfelelő műveletek végezhetők, például egy értékadással nem változtathatjuk meg a típust.

A `var`-t tipikusan akkor alkalmazzuk, ha:

- hosszú típusneveket nem akarunk kiírni
- feleslegesnek tartjuk az inicializáció mindkét oldalán kiírni ugyanazt a típust
- anonim típusokat használunk (később)

2.2.3 Init-only setter

Az objektum inicializáció működéséhez szükséges a megfelelő láthatóságú setter. Viszont egy ilyen settert nem csak objektum inicializációkor lehet használni, hanem bármikor átállíthatjuk egy példány adatát (mutáció).

Az alábbi példa egy ilyen utólagos módosításra / mutációra.

```
var watson = new Dog { Name = "Watson" };
watson.Name = "Sherlock";
```

Ez így hiba nélkül lefordul.

Kizárólag az inicializációra korlátozhatjuk a setter meghívását az init-only setterrel (`init` kulcsszó).

```
public class Dog
{
    public string Name { get; init; }
    //...
}
```

Ezután az inicializációs sor továbbra is lefordul, de a névátírásos már nem. Ez utóbbi sort kommentezzük ki.

Init-only setter konstruktorból

Init-only settert az osztály konstruktorából is meg lehet hívni - hiszen az is inicializáció.

Használata

Init-only settert több okból kifolyólag is használhatunk, például a típus példányainak immutábilis kezelését akarjuk kikényszeríteni, vagy csak inicializációra akarjuk korlátozni a propertyk beállítását, de nem akarunk ehhez konstruktort írni.

Jelen formájában az init-only setter nem tudja helyettesíteni a kötelező konstruktor paramétert, mert nem kötelező kitölteni ezt a propertyt. Erre a megoldás a C# 11-ben bevezetett `required` kulcsszó a property előtt.

```
public class Dog
{
    public required string Name { get; init;  }
    //...
}
```

Ezzel kötelezővé válik a `Name` kitöltése, ha a `Dog` példányt inicializáljuk.

2.2.4 Indexer operátor, nameof operátor, index inicializáló

A collection initializer analógiájára jött létre az *index initializer* nyelvi elem, ami a korábbihoz hasonlóan sorban hív meg egy operátort, hogy már inicializált objektumot kapjunk vissza. A különbség egyrészt a szintaxis, másrészt az ilyenkor meghívott metódus, ami az index operátor.

operátor felüldefiniálás

Saját típusainkban lehetőségünk van definiálni és felüldefiniálni operátorokat, mint pl. +, -, indexelés, implicit cast, explicit cast, stb.

Tegyük fel, hogy egy kutyához bármilyen, üzleti logikában nem felhasznált információ kerülhet, amire általános struktúrát szeretnénk. Vegyünk fel a `Dog` osztályba egy `string-object` szótárat, amiben bármilyen további információt tárolhatunk! Ezen felül állítsuk be a `Dog` indexerét, hogy az a `Metadata` indexelését végezze:

```
public class Dog
{
    //...
    public Dictionary<string, object>  Metadata { get; } = new();
    public object this[string key]
    {
        get { return Metadata[key]; }
        set { Metadata[key] = value; }
    }
}
```

Konstruktor típus nélkül

A `new` operátor utáni konstruktorhívás sok esetben elhagyható, ha a bal oldal alapján amúgy is tudható a típus.

névtér hivatkozások

Az újabb projektsablonok sokkal kevesebb névtérdeklarációt (`using`) generálnak alapból. Ha kell, vegyük fel a szükségeseket a fel nem oldott néven állva a gyorsművelet (villanykörte) eszközzel (Ctrl + .)

Az objektum inicializáló és az index inicializáló vegyíthető, így az alábbi módon tudunk felvenni további tulajdonságokat a kutyákhoz a legfelső szintű kódba:

```
var pimpedli = new Dog
{
   Name = "Pimpedli",
   DateOfBirth = new DateTime(2006, 06, 10),
   ["Chip azonosító"] = "123125AJ"
};
```

Mivel indexelni általában kollekciókat szokás (tömb, lista, szótár), ezért ezekben az esetekben igen jó eszköz lehet az index inicializáló. Vegyünk fel egy új kutyaszótárt a kutyák kitenyésztése után:

```
var dogs = new Dictionary<string, Dog>
{
   ["banan"] = banan,
   ["watson"] = watson,
   ["unnamed"] = unnamed,
   ["unknown"] = unknown,
   ["pimpedli"] = pimpedli
};

foreach (var dog in dogs)
   Console.WriteLine($"{dog.Key} - {dog.Value}");
```

Próbáljuk ki - minden név-kutya párt ki kell írnia a szótárból.

Elsőre jó ötletnek tűnhet kiváltani a szövegliterálokat a `Name` property használatával.

```
var dogs = new Dictionary<string, Dog>
{
   [banan.Name] = banan,
   [watson.Name] = watson,
   [unnamed.Name] = unnamed,
   [unknown.Name] = unknown,
   [pimpedli.Name] = pimpedli
};
//ArgumentNullException!
```

Ez azonban kivételt okoz, amikor a kutya neve nincs kitöltve, azaz `null` értékű. Esetünkben elég lenne az adott változó neve szövegként. Erre jó a `nameof` operátor.

```
var dogs = new Dictionary<string, Dog>
{
   [nameof(banan)] = banan,
   [nameof(watson)] = watson,
   [nameof(unnamed)] = unnamed,
   [nameof(unknown)] = unknown,
```

```
    [nameof(pimpedli)] = pimpedli
};
```

Ez a változat már nem fog kivételt okozni.

A `nameof` operátor sokfajta nyelvi elemet támogat, vissza tudja adni egy változó, egy típus, egy property vagy egy függvény nevét is.

A szótár feltöltését megírhatjuk kollekció inicializációval is. Ehhez kihasználjuk, hogy a szótár típus rendelkezik egy `Add` metódussal, amelyik egyszerűen egy kulcsot és egy hozzátartozó értéket vár:

```
var dogs = new Dictionary<string, Dog>
{
    { nameof(banan), banan },
    { nameof(watson), watson },
    { nameof(unnamed), unnamed },
    { nameof(unknown), unknown },
    { nameof(pimpedli), pimpedli }
};
```

2.2.5 Using static

Ha egy osztály statikus tagjait vagy egy statikus osztályt szeretnénk használni, lehetőségünk van a `using static` kulcsszavakkal az osztályt bevonni a névfeloldási logikába. Ha a `Console` osztályt referáljuk ilyen módon, lehetőségünk van a rajta levő metódusok meghívására az aktuális kontextusunkban anélkül, hogy az osztály nevét kiírnánk:

```
using System;
using static System.Console;
//..
foreach (var dog in dogs)
    /*Console.*/WriteLine($"{dog.Key} - {dog.Value}");
/*Console.*/WriteLine(banan);
/*Console.*/ReadLine();
```

névfeloldás

Az általános névfeloldási szabály továbbra is él: ha egyértelműen feloldható a hivatkozás, akkor nem szükséges kitenni a megkülönböztető előtagot (itt: osztály), különben igen.

2.2.6 Nullozható típusok

Természetesen a referenciatípusok mind olyan típusok, melyek vehetnek fel `null` értéket, viszont esetenként jó volna, ha a `null` értéket egyébként felvenni nem képes típusok is lehetének ilyen értékűek, ezzel pl. jelezvén, hogy egy érték be van-e állítva vagy sem. Pl. egy szám esetén a 0 egy konkrét, helyes érték lehet a domain modellünkben, a `null` viszont azt jelenthetné, hogy nem vett fel értéket.

Vizsgáljuk meg, hogy a konzolra történő kiíráskor miért lesz az aktuális év **Watson** kutya életkora! Valamelyik `Console.WriteLine` sorhoz vegyünk fel egy töréspontot (`F9`), majd debuggolás közben a **Locals** ablakban (debuggolás közben menu:Debug[Windows > Locals]) figyeljük meg az egyes példányok adatait. Watsont kinyitva láthatjuk, hogy a turpisság abból fakad, hogy a `DateOfBirth` adat típusa, a `DateTime` nem referenciatípus, és alapértelmezés szerinti értéket veszi fel, ami **0001. 01. 01. 00:00:00** - hiszen nem állítottunk be mást.

Ismeretlen születési dátumú, korú egyedek helyes tárolásához az `Age` tulajdonság típusát változtassuk `int?`-re! Az `int?` szintaktikai édesítőszere a `Nullable<int>`-nek, egy olyan struktúrának, ami egy `int` értéket tárol, és tárolja, hogy az be van-e állítva vagy sem. A `Nullable<int>` szignatúráit megmutathatjuk, hogyha a kurzort a típusra helyezve `F12`-t nyomunk.

Módosítsuk a `Dog` `Age` és `DateOfBirth` tulajdonságait is, hogy tudjuk, be vannak-e állítva az értékeik:

```
public class Dog
{
    //...

    public DateTime? DateOfBirth { get; set; }
    private int? AgeInDays => (-DateOfBirth?.Subtract(DateTime.Now))?.Days;
    public int? Age => AgeInDays / 365;
    public int? AgeInDogYears => AgeInDays * 7 / 365;

    //...
}
```

Aritmentikai operátorok

Örvendezzünk, hogy az alap aritmetikai operátorok pont úgy működnek, ahogy szeretnénk (`null` bemenetre `null` eredmény), nem kellett semmilyen trükk.

Az `AgeInDays` akkor ad vissza `null` értéket, ha a `DateOfBirth` maga is `null` volt. Tehát ha nincs megadva születési dátumunk, nem tudunk életkort sem számítani. Ennek kifejezésére használhatjuk a `?.` (Elvis, magyarban **Kozsó** - `null` conditional operator) operátort: a kiértékelendő érték jobb oldalát adja vissza, ha a bal oldal nem `null`, különben `null`-t. A kifejezést meg kellett változtatnunk, hogy a `DateOfBirth`-ből vonjuk ki a jelenlegi dátumot és ezt negáljuk, ugyanis a `null` vizsgálandó érték a bináris operátor bal oldalán kell, hogy elhelyezkedjen.

Elvis operátor

Az Elvis operátor nevének eredetére több magyarázatot is lehet találni, a források annyiban nagyrészt megegyeznek, hogy a kérdőjel tekeredő része az énekes jellegzetes bodorodó hajviseletére emlékeztet, a pontok pedig a szemeket jelölik, így végülis a ?. egy Elvis emotikonként fogható fel. Ezen logika mentén adódik a magyar megfelelő, a Kozsó operátor, hiszen a szem körül tekergőző legikonikusabb hajtincs a magyar zenei kultúrában Kozsó nevéhez köthető.

Ha így futtatjuk az alkalmazást, az `AgeInDays` és a származtatott tulajdonságok értéke `null` (vagy kiírva üres) lesz, ha a születési dátum nincs megadva.

2.2.7 Rekord típus

A rekord típusok speciális típusok, melyek:

- egyenlőségvizsgálat során érték típusokra jellemző logikát követnek, azaz két példány akkor egyenlő, ha adataik egyenlőek
- könnyen immutábilissá tehetők, könnyen kezelhetők immutábilis típusként

A `Dog` típus ezzel szemben jelenleg:

- nem immutábilis, hiszen a születési dátum bármikor módosítható (sima setter)

- egyenlőségvizsgálat során a normál referencia szerinti összehasonlítást követ

Az automatikusan generálódó egyedi azonosítót iktassuk ki a `Dog` osztályból, hogy az adat alapú összehasonlítást könnyebben tesztelhessük.

```
public Guid Id { get; } = Guid.Empty;
```

Vegyünk fel egy logikailag megegyező példányt.

```
var watson = new Dog { Name = "Watson" };
var watson2 = new Dog { Name = watson.Name };
```

Ismét álljunk meg debug során valamelyik `WriteLine` soron. A **Locals** ablakban nézzük meg, hogy a két példány minden adata megegyezik. A **Watch** ablakban (debuggolás közben menu:Debug[Windows > Watch > Watch 1]) értékeljük ki a `watson == watson2` kifejezést. Láthatjuk, hogy ez az egyenlőségvizsgálat hamist ad, ami technikailag helyes, mert két különböző memóriaterületről van szó, a referenciák nem ugyanoda mutatnak a memóriában. Sok esetben azonban nem ezt szeretnénk, hanem például a dupla rögzítés elkerülésére az adatok alapján történő összehasonlítást, ami érték típusoknál van. Referencia típusoknál klasszikusan ezt a `GetHashCode`, `Equals` függvények felüldefiniálásával értük el (vagy az `IComparable<T>`, `IComparer<T>` interfészre épülő logikákkal). Egy újabb lehetőség a rekord típus használata.

Pozíció alapú megadás

Vegyünk fel a `Dog` típus adatainak megfelelő rekord típust, mindössze egy kifejezésként. A `Dog` típus alá:

```
public record class DogRec(
    Guid Id,
    string Name,
    DateTime? DateOfBirth=null,
    Dictionary<string, object> Metadata=null
);
```

> **Note**
>
> A `record class` jelölőből a `class` elhagyható.

Ez az ún. pozíció alapú megadási forma, ami a leginkább rövidített megadási formája a rekord típusnak. Ebből a rövid formából, mindenfajta extra kód írása nélkül a fordító számos dolgot generál:

- a zárójelen belüli felsorolásból konstruktort és dekonstruktort
- a zárójelen belüli felsorolás alapján propertyket `get` és `init` tagfüggvényekkel
- alapértelmezett logikát az érték szerinti összehasonlításhoz
- klónozó és másoló konstruktor logikákat
- alapértelmezett formázott kiírást, szöveges reprezentációt (`ToString` implementációt)

Így egy könnyen kezelhető, immutábilis, az összehasonlításokban érték típusként viselkedő adatosztályunk lesz.

> **Warning**
>
> Az `Id` -nek nem tudjuk beállítani ebben a formában az alapértelmezett `Guid.Empty` értéket vagy a `Metadata` -nak az új példányt, mert az egyenlőségjeles kifejezésekből alapértelmezett konstruktorparaméter-értékek lesznek, amik csak statikus, fordítási időben kiértékelhető kifejezések lehetnek.

Vegyünk fel a többi Watson példány mellé két újabbat, de itt már az új rekord típusunkat használjuk.

```
var watson3 = new DogRec(Guid.Empty, "Watson");
var watson4 = new DogRec(Guid.Empty, "Watson");
```

A fentebbi **Watch** ablakos módszerrel ellenőrizzük a `watson3 == watson4` kifejezés értékét. Ez már igaz érték lesz az adatmező alapú összehasonlítási logika miatt.

Próbáljuk ki ugyanezt a kiértékelést az alábbi változattal:

```
var watson3 = new DogRec(Guid.Empty, "Watson");
var watson4 = new DogRec(Guid.Empty, "Watson", DateTime.Now.AddYears(-1));
```

Ez hamis értéket ad, az egyenlőségnek minden mezőre teljesülnie kell, nem csak a mindkettőben kitöltöttekre.

A `DogRec` típus alapvetően immutábilis, a példányainak alapadatai inicializálás után nem módosíthatók. Próbáljuk felülírni a nevet.

```
var watson3 = new DogRec(Guid.Empty, "Watson");
var watson4 = new DogRec(Guid.Empty, "Watson", DateTime.Now.AddYears(-1));
watson4.Name = watson3.Name + "_2"; //<= nem fordul
```

Nem fog lefordulni, mert minden property init-only típusú. A sor jobboldala egyébként lefordulna, tehát a lekérdezés (getter hívás) működne.

Ha immutábilis típusokkal dolgozunk, akkor mutáció helyett új példányt hozunk létre megváltoztatott adatokkal. Alapvetően ezt az OO nyelvekben másoló konstruktorral oldjuk meg. A rekord típusnál ennél is továbbmenve másoló kifejezést használhatunk.

```
var watson4 = new DogRec(Guid.Empty, "Watson", DateTime.Now.AddYears(-1));
var watson5 = watson4 with { Name = "Sherlock" };
WriteLine(watson4);
WriteLine(watson5);
```

Futtatáskor a konzolban gyönyörködjünk a rekord típusok alapértelmezetten is olvasható szöveges kiírásában.

A másoló kifejezésben a `with` operátor előtt megadjuk, melyik példányt klónoznánk, majd az **inicializáció részeként** milyen értékeket állítanánk át, ehhez az objektum inicializációs szintaxist használhatjuk. Fontos eszünkbe vésni, hogy a másolás eredményeként új példány jön létre, új memóriaterület foglalódik le. Gondoljunk erre akkor, amikor egy ciklusban használjuk ezt a módszert sok egymást követő módosításra.

Mire is jó a rekord típus

Mire jó a rekord típus, az immutabilitás? Az immutábilis típussokkal való hatékony és eredményes munka másfajta, az imperatív nyelvekhez szokott fejlesztők számára szokatlan módszereket kíván. Vannak területek, ahol ez a befektetés megtérül, ilyen például a többszálú környezet. A legtöbb szálkezeléssel kapcsolatos probléma ugyanis a szálak által közösen használt adatstruktúrák mutációjára vezethető vissza (ún. *race condition*, versenyhelyzet). Nincs mutáció - nincs probléma. (*No mutation - no cry*)

Kitérő: a szótár visszavág

A rekord típus által biztosított kellemes tulajdonságok csak akkor érvényesek, ha nem keverjük hagyományos referencia típusokkal.

A szokásos módszerrel ellenőrizzük le, hogy a `watson5 == watson6` kifejezés igaz-e. Igen, hiszen minden kitöltött adatuk egyezik.

```
var watson4 = new DogRec(Guid.Empty, "Watson", DateTime.Now.AddYears(-1));
var watson5 = watson4 with { Name = "Sherlock" };
var watson6 = watson4 with { Name = "Sherlock" };
WriteLine(watson4);
WriteLine(watson5);
WriteLine(watson6);
```

Vigyünk be egy ártatlan inicializációt a `Metadata` propertyre.

```
var watson4 = new DogRec(Guid.Empty, "Watson", DateTime.Now.AddYears(-1));
var watson5 = watson4 with { Name = "Sherlock", Metadata = new Dictionary<string, object>() };
var watson6 = watson4 with { Name = "Sherlock", Metadata= new Dictionary<string, object>() };
WriteLine(watson4);
WriteLine(watson5);
WriteLine(watson6);
```

Ezzel eléggé illogikus módon hamisra változik a `watson5 == watson6` kifejezés. Az oka az, hogy a `Metadata` szótár egy klasszikus referencia típus, az összehasonlításnál a klasszikus memóriacím-összehasonlítás történik, viszont az a két új szótár példány esetében eltérő lesz. A formázott szöveges kiírásba is belerondít a szótár, mert ott is a szótár típus alapértelmezett szöveges reprezentációja jut érvényre, ami a típus neve.

Klónozzunk tovább, aztán próbáljunk mutációt végrehajtani a `Metadata` szótáron.

```
var watson6 = watson4 with { Name = "Sherlock", Metadata = new Dictionary<string, object>() };
var watson7 = watson6 with { Name = "Watson" };
watson7.Metadata.Add("Chip azonosító", "12345QQ");
WriteLine(watson4);
```

Ez lefordul, pedig ez mutáció. A **Locals** ablakban figyeljük meg a `watson6` és `watson7` szótárait: **mindkettőbe** bekerült a chip azonosító. Ez az ún. *shallow copy* jelenség, amikor nem a szótár memóriaterülete klónozódik, csak a rá mutató referencia, ami azt eredményezi, hogy a két példánynak közös szótára lesz.

Összességében az adatstruktúránkban megjelenő klasszikus referencia típus elrontja:

- az immutabilitást
- az érték szerinti összehasonlítást
- a formázott szöveges megjelenést
- a klónozást

Immutabilitás

Immutábilis környezetben törekedjünk arra, hogy a **teljes** adatstruktúránk támogassa az immutábilis kezelést.

Normál megadás

Ha nincs szükségünk a kikényszerített immutabilitásra, akkor használhatjuk a rekord normál megadását. Fogjuk a `Dog` osztályt, másoljuk le a kódját, adjunk neki más nevet és `class` helyett `record` jelölőt.

A `Dog` osztály fölé:

```
public record DogRecExt
{
    public string Name { get; init; }
    public Guid Id { get; } = Guid.Empty;
    public DateTime? DateOfBirth { get; set; }
    public Dictionary<string, object> Metadata { get; } = new();

    private int? AgeInDays => (-DateOfBirth?.Subtract(DateTime.Now))?.Days;
    public int? Age => AgeInDays / 365;
    public int? AgeInDogYears => AgeInDays * 7 / 365;

    public object this[string key]
    {
        get { return Metadata[key]; }
        set { Metadata[key] = value; }
    }
}
```

ToString

A `ToString` implementációját elhagytuk az előző szakaszban említettek miatt.

A `Program.cs`-be:

```
var watson8 = new DogRecExt { Name = "Watson" };
watson8.DateOfBirth = DateTime.Now.AddYears(-15);
var watson9 = watson8 with { };
WriteLine(watson8);
WriteLine(watson9);
```

Ellenőrizzük le a rekord tulajdonságokat:

- A konzol kimeneten a formázást, továbbá a mutáció működését, azaz a `watson8` születési dátuma a beállított lesz.
- Ez nem csoda, hiszen a property deklarációban engedtük a mutációt.
- A konzol kimeneten megfigyelt példányadatokon a klónozó kifejezés működését. Semmi különös, ugyanúgy működik, mint a tömör formánál.
- A **Watch** ablakban `watson8 == watson9` egyenlőséget. Ez igaz, mert minden adattagjuk egyezik.

record struct

A rekordoknak további válfajai vannak, ugyanis struktúra is lehet rekord, ilyenkor a `record struct` kulcsszó párt használjuk a típus deklarációjánál. Sőt, a `readonly record struct` egy immutábilis `record struct`. Ezen válfajok nyilván különbözőképpen viselkednek, mely viselkedéseket itt most nem részletezzük, de a dokumentációban megtalálhatók.

2.3 LINQ

2.3.1 Előkészítés

A gyakorlat kezdetén klónozzuk le a kiinduló projektet az alábbi paranccsal:

```
git clone https://github.com/bmeviauav23/Linq-lab.git
```

Nyissuk meg Visual Studio-ban a *HelloLinq.sln* solution fájlt.

Megnyitás után tekintsük át a kiinduló projektben levő fájlokat:

- **Program.cs**: a legfelső szintű kódot tartalmazó osztály. Található benne egy `Dogs` változó, ami a `Dog` osztály statikus `Repository` tulajdonságába hív át.
- **Dog.cs**: a korábbi gyakorlatokon használt adatmodell (apróbb módosításokkal).

  Bekerült egy `Siblings` tulajdonság, a `ToString` pedig kiírja a kutyához tartozó testvérek számát is (ehhez a `TrimPad` bővítő metódust használja).

  A statikus `Repository` tulajdonság mögött egy lustán inicializált `Lazy<T> RepositoryHolder` található, ami egy megfelelően formázott bemeneti CSV fájlból elkészíti számunkra az adatmodellt, amivel a későbbiekben dolgozunk. Ennek implementációját elég a gyakorlat végén megnézni. Az `Import` és `Export` függvények a kutyák sorosítását végzik el mindkét irányban.
- **Extensions/StringExtensions.cs**: ez az osztály tartalmaz egy segédmetódust a formázott kiíráshoz. A `Dog` `ToString` metódusa használja fel. A bővítő metódusos részben lesz jelentősége.
- **dogs.csv**: egy pontosvesszővel tagolt adathalmaz, amelyben 100 darab előre felvett kutya adata található. Innen puskázhatunk, ha ellenőrizni akarjuk, hogy helyesek-e a programunk eredményei.

A kiinduló projektben a globális implicit névtérhivatkozások ki vannak kapcsolva. A **csproj** fájlban megnézhetjük (menu:jobb klikk a projekten[Edit Project File]):

```
<ImplicitUsings>disable</ImplicitUsings>
```

2.3.2 Lambda kifejezések, delegátok

Gyakori feladat, hogy objektumok kollekciójával kell dolgoznunk. Képesek vagyunk olyan jellegű segédfüggvényeket készíteni, amik például egy kollekcióban kikeresik az összes olyan elemet, amely egy megadott feltételnek eleget tesz.

A `Program.cs` fájlban látható ennek a kezdeti naiv változata, szemrevételezzük:

```
static List<Dog> ListDogsByNamePrefix(IEnumerable<Dog> dogs, string prefix)
{
    var result = new List<Dog>();
    foreach (var dog in dogs)
    {
        if (dog.Name.StartsWith(prefix, StringComparison.OrdinalIgnoreCase))
        {
            result.Add(dog);
        }
    }
    return result;
}
```

Próbáljuk ki!

A kód működik, viszont nem újrahasznosítható. Ha bármi más alapján szeretnénk keresni a kutyák között (pl. a neve tartalmaz-e egy adott szövegrészt), mindig egy új segédfüggvényt kell készítenünk, ami rontja a kód újrahasznosíthatóságát.

Oldjuk meg úgy, hogy az általános problémát is megoldjuk! Ehhez az szükséges, hogy a kollekciónk egyes elemein kiértékelhessünk egy, a hívó által megadott predikátumot. Készítsük el az általánosabb változatot, ehhez felhasználhatjuk a `ListDogsByNamePrefix` kódját.

```
static List<Dog> ListDogsByPredicate(
    IEnumerable<Dog> dogs, Predicate<Dog> predicate)
{
    var result = new List<Dog>();
    foreach (var dog in dogs)
    {
        if (predicate(dog))
        {
            result.Add(dog);
        }
    }
    return result;
}
```

A legfelső szintű kódban így hívhatjuk meg (felhasználhatjuk az eredeti ciklust):

```
foreach(var dog in ListDogsByPredicate(Dogs, delegate (Dog d)
    {
        return d.Name.StartsWith(searchText, StringComparison.OrdinalIgnoreCase);
    })
)
{
    Console.WriteLine(dog);
}
```

Egy egy bemenő paraméterű és egy logikai (`bool`) értéket visszadó függvényt definiálunk helyben (*inline*) és ezt (illetve a referenciáját) adjuk át. Használjunk inkább lambda kifejezést, az jóval rövidebben leírható - egyelőre csak nézzük meg, de ne integráljuk a kódba:

```
d => d.Name.StartsWith(searchText, StringComparison.OrdinalIgnoreCase);
```

Lambda kifejezések szintaktikája

Lambda kifejezéssel az egyetlen kifejezésből álló függvényeket adhatjuk meg nagyon kompakt módon. A `=>` -tól balra elnevezzük a bemenő paramétereket, jobbra pedig felhasználhatjuk. A `return` , `{}` és egyéb sallangokat elhagyhatjuk.

Vessük össze, hogy az első esetben explicit megadtuk, hogy a bemenő paraméterünk `Dog` , most viszont nem. Ezt a fordító statikus kódanalízis alapján el tudja dönteni: a `d` változónk nem lehet más, csak `Dog` (statikus) típusú, ezért csak így használhatjuk, viszont nem kell kiírnunk a típust.

A lambda kifejezések egy lehetséges módja a delegátok leírásának. A delegát kódot reprezentál, viszont a kódot kezelhetjük adatként is.

Próbáljuk meg a delegátunkat kivenni egy implicit típusú változóba a ciklus előtt:

```
// fordítási hiba!
var predicate =
```

```
    d => d.Name.StartsWith(searchText, StringComparison.OrdinalIgnoreCase);
foreach (var dog in ListDogsByPredicate(Dogs, predicate))
{
    Console.WriteLine(dog);
}
```

Fordítási hibát kapunk, lambda kifejezés típusa nem lehet implicit eldönthető az inicializációs sorban: sem a bemenő paraméter pontos típusát nem tudjuk (`Dog` ? `Puppy` ?), sem a visszatérési értéket (`bool` ? `object` ? `void` ?). Tehát explicit meg kell adnunk a típust:

```
Predicate<Dog> predicate =
    d => d.Name.StartsWith(searchText, StringComparison.OrdinalIgnoreCase);
```

Ezután fordul és fut is az alkalmazásunk.

Predicate beépített deleagate típus

Ehhez tudnunk kellett, hogy a `Predicate<T>` megfelelő szignatúrájú. Mutassuk meg ezen típus dokumentációját vagy tegyük a kurzort a típusra és nyomjunk `F12`-t.

2.3.3 `Func<>`, `Action<>`

Ismerkedjünk meg a `Func` és `Action` általános delegáttípusokkal. Ezzel a két generikus típussal (pontosabban a változataikkal) gyakorlatilag az összes gyakorlatban előforduló függvényszignatúrát le lehet fedni. Például a fenti szűrőlogikát is átírhatnánk erre:

```
Func<Dog, bool> dogFunc =
    d => d.Name.StartsWith(searchText, StringComparison.OrdinalIgnoreCase);
```

A `dogFunc` és a `predicate` kompatibilisnek tűnhetnek (elvégre a jobboldaluk ugyanaz), ám ha lecserélnénk pl. a `ListDogsByPredicate(Dogs, predicate)` hívásban a `predicate`-et `dogFunc`-ra, a kód nem fordulna, ugyanis a két delegáttípus nem kompatibilis.

Az `Action<>` hasonló elven működik, visszatérési érték nélküli függvényekre.

Egyéb delegate típusok

Ha minden esetre jók, miért vannak használatban `Action<>` és `Func<>`-on kívül más delegáttípusok? Egyrészt történelmi okok miatt. Később jelentek meg, mint a specifikusak, például a `Predicate<T>`. Másrészt a specifikusabbak a nevükkel kifejezőbbek lehetnek.

Tiszta függvények

A fenti predikátumváltozataink mind nem tiszta függvények (pure function), ugyanis olyan adattól is függ a visszatérési értéke, ami nem szerepel a paraméterlistáján - ez esetünkben a `searchText` változó. A kódunk azért működik, mert a delegát megadásakor a `searchText` aktuális értékét elkapjuk (capture), belerakjuk a függvénylogikába.

Próbáljuk a `dogFunc`-ot `var`-ként deklarálni.

```
//Fordítási hiba!
var dogFunc =
   d => d.Name.StartsWith(searchText, StringComparison.OrdinalIgnoreCase);
```

A fordító nem tudja meghatározni a `d` paraméter típusát, ezért kapjuk a fordítási hibát. Adjuk meg explicit a paraméter típusát.

```
var dogFunc =
   (Dog d) => d.Name.StartsWith(searchText, StringComparison.OrdinalIgnoreCase);
```

Debugger-rel ellenőrizhetjük, hogy a `dogFunc` valódi típusa `Func<Dog, bool>` lesz.

2.3.4 IEnumerable\<T> bővítő metódusok

Vigyük tovább az általánosítást. Írjunk olyan logikákat, mely nem csak kutyák listájára, hanem bármilyen felsorolható (enumerálható) kollekcióra működik. Írjunk `IEnumerable<T>` típuson működő segédfüggvényeket.

Hozzunk létre egy `EnumerableExtensions` (I betű nélkül, az ugyanis interfészre utal) nevű fájlt az `Extensions` mappában! Elsőként valósítsuk meg az összegző logikát.

```
namespace HelloLinq.Extensions.Enumerable;

public static class EnumerableExtensions
{
   public static int Sum<T> (IEnumerable<T>  source, Func<T, int>  sumSelector)
   {
      var result = 0;
      foreach (var elem in source)
      {
         result += sumSelector(elem);
      }
      return result;
   }
}
```

Hívjuk meg a legfelső szintű kódból.

```
using HelloLinq.Extensions.Enumerable;

IEnumerable<Dog> Dogs = Dog.Repository.Values;
foreach (var dog in Dogs)
{
   Console.WriteLine(dog);
}

Console.WriteLine(
   "Életkorok összege: "
   + $"{EnumerableExtensions.Sum(Dogs, d => d.Age ?? 0)}");

string searchText;
```

A segédfüggvények hátránya, hogy ismernünk kell a segédosztály nevét. Továbbá jobb lenne, ha a kollekción közvetlenül hívhatnánk az összegző függvényt. Erre megoldás a bővítő metódus.

A bővítő metódusok:

- statikus osztályban definiálhatók
- statikus függvények
- első paramétere előtt `this` jelöli, hogy melyik típust bővítik

Az első paraméter elé tegyük be a `this` jelölőt.

```
public static int Sum<T> (
    this IEnumerable<T>  source,
    Func<T, int>  sumSelector)
    {
        // ...
    }
```

Most már használhatjuk azt a szintaxist, mintha a kollekciónak eleve lenne összegző függvénye:

```
Console.WriteLine($"Életkorok összege: {Dogs.Sum(d => d.Age ?? 0)}");
```

OO egységbezárási elv

A bővítő metódusok semmilyen módon nem bontják meg a típusok egységbezárási képességeit. A függvények implementációi a bővítendő típusok kívülről is elérhető függvényeit, propertyjeit használhatják, privát adattagokhoz, függvényekhez nem férnek hozzá.

Osztály nevének feloldása

A bővítő metódusok alkalmazásakor nagyon fontos, hogy bár a bővítő metódus osztályának nevét nem írjuk ki, az osztály nevének feloldhatónak kell lennie, azaz az osztály névterét `using` direktívával be kell hivatkoznunk. Egy próba erejéig kommentezzük ki a `using HelloLinq.Extensions.Enumerable;` sort és ellenőrizzük, hogy nem fordul a kódunk, a bővítő metódus nevét a fordító nem tudja feloldani.

Gyakorlásképpen írhatunk további gyakori adatfeldolgozási műveletekre függvényeket, mint amilyen az átlagszámítás, min-max keresés.

Megoldás

```
public static class EnumerableExtensions
{
  //...

  public static double Average<T>(this IEnumerable<T> source, Func<T, int> sumSelector)
  {
    var result = 0.0; // Az osztás művelet miatt double
    var elements = 0;
    foreach (var elem in source)
    {
      elements++;
      result += sumSelector(elem);
    }
    return result / elements;
  }

  public static int Min<T>(this IEnumerable<T> source, Func<T, int> valueSelector)
  {
    int value = int.MaxValue;
    foreach (var elem in source)
    {
      var currentValue = valueSelector(elem);
      if (currentValue < value)
      {
        value = currentValue;
      }
    }
    return value;
  }

  public static int Max<T> (this IEnumerable<T> source, Func<T, int> valueSelector)
    => -source.Min(e => -valueSelector(e));
}
```

Próbáljuk ki az új függvényeket. Mivel a `Dogs` típusa `IEnumerable<Dog>`, így a bővítő metódusok bővítendő típusa illeszkedik rá.

```
Console.WriteLine($"Átlagos életkor: {Dogs.Average(d => d.Age ?? 0)}");
Console.WriteLine(
  $"Minimum-maximum életkor: " +
  $"{Dogs.Min(d => d.Age ?? 0)} | {Dogs.Max(d => d.Age ?? 0)}");
```

StringExtensions.cs

A `StringExtensions` osztályban egy lambdaként megvalósított bővítő metódust láthatunk, ami egy szöveget adott hosszra (szélességre) egészít ki szóközökkel. A függvényt a `Dog` `ToString` metódusa használja fel.

2.3.5 Gyakori lekérdező műveletek, yield return

Gyakran előfordul, hogy egy listát szűrni vagy projektálni szeretnénk. Írjunk saját generátort ezekhez a műveletekhez is az `EnumerableExtensions`-be:

```
public static IEnumerable<T> Where<T>(
  this IEnumerable<T>  source, Predicate<T>  predicate)
{
  foreach (var elem in source)
  {
    if (predicate(elem))
    {
      yield return elem;
```

```
        }
    }
}

public static IEnumerable<TValue> Select<T, TValue>(
    this IEnumerable<T>  source, Func<T, TValue> selector)
{
    foreach (var elem in source)
    {
        yield return selector(elem);
    }
}
```

Próbáljuk ki a legfelső szintű kód elején, válasszuk ki a 2010 előtt született kutyák nevét és korát egy stringbe:

```
foreach (var text in Dogs
    .Where(d => d.DateOfBirth?.Year < 2010)
    .Select(d => $"{d.Name} ({d.Age})"))
{
    Console.WriteLine(text);
}
```

A yield return haszna

A `yield return` egy hasznos eszköz, ha IEnumerable-t kell produkálnunk visszatérési értékként. Segítségével mindig csak akkor állítjuk elő a következő elemet, amikor a hívó kéri. A működését debuggerrel is figyeljük meg: tegyünk breakpointot a két `yield return` sorra, majd F10-zel kövessük végig, ahogy a `foreach` elkéri a `Select`-től a következő elemet, ami emiatt elkéri a `Where`-től, majd újraindul a ciklus. A hívások állapotgépként működnek, a következő meghíváskor onnan folytatódnak, ahonnan az előző `yield return`-nél kiléptünk.

Nem nagy meglepetés, hogy az általunk megírt `Sum`, `Average` (melyek egyedi visszatérésűek), `Select` és `Where` (amik szekvenciális visszatérésűek, generátorok) metódusok mind a .NET keretrendszer részét képezik (a `System.Linq.Enumerable` statikus osztályban definiált bővítő metódusok). A **LINQ** — **L**anguage **IN**tegrated **Q**uery — ezeket a műveleteket teszi lehetővé `IEnumerable` interfészt megvalósító objektumokon. A LINQ függvények bővítő metódusként lettek hozzáadva meglevő funkcionalitáshoz (kollekciókhoz, lekérdezésekhez), sőt, külső library-k is adnak saját LINQ bővítő metódusokat.

Cseréljük le a `Program.cs`-ben a `using HelloLinq.Extensions.Enumerable` hivatkozást `using System.Linq`-re: az általunk megírt kód továbbra is ugyanazt az eredményt produkálja! Nézzük meg, hogy hol vannak definiálva ezek a függvények a keretrendszeren belül: a kurzort tegyük a kódban oda, ahol valamelyik korábban megírt függvényünket hívnánk, majd nyomjunk F12-t. Próbáljuk ki, hogy továbbra is az elvárt módon működik-e a programunk.

Implicit usings

A névtércsere helyett bekapcsolhatjuk a globális implicit névtér funkciót, mert a `System.Linq` névtér is egy implicit hivatkozott névtér. Ehhez a projektfájlban az `<ImplicitUsings>disable</ImplicitUsings>` beállítást írjuk át `enable`-re, majd a `using HelloLinq` -en kívül minden névtérhivatkozást töröljünk a `Program.cs`-ből.

2.3.6 Anonim típusok

Lekérdezéseknél gyakran használatosak az anonim típusok, amelyeket jellemzően lekérdezések eredményének ideiglenes, típusos tárolására használunk. Az anonim típusokkal lehetőségünk van *inline* definiálni olyan osztályokat,

amelyek jellemzően csak dobozolásra és adattovábbításra használtak. Vegyük az alábbi példákat a legfelső szintű kód elején:

```
var dolog1 = new { Name = "Alma", Weight = 100, Size = 10 };
var dolog2 = new { Name = "Körte", Weight = 90 };
```

Korábban már említettük a `var` kulcsszót, amellyel implicit típusú, lokális változók definiálhatók. Az értékadás jobb oldalán definiálunk egy-egy anonim típust, amelynek felveszünk néhány tulajdonságot. A tulajdonságok mind típusosak maradnak, a típusrendszerünk továbbra is sértetlen. Az implicit statikus típusosság nem csak a `var` kulcsszóban jelenik meg tehát, hanem az egyes tulajdonságok típusában is.

Az anonim típusok:

- csak referencia típusúak lehetnek (objektumok, nem pedig struktúrák),
- csak publikusan látható, csak olvasható tulajdonságokat tartalmazhatnak,
- eseményeket és metódusokat nem tartalmazhatnak (delegate példányokat tulajdonságban viszont igen),
- szerelvényen belül láthatók (`internal`) és nem származhat belőlük másik típus (`sealed`)
- típusnevét nem ismerjük, így hivatkozni sem tudunk rá, csak a `var` -t tudjuk használni
- nem használhatók ott, ahol a `var` típus se használható, többek között nem adhatjuk át függvénynek és nem lehet visszatérési érték sem

Ha az egeret a `var` kulcsszavak vagy egyes tulajdonságnevek fölé visszük, láthatjuk, hogy valóban fordítási idejű típusokról van szó.

> **IntelliSense**
>
> Figyeljük meg, hogy az **IntelliSense** is működik ezekre a típusokra, felkínálja a típus property-jeit.

A fordító újra is hasznosítja az egyes típusokat:

```
var dolgok = new { Name = "Gyümölcsök", Contents = new[] { dolog1, dolog2 } };
```

A `Contents` tulajdonság típusa a fenti anonim objektumaink tömbje, ezért nem is adhatnánk meg másképpen (nem tudjuk a nevét, amivel hivatkozhatunk rá). A fordító most panaszkodik, ugyanis a két dolog típusa nem implicit következtethető. Ha felvesszük a `Size` tulajdonságot a `dolog2` definíciójába, máris fordul.

```
var dolog2 = new { Name = "Körte", Weight = 90, Size = 12 };
```

Ha végeztünk az anonim típusokkal való ismerkedéssel, az ezekkel kapcsolatos kódsorokat kikommentezhetjük.

2.3.7 LINQ szintaxisok

Az előző részben ismertetett jellegű lekérdezések nagyban hasonlítanak azokhoz, amiket adatbázis-lekérdezésekben alkalmazunk. A különbség itt az, hogy imperatív szintaxist használunk, szemben pl. az SQL-lel, ami deklaratívat. Ezért is van jelen a C# nyelvben az ún. *query syntax*, amely jóval hasonlatosabb az SQL szintaxisához, így az adatbázisokban jártas fejlesztők is könnyebben írhatnak lekérdezéseket. Ugyanakkor nem minden lekérdezést tudunk query syntax-szal leírni.

Miért nem lehet mindent megírni query syntaxban?

Ennek oka, hogy az operátorok bevezetése egy nyelvben elég drága - le kell például foglalni az operátor nevét, amit utána korlátozottan lehet csak használni másra. Ezért sem csinálták meg minden LINQ függvénynek az operátor párját, csak az SQL-ben gyakrabban használatosabbaknak.

Az előzőhöz hasonló lekérdezést megírhatunk az alábbi módon query syntax használatával:

```
using HelloLinq.Extensions;

//...
IEnumerable<Dog> Dogs = Dog.Repository.Values;

var query = from d in Dogs
        where d.DateOfBirth?.Year < 2010
        select new
        {
           Dog = d,
           AverageSiblingAge = d.Siblings.Average(s => s.Age ?? 0)
        };
foreach (var meta in query)
{
   Console.WriteLine(
      $"{meta.Dog.Name} - {meta.AverageSiblingAge}");
}
```

A query szintaxis végül a korábban is használt, ún. *fluent szintaxis*sá fordul. Ennek igazolására nézzük meg F12-vel, hogy hol vannak definiálva az újonnan megismert operátorok (`select` , `where`). A két szintaxist szokás ötvözni is, jellemzően akkor, ha query szintaxisban írjuk a lekérdezést, és a hiányzó funkcionalitást fluent szintaxissal pótoljuk.

Fluent szintaxis

A fluent szintaxist olyan kialakítású API-knál alkalmazhatjuk, ahol a függvények a tartalmazó típust várják (egyik) bemenetként és azonos (vagy leszármazott) típust adnak vissza. A LINQ-nél ez a típus az `IEnumerable<>` .

Ezen az órán memóriabeli adatforrásokkal dolgoztunk (konkrétan a `Dogs` nevű `Dictionary<,>` típusú változóval), a LINQ operátorok közül a memóriabeli listákon dolgozókat használtuk, melyeket az `IEnumerable<>` interfészre biggyesztettek rá bővítő metódusként. Ezt a LINQ API-t teljes nevén *LINQ-to-Objects*nek hívják, de gyakran csak LINQ-ként hivatkozzák.

2.3.8 Kitekintő: Expression\<>, LINQ providerek

Vegyük az alábbi nagyon egyszerű delegate-et és ennek `Expression<>` -ös párját.

```
Func<int, int>  f = x => x + 1;
Expression<Func<int, int>> e = x => x + 1;
```

Nézzük meg debuggolás közben a **Watch** ablakban a fenti két változót. Az `f` egy delegate, lefordított *kód*ra mutató referencia, az *`Expression`* *a jobb oldali kifejezésből épített (fa struktúrájú) *adat*.

A fát kóddá fordíthatjuk a `Compile` metódus segítségével, mely a lefordított függvény referenciáját (delegát példány) adja vissza, amit a függvényhívás szintaxissal hívhatunk meg. Ebből áll össze az alábbi fura kinézetű kifejezés:

```
Console.WriteLine(e.Compile()(5));
```

Bár az `Expression<>` emiatt okosabb választásnak tűnik, ám a LINQ-to-Objects alapinterfészének (ami a lekérdezőfüggvényeket biztosítja) függvényei `Func<>` / `Action<>` delegátokat várnak. Ami nem csoda, hiszen memóriabeli listákat általában sima programkóddal dolgozunk fel, nincs értelme felépíteni kifejezésfát csak azért, hogy utána egyből kóddá fordítsuk. Emellett más, memóriabeli adatokon dolgozó LINQ technológia is létezik, pl. LINQ-to-XML saját API-val (nem `IEnumerable<>` alaptípussal).

A nem memóriabeli adatokon, hanem például külső adatbázisból dolgozó LINQ provider-ek viszont `IQueryable<>`-t valósítanak meg. Az `IQueryable<>` az `IEnumerable<>`-ból származik, így neki is vannak `Func<>` / `Action<>`-ös függvényei, de emellett `Expression<>`-ösek is. Ez utóbbiak teszik lehetővé, hogy ne csak .NET kódot generáljanak a lambda kifejezésekből, hanem helyette pl. SQL kifejezést - hiszen egy relációs adatbázis adatfeldolgozó nyelve nem .NET, hanem valamilyen SQL dialektus.

A LINQ providerek általános működése

Bemenetük: query függvényeknek (`IQueryable<>` vagy `IEnumerable<>` függvényei vagy pl. `XDocument`) paraméterül adott lambdák (`Func<>` vagy `Expression<>`)

Kimenetük: az adatforrásnak megfelelő nyelvű, a query-t végrehajtó kód (.NET kód vagy SQL).

LINQ-to-Objects esetén nincs valódi LINQ provider (a provider az `IQueryable.Provider`-en keresztül érhető el, de a `List<>` nem `IQueryable`!), hiszen nincs feladata: kódot kap bemenetül, ugyanazt kellene kimenetül adnia. A *LINQ-to-XML* is hasonló elven működik.

Valódi LINQ providert valósít meg például az *Entity Framework*, de ezt a technológiát később tárgyaljuk.

2.4 C# alapok IV.

Ezen a gyakorlaton több különféle nyelvi konstrukciót tekintünk át, vegyesfelvágott jelleggel. Az egyes fő témaköröket külön projektként dolgozzuk ki. A projekteket hozzáadhatjuk az elsőként létrehozott projekt solutionjéhez (menu:jobbklikk a solution-ön[Add > New project]). Hozzáadás után ne felejtsük el átállítani a futtatandó projektet: menu:jobbklikk a projekten[Set as Startup Project].

2.4.1 Bejárási problémák

Enumerátorok használata esetén két alapvető problémába ütközünk: az egyik a mögöttes kollekció módosulása bejárás során, a másik pedig a késleltetett kiértékelésből adódó mellékhatások kezelése.

Kollekció módosulása bejárása során

Szűrjünk le egy számokat tartalmazó kollekciót csak azokra az elemekre, amik megfelelnek egy feltételnek, és ezeket távolítsuk el a kollekcióból!

```
var numbers = Enumerable.Range(1, 8).ToList();
foreach (var p in numbers)
{
   if (p % 2 == 0)
   {
      numbers.Remove(p);
   }
}
numbers.ForEach(Console.WriteLine);
```

Futtatáskor kivételt kapunk. Mi a probléma? A kollekciót bejárás közben szerettük volna módosítani, viszont ez könnyen nem várt működést (túlcímzést, nemdeterminisztikus bejárást) tenne lehetővé, ezért kivételt kapunk. Oldjuk meg a problémát: nem módosíthatjuk a forrás objektumot bejárás közben, tehát ne azt a kollekciót járjuk be, másoljuk le!

```
foreach (var p in numbers.ToList()) // a ToList bekerült
{
   // ...
}
```

Ez megoldja a problémát, sikerül eltávolítani az elemeket a kollekcióból. De miért? A `ToList` `IEnumerable` bővítő, tehát bejárhatja a kollekciót, ezután pedig egy **másik** `List<>` objektumban tárolja az elemeket. Így tehát két listánk lesz (a `numbers` és a `numbers.ToList` visszatérési értéke), amik kezdetben egymás klónjai, menet közben az egyikből veszünk ki, a másikon pedig iterálunk.

Kivételek

Bár a fenti az általános szabály, bizonyos kollekciók bizonyos módosító műveletei mégsem dobnak kivételt, ilyen például a `Dictionary<,>` `Remove` és `Clear` műveletei.

Azonnali és késleltetett kiértékelés

Amennyiben egy metódus generátor (`IEnumerable` vagy `IEnumerable<>` visszatérési értékű), az egyes elemeken történő iteráció a generátorok egymásba ágyazását jelenti, azaz az egyes generátorokban a `yield return` által visszaadott értéket fogja az enumerátor `MoveNext` metódusa visszaadni. Amíg az `IEnumerable` -re van referenciánk, és nem járjuk azt közvetlenül be, addig *késleltetett kiértékelésről* beszélünk.

Az eddigiek alá:

```
var i = 0;
foreach (var n in numbers
                .Where(p => p > 2)
                .Select(p => new { p, x = ++i }))
{
    Console.WriteLine($"{n} - {i}");
}

Console.WriteLine();

i = 0;
foreach (var n in numbers
                .Where(p => p > 2)
                .Select(p => new { p, x = ++i })
                .ToList())
{
    Console.WriteLine($"{n} - {i}");
}
```

A `ToList` hívásunk először bejárja az iterátort és visszaad egy listát, amelybe összegyűjti az `IEnumerable` elemeit. Ezért az `i` változónk a második esetben nem együtt inkrementálódik a bejárással, mert az kétszer történik meg. Az első bejáráskor (a `ToList` hívásakor) inkrementálódik az `i` értéke, másodjára pedig már csak bejárjuk a kapott listát. Eddigre az `i` értéke már meg van növelve.

Ezzel a megközelítéssel futásidőben is állíthatunk össze egy időben változó lekérdezést, amit majd egyszer, a későbbiekben fogunk bejárni (pl. sorosításkor).

2.4.2 Aszinkron működés

Töltsünk le egy HTML oldalt, és ezen a problémán keresztül bemutatjuk az aszinkron programozási modellt. A `HttpClient` működésének a részletesebb ismertetése most nem téma, csak a legalapvetőbb funkciókat fogjuk használni.

A fő gond, hogy a hosszan futó műveletek blokkolhatják a fő/UI/aktuális szál futását, mindez kliens alkalmazások esetében úgy jelentkezik, hogy nem lesz az alkalmazásunk reszponzív a felhasználói bemenetekre; szerveralkalmazások esetében pedig az adott kérést kiszolgáló szál feleslegesen blokkolódik, amikor esetleg mással is tudna foglalkozni.

Ötlet: a hosszan tartó műveleteket végezzük aszinkron módon, és ha az befejeződött az eredményről valamilyen módon értesüljünk. A keretrendszer többféle mintát kínál erre:

- Asynchronous Programming Model (APM),
- Event-based Asynchronous Pattern (EAP),
- Task-based Asynchronous Pattern (TAP).

Mi most a legutóbbival foglalkozunk csak, a többi jórészt elavultnak számít ma már.

TAP és async/await alapjai

A TAP-ra már C# nyelvi támogatást is kapunk az `async` / `await` kulcsszavakon keresztül. Vegyünk fel egy új metódust és hívjuk meg a legfelső szintű kódban. A megírt metódus írása során hivatkozzuk be a `System.Net.Http` névteret. A kód semmi mást nem csinál, csak elindít aszinkron módon egy HTTP GET kérést a megadott URL-re, illetve a válasz tartalmát is aszinkron módon kiolvassa és egy részét kiírja a konzolra.

```
LoadWebPageAsync();
Console.WriteLine("Ez a vége");
Console.ReadKey();

static async void LoadWebPageAsync()
{
  using (var client = new HttpClient())
  {
    var response = await client.GetAsync(new Uri("http://www.bing.com"));
    Console.WriteLine(response.StatusCode.ToString());

    var content = await response.Content.ReadAsStringAsync();
    Console.WriteLine(content.Take(1000).ToArray());
  }
}
```

await: Mindig egy `Task` `await`-elhető (vagy taszk szerű dolog: vagyis van neki `GetAwaiter` metódusa, ami meghatározott metódusokkal rendelkező objektummal tér vissza)! Akár létre is hozhatunk egy `Task`-ot, amit egy lokális változóban tárolunk, akkor azt is tudjuk `await`-elni.

async: Ha await-elni akarunk, akkor muszáj `async`-nak lennie a tartalmazó metódusnak, mert ilyenkor építi fel a fordító az aszinkron végrehajtáshoz szükséges állapotgépet.

Debuggoljuk ki! Minden `Console`, `async` sorra tegyünk töréspontot, debuggolás során (F5) kövessük végig, milyen sorrendben éri el őket a végrehajtás. Nézzük meg, melyik rész milyen szálon fut le (debug közben menu:Debug[Windows > Threads]). A `LoadWebPageAsync` utáni rész előbb fog lefutni, mint az első `await` utáni rész. Az `await` utáni rész nem a *Main Thread*-en fut. Figyeljük meg azt is, hogy az *Ez a vége* szöveg hamarabb kiíródik, mint a HTML oldal letöltése.

Próbáljuk ki a `Console.ReadKey`-t kikommentezve is, ilyenkor jó eséllyel hamarabb leáll a process, minthogy a `Task` befejeződne. Az ilyen fire-and-forget típusú hívásoknál nem figyel arra senki, hogy itt még valami háttérművelet folyik.

> **async void kerülendő**
>
> Az `async void` általában helytelen kód, mert nem lehet bevárni a háttérművelet végét. Az `async Task` máris jobb a bevárhatóság és a hibakezelés miatt, és alig kell módosítani a kódot. Kivétel, amikor valamiért kötelező a `void`, például, ha esemény vagy interfész előírja.

Az oldalletöltés bevárása

Módosítsuk úgy a kódot, hogy a `LoadWebPageAsync` utáni rész várja meg a letöltés befejeződését. Ez akkor jó például, ha a letöltés után valamit még szeretnék elvégezni a hívó függvényben.

Módosítsuk a `LoadWebPageAsync` fejlécét, hogy taszkot adjon vissza:

```
public static async Task LoadWebPageAsync() //void helyett Task
```

Várjuk be az aszinkron művelet végét a legfelső szintű kódban.

```
await LoadWebPageAsync(); //await bekerült
Console.WriteLine("Ez a vége");
/*Console.ReadKey();*/
```

Figyeljük meg, hogy így már az *Ez a vége* felirat már a letöltés után jelenik meg.

`await`-et használtunk a legfelső szintű kódban, ilyenkor automatikusan `async` kulcsszóval ellátott `Main` generálódik - valami hasonló, mint az alábbi kódrészlet.

```
await LoadWebPageAsync();
Console.WriteLine("Ez a vége");
//Console.ReadKey();
```

Háttérművelet eredményének visszaadása

Alakítsuk át, hogy a weboldal tartalmának kiíratása a legfelső szintű kódban történjen, és a `LoadWebPageAsync` csak adja vissza a tartalmat `string`-ként. Ehhez módosítsuk a visszatérési értéket `Task<string>`-re, így az `await` már eredménnyel fog tudni visszatérni.

```
var content = await LoadWebPageAsync();
Console.WriteLine(content);
Console.WriteLine("Ez a vége");
Console.ReadKey();
static async Task<string> LoadWebPageAsync() //generikus paraméter
{
    using (var client = new HttpClient())
    {
        var response = await client.GetAsync(new Uri("http://www.bing.com"));
        Console.WriteLine(response.StatusCode.ToString());

        var content = await response.Content.ReadAsStringAsync();
        return new string(content.Take(1000).ToArray());
    }
}
```

A `return` valójában ezen `Task` eredményét állítja be `async` metódusok esetében, és nem egy nemgenerikus `Task` objektummal kell visszatérjünk.

2.4.3 Nem(igazán) nullozható referencia típusok

Korábban láttuk, hogy hogyan lehet egy érték típusnak null értéket adni (`Nullable<T>`). Az érem másik oldala a C# 8-ban megjelent nem nullozható referencia típusok. Nem egy új típust vezettek be, hanem az eddig megszokott típusneveket értelmezi máshogyan a fordító. A projektfájlban az alábbi beállítás kapcsolja be ezt a funkciót.

```
<Nullable>enable</Nullable>
```

> **Egyéb konfigurációs lehetőségek**
>
> Ezen kívül még preprocessor direktívákkal is szabályozhatjuk a működést.

Induljunk ki egy egyszerű személyeket nyilvántartó adatosztályból, ahol elhatározzuk, hogy a középső név kivételével a többi névdarab nem nullozható szöveg lesz.

```
Console.WriteLine("Hello World!");
class Person
{
   string FirstName;   // Not null
   string? MiddleName; // May be null
   string LastName;    // Not null
}
```

Ez máris számos figyelmeztetést generál. A nem nullozható referencia típusok bekapcsolásával alapesetben nem hibák, csak új figyelmeztetések generálódnak. A vezetéknév és keresztnév adatoknak nem szabadna `null` értékűnek lennie (a sima `string` típus nem nullozható típust jelent), viszont így az alapérték nem egyértelmű, explicit inicializálnunk kellene.

Fontos megértenünk, hogy a string típus fizikailag továbbra is lehet null értékű, mindössze a fordító számára jelezzük, hogy szándékunk szerint sohasem szabadna `null` értéket felvennie. A fordító cserébe figyelmeztet, ha ezt megsértő kódot detektál.

Az egyik legkézenfekvőbb megoldás (az inline inicializáció mellett), ha konstruktorban inicializálunk konstruktorparaméter alapján. Adjunk konstruktort a típusnak:

```
public Person(string fname, string lname, string? mname)
{
   FirstName = fname;
   LastName = lname;
   MiddleName = mname;
}
```

> **Rebuild**
>
> Ha biztosan látni akarjuk az összes figyelmeztetést, akkor sima Build művelet helyett használjuk a Rebuild-et.

Ezzel meg is oldottunk minden figyelmeztetést.

> **Konstruktorok**
>
> Sajnos a kötelezően konstruktoron keresztüli inicializáció nem mindig működik, például a sorosítók általában nem szeretik, ha nincs alapértelmezett konstruktor.

Mennyire okos a fordító a `null` érték detektálásában? Nézzünk pár példát! Az alábbi statikus függvényt tegyük bele a `Person` osztályunkba és vegyük fel a `using static System.Console;` névtérhivatkozást is.

```
static void M(string? ns)
{
   WriteLine(ns.Length); // (1)!
   if (ns != null)
   {
      WriteLine(ns.Length); // (2)!
   }
   if (ns == null)
   {
      return;
   }
   WriteLine(ns.Length); // (3)!
   ns = null;
   WriteLine(ns.Length); // (4)!
   string s = default(string); // (5)!
```

```
    string[] a = new string[10]; // (6)!
}
```

1. Figyelmeztetés lehetséges `null` értékre, mert a típusa szerint nullozható.
2. Ha egy egyszerű `if`-fel levizsgáljuk, akkor máris ok. Pedig pl. többszálú környezetben az `if` kiértékelése és ezen sor végrehajtása között a változó akár `null` értékre is beíródhat.
3. Az előtte lévő rövidzár is megnyugtatja a fordítót, így itt sincs figyelmeztetés.
4. Ezt az előző sor alapján figyelmeztetéssel jutalmazza.
5. Ez is figyelmeztetés, a `default` operátor által adott értékkel (`null`) nem inicializálhatunk.
6. Ez viszont nem figyelmeztetés, pedig egy csomó `null` jön létre. Ha ez figyelmeztetés lenne, az aránytalanul megnehezítené a tömbök kezelését.

Látható, hogy az egyszerűbb eseteket jól kezeli a fordító, de korántsem mindenható, illetve nem mindig szól akkor sem, amikor egyébként szólhatna.

A további példákhoz vegyünk fel pár segédfüggvényt a `Person` osztályba:

```
private Person GetAnotherPerson()
{
    return new Person(LastName, FirstName, MiddleName ?? string.Empty);
}

private void ResetFields()
{
    FirstName = default!;
    LastName = null!;
    MiddleName = null;
}
```

Látható, hogy vannak megkerülő megoldások arra, hogy ráerőszakoljuk a fordítóra az akaratunkat, a felkiáltójel használatával beírhatunk `null` értékeket nem nullozható változókba (ez az ún. **null forgiving operator**). Illetve `string` esetén null helyett használhatjuk az üres `string` értéket - ami nem biztos, hogy sokkal jobb a `null` értéknél. Mindenesetre ezek a függvények nem okoznak újabb figyelmeztetéseket.

Nézzük meg, hogy mennyire tudja lekövetni a fenti függvények működését a fordító. Vegyünk fel ennek tesztelésére egy újabb függvényt a `Person` osztályba:

```
void M(Person p)
{
    if (p.MiddleName != null)
    {
        p.ResetFields();
        WriteLine(p.MiddleName.Length); // (1)!

        p = GetAnotherPerson();
        WriteLine(p.MiddleName.Length); // (2)!
    }
    p.FirstName = null;         // (3)!
    p.LastName = p.MiddleName;  // (4)!
}
```

1. A fordító nem követi le, hogy a `ResetFields` veszélyes módon változtatja az állapotot, csak azt nézi, hogy az `if` már kivédte a veszélyt.
2. Ez egy fals pozitívnak tűnő eset, az előző sorban lévő függvény alapján a `p.MiddleName` nem lehetne `null` , de a fordító csak azt figyeli, hogy a beburkoló `if` ellenőrzése a `p` megváltozása miatt már nem érvényes.
3. Egyértelműen jogos figyelmeztetés.
4. Jogos a figyelmeztetés, mert nem kezeljük a `p.MiddleName == null` esetet.

Struktúratagok esetén is a fals negatív eset jön elő. Próbáljuk ki, akár a `Person` osztályba írva:

```
struct PersonHandle
{
   public Person person;
}
```

Nem kapunk figyelmeztetést.

A felkiáltójeles ráerőszakolást a `ResetFields` -ben látható ámokfutás helyett inkább a fals pozitív esetek kezelésére használjuk. Javítsuk ki a `GetAnotherPerson` hívás miatti fals pozitív esetet az `M(Person)` függvényben:

```
p = GetAnotherPerson();
WriteLine(p.MiddleName!.Length); //bekerült egy '!'
```

Figyeljük meg, ahogy a figyelmeztetés eltűnik.

Ha igazán elkötelezettek vagyunk a `null` kiirtása mellett, akkor bekapcsolhatjuk, hogy minden, a `null` kezelés miatti, fordító által detektált figyelmeztetés legyen hiba. A projekt beállítási között (menu:a projekten jobbklikk[Properties]), a *Build* lapon adjuk meg a *Treat specific warnings as errors* opciónak a `nullable` értéket. (Ha több értéket akarunk megadni, akkor a `;` elválasztót alkalmazhatjuk.)

Ellenőrizzük, hogy tényleg hibaként jelennek-e meg az eddigi `null` kezelés miatti figyelmeztetések.

Mivel ez csak egy példakód, ne javítsuk ki a hibákat, csak távolítsuk el a projektet a solutionből (menu:a projekten jobbklikk[Remove]).

2.4.4 Tuple nyelvi szinten, lokális függvények, Dispose minta

Tuple nyelvi szinten, lokális függvények

Készítsünk Fibonacci számsor kiszámolására alkalmas függvényt, ahol használjuk ki az alábbi két új nyelvi elemet. Természetesen nagyon sokféleképpen meg lehetne valósítani ezt a metódust, de most kifejezetten a *tuple*-ök nyelvi támogatását és lokális függvényeket szeretnénk demonstrálni.

- Lokális függvények: ezek a függvények csak adott metódusban láthatók.Két esetben érdemes őket használni: ha nem szeretnénk „szennyezni” a környező osztályt különféle privát segédmetódusokkal, vagy ha egy mélyebb, komplexebb hívási láncban nem szeretnénk a paramétereket folyamatosan továbbpasszolni, ugyanis ezek a metódusok elérik a külső scope-on található változókat is (a lenti esetben például az `x` -et).
- Value tuple típus: a tuple (ennes) több összetartozó érték összefogása, ami gyors, nyelvi szinten támogatott adattovábbítást tesz lehetővé - gyakorlatilag inline, nevesítetlen struktúratípust hozunk így létre. Publikus API-kon, függvényeken nem érdemes használni, viszont privát, belső használatnál sebességnövekedést és API tisztulást érhetünk vele el. Érték típus.

> **Referencia típusú Tuple<>**
>
> Léteznek generikus `Tuple<>` típusok is. Ezek referencia típusok, hasonló szerepet töltenek be, viszont az egyes értékeiket az elég semmitmondó `Item1` , `Item2` ... neveken lehet elérni.

```
static long Fibonacci(long x)
{
   (long Current, long Previous) Fib(long i) # (1)!
```

```
    {
        if (i == 0) return (1, 0);
        var (curr, prev) = Fib(i - 1); # (2)!
        Thread.Sleep(100); # (3)!
        return (curr + prev, curr);
    }

    return x < 0
        ? throw new ArgumentException("Less negativity please!", nameof(x))
        : Fib(x).Current;
}
```

1. Nevesített tuple visszatérés. Ez egy lokális függvény, szintaxist tekintve függvényen belüli függvény.
2. Az eredmény eltárolása egy tuple változóban. Ezzel dekonstruáljuk is, darabokra szedjük a tuple-t, mert `curr`, `prev` változón keresztül elérjük a két `long` alkotórészt. Ugyanezen sorban történik a rekurzív hívás is.
3. Lassú művelet szimulációja mesterséges késleltetéssel.

> **Dekonstrukció**
>
> A dekonstrukciós szintaxis a korábbi gyakorlaton megismert rekord típusok esetén is működik.

Dispose minta

A Dispose minta az erőforrás-felszabadítás megfelelő megvalósításához készült. Hasonló elv mentén üzemel, mint a destruktor, viszont a minta nem feltétlenül kötött az objektum életciklusának elejéhez és végéhez. Amennyiben egy objektum megvalósítja az `IDisposable` interfészt, van `Dispose` metódusa. A metódus meghívásával az objektum által használt, nem a keretrendszer által menedzselt erőforrásokat szabadítjuk fel. Nem csak memóriafoglalásra kell gondolni, hanem lehetnek nyitott fájlrendszeri handle-ök, adatkapcsolatok, stream-ek, vagy üzleti erőforrások, tranzakciók.

Mérjük meg az első pár Fibonacci szám kiszámítását (a mesterséges késleltetéssel):

```
var sw = Stopwatch.StartNew();
foreach (var n in Enumerable.Range(1, 15))
{
  Console.WriteLine($"{n}: {Fibonacci(n)}");
}
sw.Stop();
Console.WriteLine($"Elapsed: {sw.ElapsedMilliseconds}");
Console.ReadKey();
```

Ez így jó, működik, viszont nem újrahasznosítható ez az időmérési mechanizmus.

Készítsünk egy saját időmérő osztályt `StopwatchWrapper` néven, ami a `Stopwatch` használatát egyszerűsíti a **Dispose** mintán keresztül.

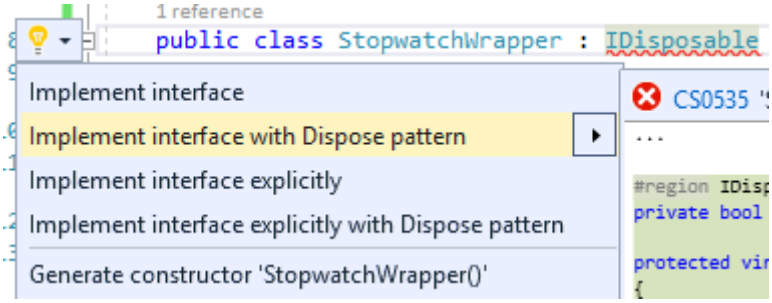
```
public class StopwatchWrapper : IDisposable
{
    public Stopwatch Stopwatch { get; }

    public string Title { get; }

    public StopwatchWrapper(string? title = default)
    {
        Title = title ?? Guid.NewGuid().ToString();
        Console.WriteLine($"Task {title} starting at {DateTime.Now}.");
        Stopwatch = Stopwatch.StartNew();
```

```
    }
}
```

Ha kérjük a villanykörte segítségét az `IDisposable`-ön, akkor 2x2 lehetőségünk van: megvalósítjuk az interfészt implicit vagy explicit, illetve megvalósítjuk-e az interfészt a Dispose mintát alkalmazva. Valósítsuk meg implicit a Dispose mintát!



Dispose minta implementálása IntelliSense segítségével

Fussuk át a generált kódot, ami szépen kommentezett. A pattern lényege, hogy a nem menedzselt erőforrásokat (*unmanaged objects / resources*) szükséges felszabadítanunk, amit a `Dispose` metódusokban, illetve menedzselt kód esetén a kommentekkel kijelölt helyen érdemes elvégeznünk. Készítsük el az időmérő mechanizmust!

```
protected virtual void Dispose(bool disposing)
{
    if (!disposedValue)
    {
        if (disposing)
        {
            Stopwatch.Stop();
            Console.WriteLine(
                $"Task {Title} completed in { Stopwatch.ElapsedMilliseconds} ms "+
                $"at { DateTime.Now}");
        }
        disposedValue = true;
    }
}
```

Csak felügyelt erőforrásokkal (*managed objects*) dolgozunk, így csak egy helyen kellett a leállító logikát megadnunk.

Az `IDisposable` interfészt megvalósító elemekkel használhatjuk a `using` konstrukciót:

```
using (new StopwatchWrapper("Fib 1-15"))
{
    foreach (var n in Enumerable.Range(1, 15))
    {
        Console.WriteLine($"{n}: {Fibonacci(n)}");
    }
}
```

Tehát a `using` használatával a blokk elejét és végét tudjuk kezelni. Gyakorlatilag egy `try-finally`-val ekvivalens a minta, a `finally`-ben meghívódik a `Dispose` metódus.

Jelenleg csak a folyamat végén kapunk jelentést az eltelt időről. Részidők kiírásához készítsünk egy segédfüggvényt a `StopwatchWrapper`-be:

```
public void Snapshot(string text) =>
    Console.WriteLine(
        $"Task {Title} snapshot {text}: {Stopwatch.ElapsedMilliseconds} ms"
    );
```

Hívjuk meg a `foreach` ciklusból:

```
using (var sw = new StopwatchWrapper("Fib 1-15"))
{
    foreach (var n in Enumerable.Range(1, 15))
    {
        sw.Snapshot(n.ToString());
        Console.WriteLine($"{n}: {Fibonacci(n)}");
    }
}
```

2.5 ASP.NET Core alapszolgáltatások

2.5.1 Projekt létrehozása

Ezen a gyakorlaton nem a beépített API projektsablont fogjuk felhasználni, hanem egy üres ASP.NET Core projektből próbáljuk felépíteni és megérteni azt a funkcionalitást, amit egyébként az előre elkészített VS projektsablonok adnának készen a kezünkbe.

Generálás

Hozzunk létre a Visual Studioban egy új, C# nyelvű projektet az *ASP.NET Core Empty* sablonnal, a neve legyen *HelloAspNetCore*. Megcélzott keretrendszerként adjuk meg a *.NET 8*-ot. Minden extra opció legyen kikapcsolva, a docker és a HTTPS is.

Kitérő: NuGet és a keretrendszert alkotó komponensek helye

A .NET 8 és az ASP.NET Core gyakorlatilag teljes mértékben publikusan elérhető komponensekből épül fel. A komponensek kezelésének infrastruktúráját a NuGet csomagkezelő szolgáltatja. A csomagkezelőn keresztül elérhető csomagokat a nuget.org listázza és igény esetén a NuGet kliens, illetve a .NET Core eszközök (dotnet.exe, Visual Studio) is innen töltik le. A fejlesztőknek teljesítményszempontból nem érné meg az alap keretrendszert alkotó csomagokat állandóan letöltögetni, így a klasszikus keretrendszerekhez hasonlóan a .NET 8 telepítésekor egy könyvtárba (Windows-on ide: **C:\Program Files (x86)\dotnet**, illetve **C:\Program Files\dotnet**) bekerülnek az alap keretrendszert alkotó komponensek - lényegében egy csomó .dll különböző alkönyvtárakban. A futtatáshoz szükséges szerelvények a **shared** alkönyvtárba települnek, ezek az ún. **Shared Framework**-ök. A gépen futó különböző .NET Core/8 alkalmazások közösen használhatják ezeket. A fejlesztéshez az alapvető függőségeket a **packs** alkönyvtárból hivatkozhatjuk.

Nem fejlesztői, például végfelhasználói vagy szerver környezetben- ahol nem is biztos, hogy fel van telepítve az SDK, nem feltétlenül így biztosítjuk a függőségeket, de ennek a boncolgatása nem témája ennek a gyakorlatnak.

Eredmény

Nézzük meg, milyen projekt generálódott:

- **.csproj**: (menu:Projekten jobb gomb[Edit Project File]) a projekt fordításához szükséges beállításokat tartalmazza. Előző verziókhoz képest itt erősen építenek az alapértelmezett értékekre, hogy minél karcsúbbra tudják fogni ezt az állományt.

  Project SDK: projekt típusa (Microsoft.NET.Sdk.Web), az eszközkészlet funkcióit szabályozza, meghatározza a futtatáshoz használatos shared framework-öt, illetve meghatározza a megcélzott keretrendszert is(lásd lentebb).

  TargetFramework: *net8.0*. Ezzel jelezzük, hogy .NET 8-os API-kat használunk az alkalmazásban.
- **Connected Services**: külső szolgáltatások, amiket használ a projektünk, most nincs ilyenünk.
- **Dependencies**: a keretrendszer alapfüggőségei és egyéb NuGet csomagfüggőségek szerepelnek itt. Egyelőre csak keretrendszer függőségeink vannak.

Frameworks: két alkönyvtárat (Microsoft.AspNetCore.App, Microsoft.NETCore.App) hivatkozunk a .NET SDK **packs** alkönyvtárából. Ezek a függőségek külső NuGet csomagként is elérhetőek, de ahogy fentebb jeleztük, nem érdemes úgy hivatkozni őket.

Analyzers: speciális komponensek, amik kódanalízist végzenek, de egyébként ugyanúgy külső függőségként (NuGet csomag) kezelhetjük őket. Ha kibontjuk az egyes analizátorokat, akkor láthatjuk, hogy miket ellenőriznek. Ezek a függőségek a futáshoz nem szükségesek.

- **Properties**: duplakattra előjön a klasszikus projektbeállító felület.

  launchSettings.json: a különböző indítási konfigurációkhoz tartozó beállítások (lásd később).

- **appsettings.json**: futásidejű beállítások helye. Kibontható, kibontva a különböző környezetekre specifikus konfigurációk találhatóak (lásd később).

LEGFELSŐ SZINTŰ KÓD, MINIMÁL API

Az előző ASP.NET verzióval ellentétben, itt már az ASP.NET Core alkalmazások a születésüktől fogva klasszikus konzolos alkalmazásként is indíthatók, ekkor az alkalmazás alapértelmezett belépési pontja a legfelső szintű kód (esetleg a `Main` metódus). Az ASP.NET Core 6-os verzióban megjelent ún. **minimál API** segítségével már nem csak a konfigurációt tartalmazhatja ez a kód, hanem (egyszerű) kiszolgáló logikát is.

Esetünkben a következő lépéseket végzi el a generált kód:

- a hosztolási környezetet és az alkalmazás alapszolgáltatásait konfiguráló **builder** objektum összeállítása (`CreateBuilder` függvényhívás)
- a **builder** objektum alapján a hosztolási környezet és az alkalmazás szerkezetének felállítása (`Build` függvényhívás)
- végpontot definiál az alkalmazás gyökércímére minimál API segítségével. A végpont a meghívására a **Hello World!** szöveget adja vissza.
- a felállított szerkezet futtatása (`Run` függvényhívás)

Az igazán munkás feladat a builder megalkotása lenne, igen sok mindent lehetne benne konfigurálni, ez a kódban a `CreateBuilder` -ben történik, ami egy szokványos, az egész webalkalmazás működési környezetét meghatározó beállításokat elvégző kiinduló buildert állít elő. Ha valamit a kiinduló builderben megadottól eltérően szeretnénk, vagy új beállításokat adnánk meg, akkor a kiinduló builder objektumon történő függvényhívásokkal tehetnénk meg.

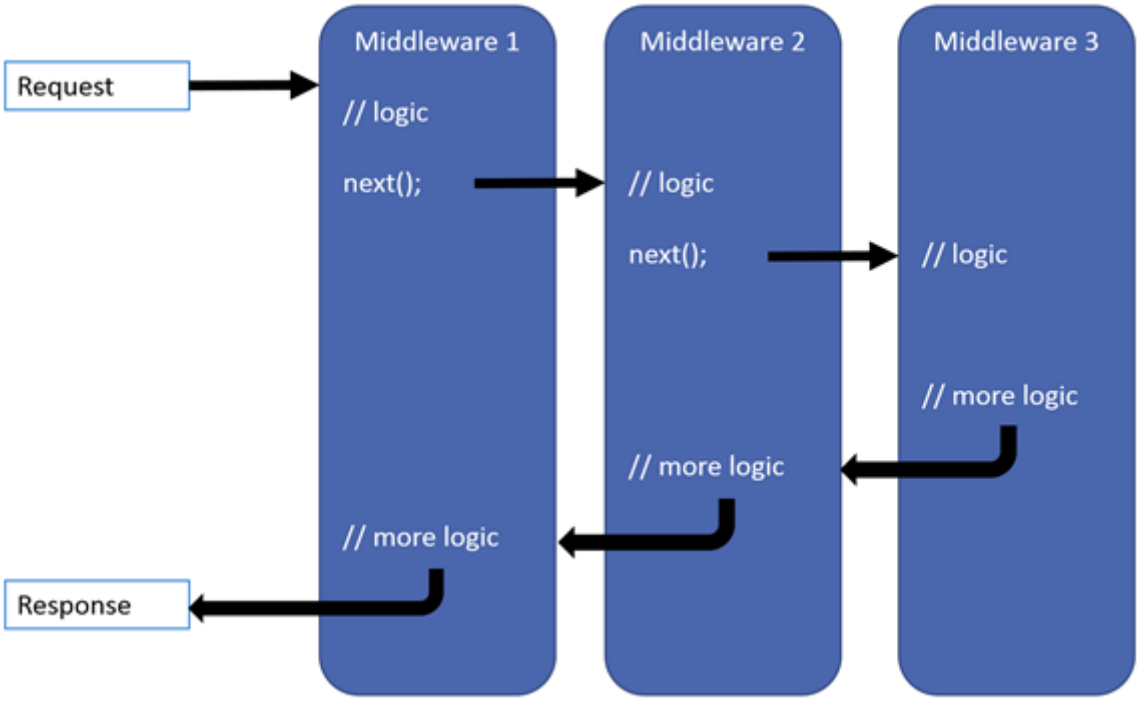
Mivel a kiinduló builderen nem végzünk semmilyen utólagos konfigurálást, így akár egy utasítással is megkaphatnánk az alkalmazásszerkezetet reprezentáló `WebApplication` példányt.

```
//var builder = WebApplication.CreateBuilder(args);
//var app = builder.Build();

var app = WebApplication.Create();
```

2.5.2 Végrehajtási pipeline, middleware-ek

Az ASP.NET Core-ban egy kérés kiszolgálása úgy történik, hogy a kérés egy csővezetéken halad (végig). A csővezeték middleware-ekből (MW) áll. Az alábbi ábra szemlélteti a middleware pipeline működését.



ASP.NET Core pipeline Forrás

Az ASP.NET Core alkalmazás alapszerkezete, hogy a befutó HTTP kérés (végig)fusson a middleware-ekből álló csővezetéken és valamelyik (alapesetben az utolsó) middleware előállítja a választ, ami visszairányban halad végig a csővezetéken. A csővezeték adja tehát az alkalmazás szerkezetét. A kiinduló csővezetéket a `WebApplication.Create` vagy a `builder.Build` építi fel, ezt utána `app.UseX` (X= MW neve) hívásokkal testreszabhatjuk, kiegészíthetjük.

Esetünkben a kiinduló csővezetékben három MW van:

- kivételkezelő middleware (`UseDeveloperExceptionPage`), ami az őt követő middleware-ek hibáit képes elkapni és ennek megfelelően egy a fejlesztőknek szóló hibaoldalt jelenít meg. Ez csak opcionálisan kerül beregisztrálásra attól függően, hogy most éppen *Development módban* futtatjuk-e az alkalmazást vagy sem. (lásd később)
- routing middleware (`UseRouting`), aminek a feladata, hogy a bejövő kérés és a végpontok (lásd lentebb) által adott információk alapján kitalálja, hogy melyik endpoint felé továbbítsa a bejövő kérést.
- végpontok middleware (`UseEndpoints`), ami a kiválasztott endpoint definíciójában megadott logika tényleges lefuttatásáért felel

Példa middleware regisztrációra

A kiinduló csővezeték regisztrálását megfigyelhetjük a `WebApplicationBuilder` forráskódjában - keressük az `app.UseX` sorokat.

A kiinduló projekt nem változtat a kiinduló csővezetéken, csak egy végpont definíciót ad meg (`app.MapGet` sor).

Fontos a sorrend

A middleware-ek sorrendje fontos. Ha nem megfelelő sorrendben regisztráljuk őket, nem megfelelő működés lehet az eredmény. A dokumentáció általában tartalmazza, hogy melyik middleware hova illeszthető be.

2.5.3 Hosztolási lehetőségek a fejlesztői gépen

Próbáljuk ki IIS Expressen keresztül futtatva, azaz a VS-ben az indítógomb (zöld nyíl) mellett az IIS Express felirat legyen! Ha nem ez a felirat van, állítsuk át az indítógomb jobb szélén lévő menüt lenyitva.

Két dolog is történik: az alkalmazásunk IIS Express webkiszolgálóban hosztolva kezd futni és egy böngésző is elindul, hogy ki tudjuk próbálni. Figyeljük meg az értesítési területen (az óra mellett) megjelenő IIS Express ikont, és azon jobbklikkelve a hosztolt alkalmazás címét (menu:jobbklikk[Show All Applications]).

A böngésző az alkalmazás gyökércímére navigál (a cím csak **localhost:port**-ból áll), így a **Hello World!** szöveg jelenik meg.

Böngésző választása

A indítógomb legördülőjében a böngésző típusát is állíthatjuk.

IIS Express

Az IIS Express a Microsoft webszerverének (IIS) fejlesztői célra optimalizált változata. Alapvetően csak ugyanarról a gépről érkező (localhost) kéréseket szolgál ki.

A másik lehetőség, ha közvetlenül a konzolos alkalmazást szeretnénk futtatni, akkor ezt az indítógombot lenyitva a projekt nevét kiválasztva tehetjük meg. Ebben az esetben egy beágyazott webszerverhez (*Kestrel*) futnak be a kérések. Próbáljuk ki a Kestrelt közvetlenül futtatva!

Most is két dolog történik: az alkalmazásunk konzolos alkalmazásként kezd futni, illetve az előző esethez hasonlóan a böngésző is elindul. Figyeljük meg a konzolban megjelenő naplóüzeneteket.

Éles hosting

Bár ezek a hosztolási opciók fejlesztői környezetben nagyon kényelmesek, érdemes áttekinteni az éles hosztolási opciókat itt. A Kestrel ugyan jelenleg már alkalmas arra, hogy kipublikáljuk közvetlenül a világhálóra, de mivel nem rendelkezik olyan széles konfigurációs és biztonsági beállításokkal, mint a már bejáratott webszerverek, így érdemes lehet egy ilyen webszervert a Kestrel elé rakni proxy gyanánt, például az IIS-t vagy nginx-et.

Rakjunk most a kiszolgáló logikánkba egy kivétel dobást a kiírás helyett, hogy kipróbáljuk a hibakezelő MW-t.

```
app.MapGet("/", () =>
{
    throw new Exception("hiba");
    //return "Hello World!"
```

```
}
);
```

Próbáljuk ki debugger nélkül (Ctrl + F5)!

Láthatjuk, hogy a kivételt a hibakezelő middleware elkapja és egy hibaoldalt jelenítünk meg, sőt még a konzolon is megjelenik naplóbejegyzésként.

2.5.4 Alkalmazásbeállítások vs. indítási profilok

Figyeljük meg, hogy most **Development** konfigurációban fut az alkalmazás (konzolban a *Hosting environment* kezdetű sor). Ezt az információt a keretrendszer környezeti változó alapján állapítja meg. Ha a **lauchSettings.json** állományt megnézzük, akkor láthatjuk, hogy az `ASPNETCORE_ENVIRONMENT` környezeti változó `Development`-re van állítva.

Próbáljuk ki Visual Studio-n kívülről futtatni. menu:Projekten jobb klikk[Open Folder in File Explorer]. Ezután a címsorba mindent kijelölve `cmd` + Enter, a parancssorba `dotnet run`.

Ugyanúgy fog indulni, mint VS-ből, mert az újabb .NET verziókban már a *dotnet run* is figyelembe veszi a **launchSettings.json**-t. A böngészőt magunknak kell indítani (most még) és elnavigálni a naplóban szereplő címre (**Now listening on: http://localhost:port** üzenetet keressünk).

Ha nem akarjuk ezt, akkor a `--no-launch-profile` kapcsolót használhatjuk a *dotnet run* futtatásánál.

Most az alkalmazásunk Production módban indul el, és ha a *localhost:5000*-es oldalt megnyitjuk a böngészőben, akkor nem kapunk hibaoldalt, de a konzolon továbbra is megjelenik a naplóbejegyzés.

Leállítás

A **dotnet run** futását Ctrl + C-vel állíthatjuk le.

Környezeti változók felvétele

A konzolban a `setx ENV_NAME Value` utasítással tudunk felvenni környezeti változót úgy, hogy az permanensen megmaradjon, és ne csak a konzolablak bezárásáig maradjon érvényben. (Admin/nem admin, illetve powershell konzolok különbözőképpen viselkednek)

Az eredeti logikánkat kommentezzük vissza.

```
app.MapGet("/", () =>
{
    //throw new Exception("hiba");
    return "Hello World!";
});
```

Az alkalmazás számára a különböző beállításokat JSON állományokban tárolhatjuk, amelyek akár környezetenként különbözőek is lehetnek. A generált projektünkben ez az **appsettings.json**, nézzünk bele - főleg naplózási beállítások vannak benne. A fájl a **Solution Explorer** ablakban kinyitható, alatta megtaláljuk az **appsettings.Development.json**-t. Ebben a *Development* nevű konfigurációra vonatkozó beállítások vannak. Alapértelmezésben az **appsettings.**

\<indítási konfiguráció neve>.json beállításai jutnak érvényre, felülírva a sima **appsettings.json** egyező értékeit (a pontosabb logikát lásd lentebb).

Állítsunk Development módban részletesebb naplózást. Az **appsettings.Development.json**-ben minden naplózási szintet írjunk `Debug`-ra.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "Microsoft.AspNetCore": "Debug"
    }
  }
}
```

Naplózási szintek

A naplózási szintek sorrendje itt található.

Próbáljuk ki, hogy így az alkalmazásunk futásakor minden böngészőbeli frissítésünk (`F5`) megjelenik a konzolon.

VS-ből is tudjuk állítani a környezeti változókat, nem kell a **launchSettings.json**-ben kézzel varázsolni. A projekt tulajdonságok *Debug* lapján az *Open debug launch profiles UI* szövegre kattintva egy dialógusablak ugrik fel, itt tudunk új indítási profilt megadni, illetve a meglévőeket módosítani. Válasszuk ki az aktuálisan használt profilunkat (projektneves), majd írjuk át az *ASPNETCORE_ENVIRONMENT* környezeti változó értékét az *Environment Variables* részen mondjuk *Production*-re.

Indítsuk ezzel a profillal és figyeljük meg, hogy már nem jelennek meg az egyes kérések a naplóban, bárhogy is frissítgetjük a böngészőt. Oka: nincs **appsettings.Production.json**, így az általános **appsettings.json** jut érvényre.

Konfigurációk forrása

Számos forrásból lehet konfigurációt megadni: parancssor, környezeti változó, fájl (ezt láttuk most), felhő (**Azure Key Vault**) stb. Ezek közül többet is használhatunk egyszerre, a különböző források konfigurációja a közös kulcsok mentén összefésülődik. A források (*configuration provider*-ek) között sorrendet adhatunk meg, amikor regisztráljuk őket, a legutolsóként regisztrált provider konfigurációja a legerősebb. Az alapértelmezett provider-ek regisztrációját elintézi a korábban látott kiinduló builder.

Statikus fájl MW

Hozzunk létre a projekt gyökerébe egy `wwwroot` nevű mappát (menu:jobbklikk a projekten[Add > New Folder]) és tegyünk egy képfájlt bele. (Ellophatjuk pl. a http://www.bme.hu honlap bal felső sarkából a logo-t)

A statikus fájlkezelést a teljes modularitás jegyében egy külön middleware-ként implementálták a *Microsoft.AspNetCore.StaticFiles* osztálykönyvtárban (az AspNetCore.App már függőségként tartalmazza, így nem kell külön hivatkoznunk), csak hozzá kell adnunk a pipeline-hoz.

```
app.UseStaticFiles();
app.MapGet("/", () => "Hello World!");
```

Próbáljuk ki!

Láthatjuk hogy a *localhost:port* címen még mindig a *Hello World!* szöveg tűnik fel, de amint a *localhost:port/[képfájlnév]*-vel próbálkozunk, a kép töltődik be. A static file MW megszakítja a pipeline futását, ha egy általa ismert fájltípusra hivatkozunk, egyébként továbbhív a következő MW-be. Az ilyen MW-eket ún. **terminális** MW-eknek hívjuk.

> **Breakpoint a lambdában**
>
> Ezt az egysoros endpoint logikára tett töréspontal is szemléltethetjük. Figyeljünk arra, hogy csak a **Hello World!** szövegre kerüljön a töréspont és ne az egész `MapGet` sorra, illetve csak akkor nézzük, hogy mi fut le, amikor a kép URL-re hívunk.

2.5.5 Web API

Minden API-nál nagyon magas szinten az a cél, hogy egy kérés hatására egy szerveroldali kódrészlet meghívódjon. ASP.NET Core-ban a Minimap API megközelítés mellett alkalmazható az MVC keretrendszer is, ahol a kódrészleteket függvényekbe írjuk, a függvények pedig ún. *kontrollerek*-be kerülnek. Egy controller általában az egy erőforrástípushoz kapcsolódó műveleteket fogja össze. Összességében tehát a cél, hogy a webes kérés hatására egy kontroller egy függvénye meghívódjon.

DummyController

Hozzunk létre egy új mappát *Controllers* néven. A mappába hozzunk létre egy kontrollert (menu:jobbklikk a Controllers mappán[Add > Controller... > a bal oldali fában Common > API > jobb oldalon API Controller with read/write actions]) `DummyController` néven. A generált kontrollerünk a *Microsoft.AspNetCore.Mvc.Core* csomagban található `ControllerBase` osztályból származik. (Ezt a csomagot sem kell feltennünk, mivel az **AspNetCore.App** függősége)

Adjuk hozzá a szolgáltatásokhoz a kontrollertámogatás szolgáltatást, és adjuk hozzá a csővezetékhez a kontroller kezelő MW-t. Az egysoros MW-t kommentezzük ki. Így néz ki a teljes legfelső szintű kód:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers(); // (1)!
var app = builder.Build();
/*var app = WebApplication.Create();*/ // (2)!
app.UseStaticFiles();
/*app.MapGet("/", () => "Hello World!");*/ // (3)!
app.MapControllers();
app.Run();
```

1. Kontrollertámogatás szolgáltatás regisztrálása
2. Mivel kell a kiinduló builder, így ezt az egysoros app inicializációt nem alkalmazhatjuk
3. Egysoros MW kikommentezve

Próbáljuk ki.

Az alapoldal üres, viszont ha az `/api/Dummy` címre hívunk, akkor megjelenik a `DummyController.Get` által visszaadott érték. A routing szabályok szabályozzák, hogy hogyan jut el a HTTP kérés alapján a végrehajtás a függvényig. Itt attribútum alapú routing-ot használunk, azaz a kontroller osztályra és a függvényeire biggyesztett attribútumok határozzák meg, hogy a HTTP kérés adata (pl. URL) alapján melyik függvény hívódik meg.

A `DummyController` osztályon lévő `Route` attribútum az `"api/[controller]"` útvonalat definiálja, melyből a `[controller]` úgynevezett token, ami jelen esetben a controller nevére cserélődik. Ezzel összességében megadtuk, hogy az `api/`

`Dummy` útvonal a `DummyController`-t választja ki, de még nem tudjuk, hogy a függvényei közül melyiket kell meghívni - ez a függvényekre tett attribútumokból következik. A `Get` függvényen levő `HttpGet` mutatja, hogy ez a függvény akkor hívandó, ha a GET kérés URL-je nem folytatódik - ellentétben a `Get(int id)` függvénnyel, ami az URL-ben még egy további szegmenst vár (ezért van egy `"{id}"` paraméter megadva az attribútum konstruktorban), amit az `id` nevű függvényparaméterként használ fel.

> **Routing lehetőségek**
>
> Az API-t publikáló alkalmazásoknál az attribútum alapú routing az ajánlott, de emellett vannak más megközelítések is, például **Razor** alapú weboldalaknál konvenció alapú routing az ajánlott. Bővebben a témakörről általánosan itt, illetve specifikusan webes API-k vonatkozásában itt lehet olvasni. A dokumentáció mennyiségéből látható, hogy a routing alrendszer nagyon szofisztikált és sokat tud, szerencsére az alap működés elég egyszerű és gyorsan megszokható.

Ha van időnk, próbáljuk ki az `/api/Dummy/[egész szám]` címet is. A `Get(int id)` függvény kódjának megfelelően, bármit adunk meg, az eredmény a *value* szöveg lesz.

2.5.6 Típusos beállítások, `IOptions<T>`

Fentebb láttuk, hogy a konfigurációt ki tudtuk olvasni az `IConfiguration` interfészen keresztül, de még jobb lenne, ha csoportosítva és csoportonként külön C# osztályokon keresztül látnánk őket.

Bővítsük az *appsettings.json*-t egy saját beállításcsoporttal (*DummySettings*):

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*", // a sor végére bekerült egy vessző
  "DummySettings": {
    "DefaultString": "My Value",
    "DefaultInt": 23,
    "SuperSecret":  "Spoiler Alert!!!"
  }
}
```

Hozzunk létre egy új mappát `Options` néven. A mappába hozzunk létre egy sima osztályt `DummySettings` néven, a szerkezete feleljen meg a JSON-ben leírt beállításcsoportnak:

```
public class DummySettings
{
    public string? DefaultString { get; set; }
    public int DefaultInt { get; set; }
    public string? SuperSecret { get; set; }
}
```

Regisztráljuk szolgáltatásként a `DummySettings` kezelését, és adjuk meg, hogy a példányt mi alapján kell inicializálni - a konfiguráció megfelelő szekciójára hivatkozzunk:

```
builder.Services.Configure<DummySettings>(
    builder.Configuration.GetSection(nameof(DummySettings)));
```

A `builder.Services` -ben regisztrált szolgáltatások valójában egy dependency injection (DI) konténerbe kerülnek regisztrálásra. Ez többek között lehetővé teszi, hogy az alkalmazáson belül konstruktorban paraméterként igényeljük a szolgáltatást. A paraméter értékét a DI alrendszer automatikusan tölti ki a regisztrált szolgáltatások alapján.

DI minta preferálása

ASP.NET Core környezetben (is) törekedjünk arra, hogy lehetőleg minden osztályunk minden függőségét a DI minta szerint a DI konténer kezelje. Ez nagyban hozzájárul a komponensek közötti laza csatolás és a jobb tesztelhetőség eléréséhez. Bővebb információ az ASP.NET Core DI alrendszeréről a dokumentációban található.

Igényeljünk `DummySettings` -t a `DummyController` konstruktorban:

```
private readonly DummySettings _options;

public DummyController(IOptions<DummySettings> options)
{
    options = options.Value;
}
```

IOptions<> és társai

Látható, hogy a beállítás `IOptions` -ba burkolva érkezik. Vannak az `IOptions` -nál okosabb burkolók is (pl. `IOptionsMonitor`), ami például jelzi, ha megváltozik valamilyen beállítás. Bővebb információ az `IOptions` és társairól a hivatalos dokumentációban található.

Az egész számot váró `Get` változatban használjuk fel az értékeket:

```
[HttpGet("{id}")]
public string Get(int id)
{
    return id % 2 == 0
        ? (options.DefaultString ?? "value")
        : options.DefaultInt.ToString();
}
```

Próbáljuk ki, hogy az `/api/Dummy/[páros szám]` , illetve `/api/Dummy/[páratlan szám]` végpontok meghívásakor a megfelelő értéket kapjuk-e vissza.

2.5.7 User Secrets

A projekt könyvtára gyakran valamilyen verziókezelő (pl. Git) kezelésében van. Ilyenkor gyakori probléma, hogy a konfigurációs fájlokba írt szenzitív információk (API kulcsok, adatbázis jelszavak) bekerülnek a verziókezelőbe. Ha egy publikus projekten dolgozunk, például publikus GitHub projekt, akkor ez komoly biztonsági kockázat lehet.

Szenzitív információk

Ne tegyünk a verziókezelőbe szenzitív információkat még privát repó esetében sem. Gondoljunk arra is, hogy a verziókezelő nem felejt! Ami egyszer már bekerült, azt vissza is lehet nyerni belőle (history).

Ennek a problémának megoldására egy eszköz a *User Secrets* tároló. Jobbklikkeljünk a projekten a Solution Explorer ablakban, majd válasszuk a *Manage User Secrets* menüpontot. Ennek hatására megnyílik egy **secrets.json** nevű fájl. Vizsgáljuk meg, hol is van ez a fájl: vigyük az egeret a fájlfül fölé - azt láthatjuk, hogy a fájl a felhasználónk saját könyvtárán belül van és az útvonal része egy GUID is. A projektfájlba (.csproj) bekerült ugyanez a GUID (a *UserSecretsId* címkébe).

Vegyünk fel egy új beállítást a **secrets.json**-ba, ami a `SuperSecret` értékét írja felül:

```
{
  "DummySettings": {
    "SuperSecret": "SECRET"
  }
}
```

Részleges felülírás

A **secrets.json**-ban csak azokat a json levél elemeket kell felvenni, amiket felül akarunk írni. Ez a módszer működik a sima **appsettings.json** környezetfüggő változóira is.

Töréspontot letéve (pl. a `DummyController` konstruktorának végén) ellenőrizzük, hogy a titkos érték melyik fájlból jön. Ehhez meg kell hívnunk böngészőből az `api/dummy` címet.

User Secrets csak Development módban

Fontos tudni, hogy a *User Secrets* tároló csak **Development** mód esetén jut érvényre, így figyeljünk rá, hogy a megfelelő módot indítsuk és a környezeti változók is jól legyenek beállítva.

Ez az eljárás tehát a futtató felhasználó saját könyvtárából a GUID alapján kikeresi a projekthez tartozó **secrets.json**-t, annak tartalmát pedig futás közben összefésüli az **appsettings.json** tartalmával. Így szenzitív adat nem kerül a projekt könyvtárába.

Titkok éles környezetben

Mivel a *User Secrets* tároló csak Development mód esetén jut érvényre, így ha az éles változatnak szüksége van ezekre a titkos értékekre, akkor további trükkökre van szükség. Ilyen megoldás lehet, ha a felhős hosztolás esetén a felhőből (pl. Azure App Service Configuration) vagy felhőbeli titoktárolóból (pl. Azure Key Vault) vagy a DevOps eszközből (pl. Azure DevOps Pipeline Secrets) töltjük be a szenzitív beállításokat.

2.5.8 Epilógus - WebApplicationBuilder

Az eddigiekből látható, hogy számos alapszolgáltatás már a `CreateBuilder` hívás által visszaadott kiinduló builderben konfigurálva van. Ilyen az alap (`IOptions` nélküli) alkalmazásbeállítások kezelése vagy a naplózás. A `CreateBuilder` a `WebApplicationBuilder` internal konstruktorát hívja.

A `WebApplicationBuilder` elődje az `IWebHostBuilder`, ez utóbbinak a dokumentációját tanulmányozva érthetjük meg, hogy mi mindent tud a kiinduló builder.

2.6 Entity Framework Core I-II.

2.6.1 Az Entity Framework leképezési módszerei

Az objektum-relációs (OR) leképzés (*mapping*) két fő részből áll: az egyik az adatbázis séma, a másik pedig egy menedzselt kódbéli objektummodell. Esetünkben a C# kódban lévő osztályokat képezzük le adatbázisbeli objektumokká, ezt hívjuk *Code-First* mapping módszernek. A másik irány is lehetséges, ha már van egy adatbázis sémánk, akkor azt is leképezhetjük Code-First modellé. Ezt a folyamatot *Reverse Engineered Code-First* nek vagy *scaffolding*--nak hívjuk (ez utóbbival nem foglalkozunk ezen gyakorlat keretében).

Akárhogy is, az Entity Framework Core (EF) mint OR leképező eszköz (ORM) használatához az alábbi összetevőkre van szükség:

- objektummodell kódban
- relációs modell az adatbázisban
- leképezés (mapping) az előbbi kettő között, szintén kódban megadva
- maga az Entity Framework Core, mint (NuGet) komponens
- Entity Framework Core kompatibilis adatbázis driver (provider)
- adatbázis kapcsolódási adatok, connection string formátumban

2.6.2 A Code-First leképezési módszer

A Code-First módszer lényege, hogy elsőként az OO entitásokat definiáljuk egyszerűen programkódban, majd a leképezést szintén programkódban. A leképezés alapján az EF eszközök képesek az adatbázis létrehozására, inicializálására és a séma változáskövetésére is (lásd lentebb a *Code-First Migrations* részt).

Az entitások definiálása

Készítsünk egy .NET 8 konzolos alkalmazást (csak **ne** EF legyen a neve), majd a projekten belül hozzunk létre egy *Entities* nevű mappát. Adjunk hozzá a mappához egyszerű osztályokat az alábbi sémának megfelelően:

- `Product` (`Id` : `int` , `Name` : `string` , `UnitPrice` : `int`)
- `Order` (`Id` : `int` , `OrderDate` : `DateTime`)
- `Category` (`Id` : `int` , `Name` : `string`)

Az osztályok legyenek publikusak, az attribútumok pedig egyszerű auto-implementált propertyk (*prop* snippet).

A `string` típusú property-k esetén figyelmeztet a fordító, hogy nem nullozható referencia típusú property inicializáció után is `null` értékű lehet. Ennek kivédésére az ajánlott módszer a `required` kulcsszó használata a property előtt, ezzel kiválthatóak a felesleges konstruktorok, amelyben csak a kötelezőség miatt várunk el paramétereket és állítjuk be a propertyk értékét.

EF Core és konstruktorok required nélkül

A fenti megoldás nem működött még C# 11-ben, így ott konstruktort kellett készítenünk. Ezt a konstruktort az EF is fogja hívni, így neki automatikusan tudnia kell, hogy melyik paraméter melyik tulajdonságot állítja - pedig ez a konstruktor szignatúrájából alapesetben nem kikövetkeztethető. Emiatt önkéntesen tartanunk kell magunkat ahhoz, hogy a konstruktorparaméter nevének és a property nevének egyeznie kell, kivéve, hogy a paramétere neve kezdődhet kisbetűvel is (**camel casing**).

Példaként így néz ki a `Product` konstruktor:

```
public Product(string name)
{
    Name = name;
}
```

Quick Action

A Visual Studio **Quick Action**-ként fel szokta ajánlani a **Generate constructor [konstruktorfejléc]** vagy **Add parameter to [konstruktorfejléc]** gyors kódgenerálási lehetőségeket, amivel létrehozhatjuk vagy bővíthetjük a szükséges konstruktort.

Mapping és egyéb metaadatok megadása I.

Eddig megadtuk az entitás nevét, a relációs attribútumok nevét és típusát, azonban ezen felül még sok mindent lehet/kell megadni: az entitás elsődleges kulcsa, idegen kulcsok, relációk, kényszerek és egyéb mapping információk (pl. hogy mi legyen a relációs attribútum oszlopneve az adatbázisban). A Code-First stratégia kétfajta módszert is kínál ezek megadására. Az egyik módszer, hogy C# attribútumokat helyezünk az entitásosztályok különböző részeire, a másik, hogy ún. fluent jellegű kódot alkalmazunk. Ez utóbbi módszer elsőre furcsán néz ki, de többet tud (van, amit attribútummal nem lehet megadni).

A fenti két módszert kiegészíti a konvenció alapú konfiguráció, amikor az EF a rendelkezésekre álló adatokból automatikusan következtet ki a metaadatokat: például gyakori, hogy az elsődleges kulcs neve tartalmazza az *id* szöveget. Az EF tehát a konvenció alapján kitalálhatja, hogy melyik ez elsődleges kulcs oszlop. Ha valamit rosszul találna ki, vagy változtatni akarunk a kitalált neveken, akkor azt az attribútumos vagy a fluent megadással tehetjük meg.

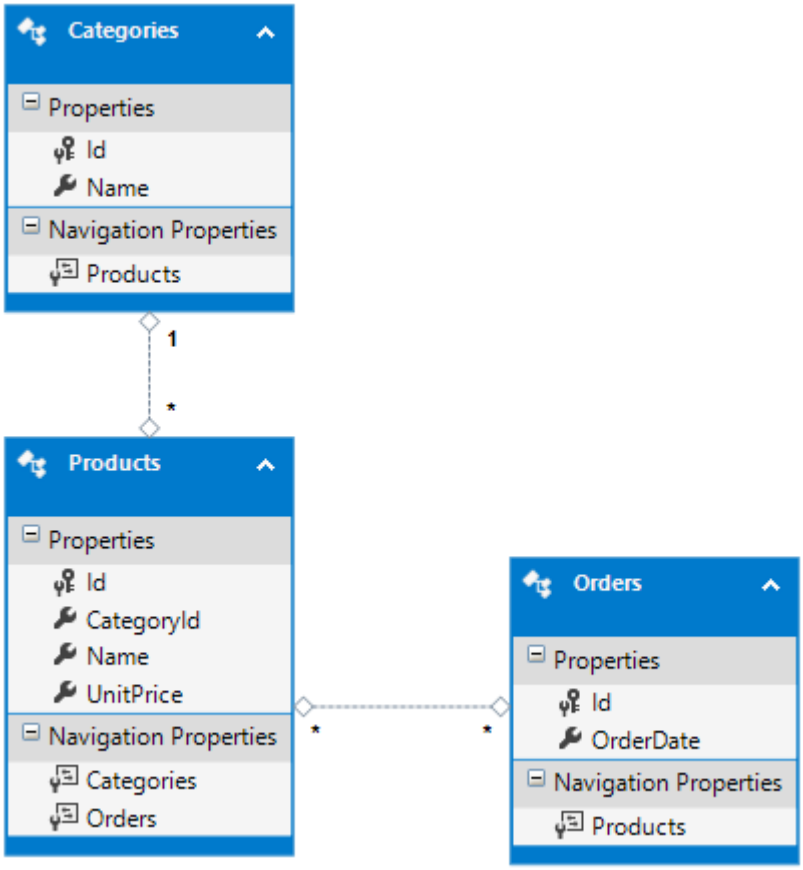
Saját konvenciók

EF Core 7 óta saját konvenciókat is megadhatunk lásd bővebben.

Elsőként azt fogjuk megnézni, hogy mit talál ki az EF, ha semmi plusz adatot nem adunk meg.

Relációk

A fő entitások közötti kapcsolatokat mutatja sematikusan az alábbi ábra:



A relációkat idegen kulcs propertyk és navigációs propertyk reprezentálják. Az idegen kulcs propertyk típusa a kapcsolat másik végén lévő entitás **kulcsának** típusa. A navigációs propertyk típusa pedig a kapcsolat másik végén lévő entitás típusa vagy ilyen típusú kollekció.

Egy konkrét kapcsolat esetében: a `Product` - `Category` egy-többes kapcsolathoz egy idegen kulcs property és egy navigációs property tartozik a `Product` osztályban és egy kollekció típusú navigációs property a `Category` -ban. A többes navigációs property-k legyenek csak olvashatók és a típusuk legyen `ICollection<>` .

Navigációs propertyk

Általánosságban nem kötelező egy kapcsolat mindkét oldalán navigációs property-t vagy külső kulcsot felvenni, de erősen javasolt és mindig jó, ha van. Az entitáson végzendő műveleteket egyszerűsíti, illetve a konvenciós logika is következtet belőle.

A navigációs propertyk referencia típusúak, így foglalkoznunk kell a nullozhatóság kérdésével. Ha a kapcsolat modellezési szempontból nem kötelező (például ha nem várnánk el, hogy minden terméknek legyen megadva a kategóriája), akkor a navigációs property típusa is legyen értelemszerűen nullozható. Ha a kapcsolat kötelező, akkor az ajánlott eljárás, hogy a navigációs property típusa ne legyen nullozható - viszont ekkor kezdeti értéket kell adnunk. Gyakori eset, hogy egy entitást betöltünk adatbázisból, de a hozzá kapcsolódó entitás(oka)t nem, ilyenkor mégis a `null` érték lenne a megfelelő. Emiatt az egyik ajánlott módszer, ha a propertyt **null forgiving** operátorral inicializáljuk `null` értékre.

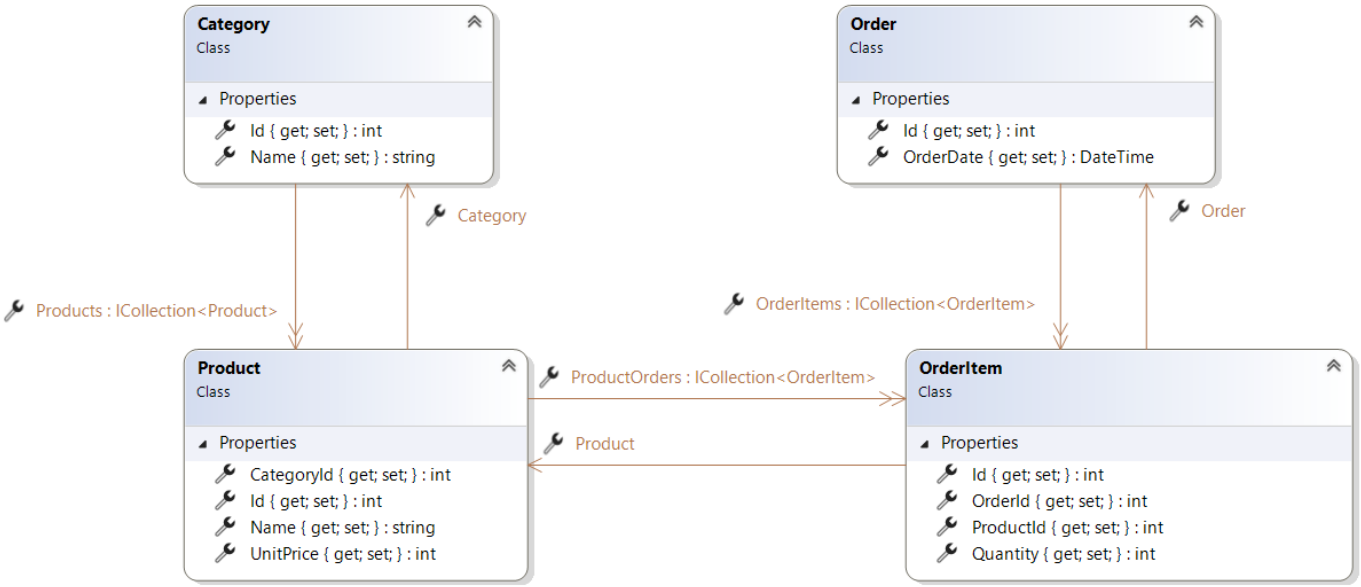
Példa: `public Category Category { get; set; } = null!;` .

Az `Order` - `Product` több-többes kapcsolatokhoz hozzuk létre a kapcsolótáblának megfelelő entitást is, ami egy-egy `Product` és `Order` közötti kapcsolatot reprezentálja.

- `OrderItem` (`Id` : `int` , `ProductId` : `int` , `OrderId` : `int` , `Quantity` : `int`)

> **Kapcsoló entitás**
>
> Nem kötelező létrehozni osztályt a kapcsolótáblának, konfigurációval is lehet érni, hogy a kapcsolótábla létrejöjjön és az EF megfelelően használja. Ezt a módszert akkor érdemes követni, ha a kapcsolótábla csupán technikai tehertétel, de ha például extra adatot is tárol, esetünkben a rendelt mennyiséget (`Quantity`), akkor jobban követhető kódot eredményez, ha explicit létrehozzuk a kapcsolótáblának megfelelő entitástípust.

Az így kialakult modell:



Kódként:

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

    public required string Name { get; set; }

    public ICollection<Product> Products { get; } = new List<Product>();
}

public class Order
{
    public int Id { get; set; }

    public DateTime OrderDate { get; set; }

    public ICollection<OrderItem> OrderItems { get; } = new List<OrderItem>();
}

public class Product
{
    public int Id { get; set; }

    public required string Name { get; set; }
    public int UnitPrice { get; set; }
```

```
    public int CategoryId { get; set; }
    public Category Category { get; set; } = null!;

    public ICollection<OrderItem> ProductOrders { get; } = new List<OrderItem>();
}

public class OrderItem
{
    public int Id { get; set; }

    public int Quantity { get; set; }

    public int ProductId { get; set; }
    public Product Product { get; set; } = null!;

    public int OrderId { get; set; }
    public Order Order { get; set; } = null!;
}
```

Vegyük észre, hogy eddig semmilyen EF specifikus kódot nem írtunk, a modellünk sima ún. *POCO* osztályokból áll.

2.6.3 Kapcsolat az adatbázissal

DbContext - NuGet

Az entitásokat definiáltuk, a mapping-et az EF eszére bíztuk, a következő lépés az adatbázisséma létrehozása a mapping alapján, amit képes az EF migrációs eszköze megoldani. Műveletet az ún. *kontext*-en keresztül tudunk végezni. Érdemes saját kontext típust létrehozni, amit az alap `DbContext`-ből származtatunk. Eddig még nem is írtunk semmilyen EF specifikus kódot, most viszont már kell a `DbContext` típus, így NuGet-ből hozzá kell adnunk a **Microsoft.EntityFrameworkCore.SqlServer** csomagot. Nem ez a csomag tartalmazza a `DbContext`-et, viszont függőségként hivatkozza (**Microsoft.EntityFrameworkCore**).

> **NuGet csomagok telepítése**
>
> A NuGet csomagok telepítéséhez segítség a dokumentációban.

> **NuGet verziók**
>
> Olyan csomagoknál, ahol a verziószámozás követi az alap keretrendszer verziószámozását, törekedjünk arra, hogy a csomagok verziói konzisztensek legyenek egymással és a keretrendszer verziójával is - akkor is, ha egyébként a függőségi szabályok engednék a verziók keverését. Ha a projektünk például .NET 8-os keretrendszert használ, akkor az Entity Framework Core és egyéb extra ASP.NET Core csomagok közül is olyan verziót válasszunk, ahol legalább a főverzió egyezik, tehát valamilyen 8.x verziót. Ez nem azt jelenti, hogy az inkonzisztens verziók mindig hibát eredményeznek, inkább a projekt általában stabilabb, ha a főverziók közötti váltást egyszerre, külön migrációs folyamat (példa) keretében végezzük.

Az Entity Framework önmagában független az adatbázis implementációktól, azokhoz különböző, adatbázisgyártó-specifikus *adatbázis providereken* keresztül kapcsolódik. A **Microsoft.EntityFrameworkCore.SqlServer** csomag hivatkozza az EF absztrakt relációs komponensét (*EntityFrameworkCore.Relational*), és tartalmazza az *MS SQL Server*-hez tartozó providert. A providert a `DbContext` `OnConfiguring` metódusában adhatjuk meg, esetünkben a `UseSqlServer` metódussal, ami egy connection stringet vár.

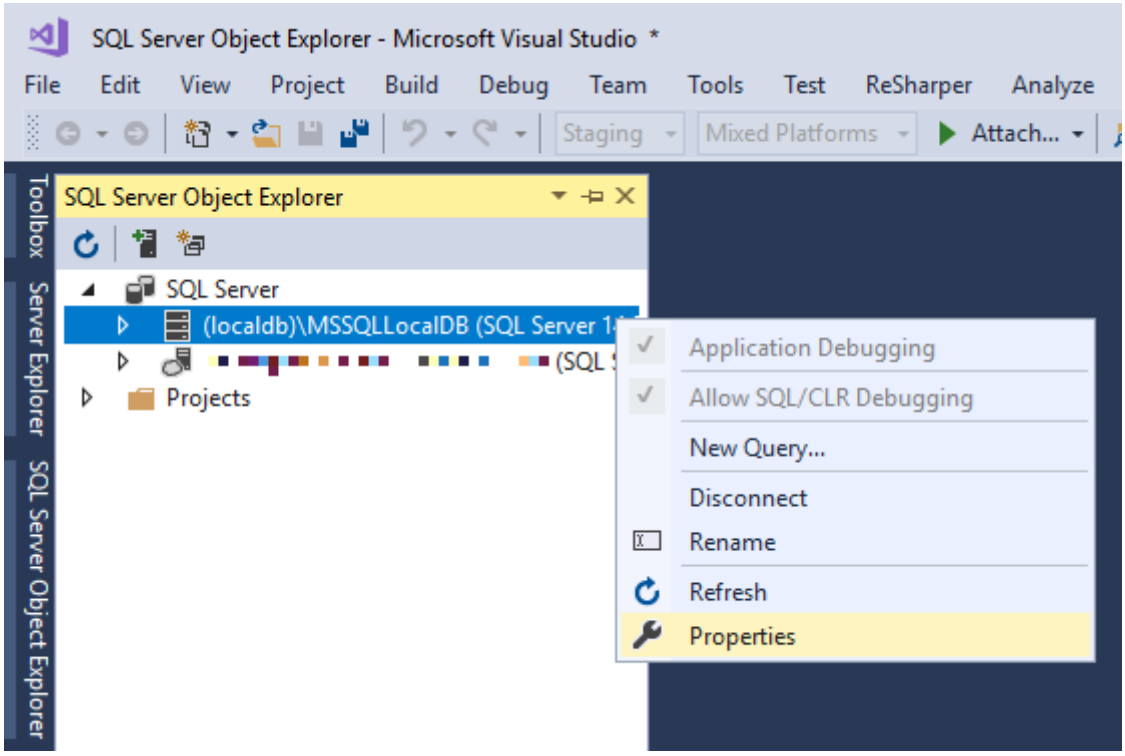
MS SQL Server helyett a *LocalDB* nevű fejlesztői adatbázist használjuk, mely fejlesztői szempontból gyakorlatilag megegyezik az MS SQL Server-rel. A LocalDB a Visual Studio-val együtt települ, minden Windows felhasználónak külön LocalDB példány indítható el. A Visual Studio az *SQL Server Object Explorer* ablak megnyitásakor automatikusan létrehozza a felhasználónkhoz tartozó, *MSSQLLocalDB* nevű példányt.

Localdb

A LocalDB külön is letölthető, illetve a vele együtt települő `sqllocaldb` parancs segítségével egyszerűen kezelhető. Minderről bővebb információ a dokumentációban olvasható.

Adjunk hozzá új osztályt a projekthez `LabDbContext` néven, ebben definiáljuk majd, hogy milyen entitáskollekciókon lehet műveleteket végezni.

Az automatikusan létrejövő MSSQLLocalDB nevű LocalDB példány connection stringjét adjuk meg, pontosabban az *SQL Server Object Explorer* ablak segítésével másoljuk ki: menu:SQL Server-t kibontva[*(localdb)\MSSQLLocalDB*-n jobbklikk > Properties > Connection String]. A kimásolt stringben az *Initial Catalog* értékét (a DB nevét) a **master**-ről változtassuk meg valamilyen más névre, például a Neptun kódunkra. Ha nincs a stringben *Initial Catalog* rész, akkor írjuk a string végére, hogy `;Initial Catalog=neptunkod`.



Connection string megszerzése az SQL Server Object Explorer-ből

Különleges karakterek

A connection stringben különleges karakterek (pl. \) vannak. Ha a kimásolt connection két " közé illesztjük be, a VS automatikusan escape-eli a különleges karaktereket. Ellenkező esetben (ha pl. a két " a beillesztés után kerül elhelyezésre a szöveg köré) az automatikus escape-elés nem történik meg, ilyenkor ne felejtsük el a @-ot a string elé írni, vagy manuálisan escape-elni a szükséges karaktereket!

```
public class LabDbContext : DbContext
{
   protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
   {
      optionsBuilder.UseSqlServer("<connstring>");
   }

   public DbSet<Product> Products => Set<Product>();
   public DbSet<Category> Categories => Set<Category>();
   public DbSet<Order> Orders => Set<Order>();
}
```

Connection String helye

A nagyobb rugalmasság érdekében érdemes a connection stringet konfigurációs fájlba helyezni, majd az ASP.NET Core konfigurációs megoldásaival felolvasni. Erre egy későbbi gyakorlaton nézünk példát.

`DbSet` property-k szerepe

A `DbSet<>` típusú tulajdonságoknak látszólag csak kényelmi funkciójuk van, a `Set<>()` függvényhívásokat egyszerűsítik, azonban valójában nagyobb a jelentőségük. Többek között ezek alapján deríti fel az EF, hogy melyek az entitásosztályok, hiszen alapvetően nincsen semmilyen megkülönböztető jellemzőjük. Alapvetően a `DbSet<>` típusú property-k típusparaméterei és az így felderített entitástípusokban lévő navigációs propertyk típusa alapján áll össze az entitástípusok köre.

Az első verziós adatelérési (DAL) rétegünk ezzel kész is van.

2.6.4 Sémamódosítás

Code-First Migrations

A kódban történő sémamódosításokat követni tudja a keretrendszer, és a változások alapján frissíteni tudja az adatbázis sémáját lefele, illetve felfele irányban is. Ezt a mechanizmust nevezzük migrációnak. Esetünkben a séma nulláról felhúzása is már módosításnak számít.

A migráció elvégzésére parancssoros utasításokat kell igénybe vennünk. Itt kétfajta megközelítés is adott: vannak **PowerShell** és vannak klasszikus **cmd** (dotnet cli) parancsaink. Fel kell telepítsük a projektünkbe valamelyik NuGet csomagot:

- **PowerShell**: Microsoft.EntityFrameworkCore.Tools (telepítsük fel most ezt)
- **Parancssor**: Microsoft.EntityFrameworkCore.Tools.DotNet

Hozzuk elő a **Package Manager Console**-t. (menu:Tools[NuGet Package Manager > Package Manager Console]). Ellenőrizzük, hogy a *Default Project* legördülőben a mi projektünk van-e kiválasztva. Az `Add-Migration <név>` paranccsal tudunk készíteni egy új migrációs lépést, így az első migrációnk a kiinduló sémánk migrációját fogja tartalmazni.

```
Add-Migration Init
```

Figyeljük meg, mit generált a projektünkbe ez a parancs. Itt a migrációhoz egy osztályt készít, ami tartalmazza azokat az utasításokat (`Up` függvény), amikkel a modellünknek megfelelő táblákat fel lehet venni. Emellett külön függvényben (`Down`) olyan utasítások is vannak, melyek ugyanezen táblákat eldobják.

Fordítás után adjuk ki az `Update-Database` parancsot, amivel egy adott migrációs állapotig próbálja frissíteni a sémát. Ha nem adunk meg sémanevet akkor a legfrissebb migrációig frissít:

```
Update-Database Init
```

Localdb migrációs hiba

Bizonyos LocalDB verzióknál hibára futhat az adatbázislétrehozás (*CREATE FILE encountered operating system error 5(Access is denied.)*), mert rossz helyen próbálja létrehozni az adatbázisfájlt. Ilyenkor az SQL Server Object Explorer ablakban bontsuk ki a LocalDB példányunk, alatta a menu:Databases mappán jobbklikk[*Add New Database*]. A megjelenő ablakban adjuk meg névként ugyanazt az adatbázisnevet, amit korábban a connection string-ben a *master* helyett megadtunk.

Ellenőrizzük le az adatbázis sémáját az *SQL Server Object Explorer* ablakban. Nézzük meg, hogy pusztán konvenciók alapján milyen tulajdonságokat talált ki az EF.

Migráció kódból

Kódból is legenerálhatnánk az adatbázist az aktuális sémával a `DbContext.Database.EnsureCreated` metódus segítségével, viszont ez a későbbiekben megnehezíti a további sémamódosítást, mivel mindig el kellene dobjuk az adatbázist, illetve a migrációt sem könnyű utólag bevezetni.

Leképezés és egyéb metaadatok megadása II. — fluent és attribútum alapú leképezés

Definiáljuk felül a kontextünkben az ős `OnModelCreating` metódusát és itt állítsunk be pár mapping információt.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   base.OnModelCreating(modelBuilder);

   modelBuilder.Entity<Category>()
      .Property(c => c.Name)
      .HasMaxLength(15);
}
```

Ezzel a `Name` property hosszát állítottuk be.

A fluent mellett próbáljuk ki az attribútumos konfigurációt is. Állítsunk át egy oszlopnevet a `Product` osztályban a `Column` attribútummal.

```
[Column("ProductName")]
public string Name { get; set; }
```

POCO

A fenti miatt az entitásmodellünk már nem POCO, mert EF specifikus attribútum jelent meg a kódjában.

Többesszámok kezelése

Érdemes megfigyelni a táblanevek kapcsán, hogy eleve többesszámosított neveket találunk az adatbázisban. Ezt az `IPluralizer` service végzi, melyhez saját implementáció is írható.

Mivel már létezik az adatbázisunk, migráció segítségével kell frissítsük az adatbázis sémáját. Készítsünk egy új migrációs lépést az `Add-Migration` utasítással és frissítsük a sémát az `Update-Database` paranccsal.

```
Add-Migration CategoryName_ProductName
Update-Database CategoryName_ProductName
```

Migrációs SQL script

Megnézhetjük az adatbázison futtatott SQL-t is a Script-Migration paranccsal. Például ez mutatja a legutóbbi módosítást érvényesítő SQL-t: `Script-Migration -From Init`

Migráció veszélyei

Természetesen mivel még nincsenek adataink az adatbázisban, akár el is dobhatnánk az adatbázist és újra legenerálhatnánk nulláról a sémát, de most kifejezetten a migrációt szeretnénk gyakorolni. Az `Add-Migration` kimenete figyelmeztet, hogy adatvesztés is történhet. Vannak veszélyes migrációs műveletek, ezért érdemes átnézni a generálódó migrációs kódot.

Migrációk szinkronban tartása

Ha valamilyen okból nem megfelelő a migrációnk, ne töröljük kézzel a generált C# kódfájlokat. Használjuk helyette a `Remove-Migration` parancsot (mindenfajta paraméter nélkül), ami a legutóbbi migrációt törli.

Nézzük meg, milyen migrációs osztályt generáltunk, és hogy ez milyen utasításokat tartalmaz.

Ellenőrizzük, hogy a `Name` oszlop most már az új kényszereknek megfelelően lett-e felvéve, és hogy a terméknév oszlop neve is megváltozott-e.

Ezzel kész a DAL rétegünk konfigurációja, egyúttal mindent kipipáltunk az anyagrész elején lévő felsorolásból.

2.6.5 Adatbázis naplózás

A következő feladat könnyebb követhetősége érdekében állítsuk be a naplózást az Entity Framework kapcsán. A kontext osztályba:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer("<connstring>") // ; törölve
        .LogTo(Console.WriteLine, LogLevel.Information);
}
```

Naplózás máshova

Ha nem a konzolt szeretnénk teleszemetelni, akkor akár a Debug kimenetre (Output ablak) is írhatunk. Ehhez a `LogTo` -nak adjuk meg paraméterként a `m => Debug.WriteLine(m)` delegátot.

2.6.6 Beszúrás

Írjunk egy egyszerű beszúró kódot a `Program.cs` -be. Várjunk paraméterül egy kontext-et, és csak akkor szúrjunk be az adatbázisba bármit, ha még üres.

```
static void SeedDatabase(LabDbContext ctx)
{
    if (ctx.Products.Any())
    {
        return;
    }

    var drink = new Category() { Name = "Ital" };
    var food = new Category() { Name = "Étel" };

    ctx.Categories.Add(drink);
    ctx.Categories.Add(food);

    ctx.Products.Add(new Product() { Name = "Sör", UnitPrice = 50, Category = drink });
    ctx.Products.Add(new Product() { Name = "Bor", Category = drink });
    ctx.Products.Add(new Product() { Name = "Tej", CategoryId = drink.Id });

    ctx.SaveChanges();
}
```

Figyeljük meg, hogy kevertük a kapcsolatok beállításánál a navigációs property szerinti, illetve a sima `Id` érték beállítást.

Hívjuk meg a legfelső szintű kódból és próbáljuk meg lekérdezni az első terméket. Rakjunk a kód végére egy `Console.ReadKey` -t, hogy legyen időnk megnézni a naplót.

```
using var ctx = new LabDbContext();
SeedDatabase(ctx);
var p = ctx.Products.FirstOrDefault();

Console.ReadKey();
```

Próbáljuk ki!

Hibára fut, mert beszúrásnál az `Id` értékes hivatkozás alapértelmezett `int` , azaz 0 értékű lesz, hiszen a kategória is új. Az új elemeknél gyakori, hogy az adatbázis osztja ki az elsődleges kulcs értéket, addig az alapértelmezett értékű. Konvenció szerint a mi `Id` oszlopaink is ilyenek lesznek (ún. IDENTITY oszlopok). A termék beszúrásakor viszont a 0 érték már nem lesz helyes, hiszen addigra a kategória kapott valamilyen kulcs értéket. Mindezt a problémát navigációs property-s hivatkozással elkerülhetjük.

Figyeljük meg a konzol naplóban, hogy a `Category` beszúrása még megtörténik, de az egyik `Product` hozzáadása már elszáll. A debuggerrel, ha megállunk a `SaveChanges` híváson, akkor látható, hogy a `CategoryId` property értéke nulla.

Figyeljük meg azt is, hogy a `SaveChanges` hívásig nem történik módosító adatbázisművelet. Az EF memóriában gyűjti a változásokat, amiket a SaveChanges-szel szinkronizálunk az adatbázisba.

Itt láthatjuk az alapértelmezett tranzakciókezelés működését is. Egy hívásban több elemet kell beszúrni, ha bármelyik művelet meghiúsul, akkor semmilyen változás nem érvényesül az adatbázisban. Általánosan igaz, hogy egy `SaveChanges` vagy minden változást érvényesít vagy semmit sem.

Javítsuk ki:

```
ctx.Products.Add(new Product("Tej")
{
   Name = "Tej",
   Category = drink //navigációs property-re váltottunk
});
ctx.SaveChanges();
```

Ennek már le kell futnia. Nézzük meg a konzolon az SQL utasításokat és a változásokat az adatbázisban. Paraméterezett `INSERT` utasításokat használ az EF, így elkerülve az SQL injection támadást.

Átmeneti azonosítók

A háttérben az EF minden új entitásnak kioszt egy átmeneti azonosítót, amit felhasználhatunk a fenti hiba elkerülésére, ha semmiképp sem akarjuk a navigációs property-ket használni. Így tudnánk a context-től elkérni: `ctx.Entry(cat_drink).Property(e => e.Id).CurrentValue`

Azonosítók kezelése

Ha egyszerre több egymásra hivatkozó elemet szúrunk be és azonosító alapján kötjük őket össze, mindig gondoljuk át, hogy a tényleges adatbázisbeli azonosítók biztosan rendelkezésre állnak-e, mert különben futásidejű kivételt kaphatunk, a fordító nem fog figyelmeztetni.

2.6.7 Ősfeltöltés (seeding) elvárt adattartalom megadásával

A kontextuskonfiguráció részeként megadhatjuk, hogy milyen adattartalmat szeretnénk az egyes táblákban látni. A kontext `OnModelCreating` függvényének végére:

```
modelBuilder.Entity<Category>().HasData(
   new Category() { Name = "Ital", Id = 1 }
);

modelBuilder.Entity<Product>().HasData(
   new Product() { Name = "Sör", Id = 1, UnitPrice = 50, CategoryId = 1 },
   new Product() { Name = "Bor", Id = 2, UnitPrice = 550, CategoryId = 1 },
   new Product() { Name = "Tej", Id = 3, UnitPrice = 260, CategoryId = 1 }
);
```

> **HasData kulcsok**
>
> Fontos, hogy ezen módszer esetén mindenképp kézzel meg kell adnunk az elsődleges kulcs értékeket. Fordítás után generáltassunk új migrációt és frissítsük is az adatbázist - ez utóbbi hibára fog futni:

```
Add-Migration Seed
Update-Database
```

A `HasData` alapján generált migrációs kód nem veszi figyelembe az időközben bekerült adatokat, csak a modellt és a többi migrációt nézi. Ha megnézzük a generált kódot, láthatjuk, hogy csak sima beszúrások. Mivel mi közben jól összeszemeteltük az adatbázist, a migráció által kiadott beszúró műveletek jó eséllyel hibára futnak.

Ha szeretnénk tiszta lappal indulni, bármikor kipucolhatjuk az adatbázist a speciális nullás migrációra való frissítéssel, majd újrahúzhatjuk a `HasData`-nak köszönhetően kezdeti adatokkal ősfeltöltve.

```
Update-Database 0
Update-Database
```

Ezek után a `SeedDatabase` hívásra nincs szükség, kommentezzük ki.

2.6.8 Lekérdezések

Minden rész után az előző szakasz kódját kommentezzük ki, hogy ne keltsen felesleges zajt a kimeneten az előző utasítás, illetve ne legyenek felesleges mellékhatások.

Kérdezzük le azoknak a termékeknek a nevét, melyeknek neve egy adott betűt tartalmaz:

```
//SeedDatabase(ctx);
//var p = ctx.Products.FirstOrDefault();

var q = from p in ctx.Products
      where p.Name.Contains("ö")
      select p.Name;

foreach (var name in q)
{
   Console.WriteLine(name);
}
```

Itt figyelhető meg a korábban már tárgyalt `IEnumerable<>` - `IQueryable<>` különbség. A `Products` property típusa `DbSet`, ami `IQueryable<>`. Az `IQueryable<>`-en történő hívások kifejezésfát (`Expression`) építenek és szintén `IQueryable<>`-t adnak vissza. A `q` értéke egy olyan `IQueryable<>`, ami `Expression`-jében tartalmazza a teljes lekérdezést. Amikor szükség van az adatra, a kifejezésfa alapján SQL generálódik és ez az SQL fut le az adatbázison.

A debuggerrel léptessük át az egyes utasításokon a program futását. A késleltetett kiértékelés miatt csak a `foreach` végrehajtása közben fog az adatbázishoz fordulni az EF, hiszen csak ekkor van ténylegesen szükség az adatra. Nézzük meg a lefuttatott SQL-t is. Sikerült az `IQueryable<>`-ben található `Expression`-t SQL utasítássá alakítania.

A fenti példában az úgynevezett query syntax-t használtuk, de ugyanezt megtehetjük a method vagy fluent syntax-sal is:

```
var q = ctx.Products
   .Where(p => p.Name.Contains("ö"))
   .Select(p => p.Name);
```

A labor során ez lesz a preferált a továbbiakban.

Az EF elég sok C# függvényt SQL-lé tud fordítani. Példaképp alakítsuk a visszaadott nevet nagybetűssé.

```
var q = ctx.Products
   .Where(p => p.Name.Contains("ö"))
   .Select(p => p.Name.ToUpper());
```

Figyeljük meg a konzolon a generált SQL-t: a projekciós részbe bekerült az `UPPER` SQL függvény.

Vegyes kiértékelés

A fák sem nőnek az égig, az EF sem tud minden C# függvényt SQL-lé fordítani. Próbáljuk ki úgy, hogy a `Contains`-t karakterrel hívjuk meg a szűrésben.

```
var q = ctx.Products
   .Where(p => p.Name.Contains('ö'))
   .Select(p => p.Name.ToUpper());
```

`InvalidOperationException`-t kapunk: ezt a lekérdezést nem tudja a provider SQL-lé fordítani. Egyik lehetőségünk, ahogy a hibaüzenet is írja, hogy kikényszerítjük a kiértékelést a nem leforduló művelet elé helyezett `AsEnumerable` vagy `ToList` (illetve ezek aszinkron változatai) hívással. Próbáljuk ki - mivel a szűrést nem sikerült átfordítani, a szűrés elé a from végére tegyük az `AsEnumerable`-t:

```
var q = ctx.Products
   .AsEnumerable()
   .Where(p => p.Name.Contains('ö'))
   .Select(p => p.Name.ToUpper());
```

Ez működik, de a konzolon megjelenő SQL utasításon látszik, hogy a **teljes** termék táblát lekérdeztük és felolvastuk a memóriába. Az `AsEnumerable` jelentése: a lekérdezés innentől LINQ-to-Objects-ként épül tovább, a lekérdezés eddigi részének memóriabeli reprezentációja lesz az adatforrás, tehát a szűrés és a projekció már memóriában fut le. Mivel a teljes lekérdezés egy része LINQ-to-Entities (adatbázis értékeli ki), a másik része LINQ-to-Objects (a .NET runtime értékeli ki), az ilyen lekérdezéseket ún. vegyes kiértékelésűnek (*mixed evaluation*), a LINQ-to-Objects részt kliensoldali kiértékelésűnek (*client evaluation*) nevezik. A `q` típusa ebben az esetben már nem `IQueryable<>`, csak `IEnumerable<>`.

> **IEnumerable és IQueryable különbségének felderítése**
>
> Érdemes összevetni a `Where` függvény definícióját (kurzorral ráállva `F12` vagy menu:jobbklikk[Go To Definition]) a két változatnál. Az első esetben `IQueryable` az adatforrás és `Expression` a feltétel, a másodiknál `IEnumerable` az adatforrás és sima delegate a feltétel.

Vegyes kiértékelés veszélyei

A vegyes kiértékelés veszélyes lehet, mert a szűrés és a projekció már memóriában fut le, azaz az adatbázisban lévő adatokat a memóriába kell olvasni, ami nagy adatmennyiség esetén lassú és erőforrásigényes lehet.

Különösen fontos, hogy lehetőleg minden EF lekérdezésünket ellenőrizzük le, hogy minden része ott fut-e le (adatbázisban vagy memóriában), ahol számítunk rá.

Másik lehetőség, ha keretrendszer korlátba ütközünk, hogy a lekérdezést megpróbáljuk úgy átírni, hogy minél nagyobb része lefuttatható legyen adatbázisban. Ez a konkrét példában egyszerű, csak vissza kell írni az első változatot.

Lekérdezések összefűzése és címkézése

Kérdezzük le egy bizonyos árnál drágább, bizonyos betűt a nevükben tartalmazó termékek nevét - mindezt két külön lekérdezésben:

```
var q1 = ctx.Products.TagWith("Névszűrés")
   .Where(p => p.Name.Contains("r"));

var q2 = ctx.Products
   .Where(p => p.UnitPrice > 20)
   .Select(p => p.Name);

foreach (var name in q2)
{
   Console.WriteLine(name);
}
```

A `TagWith` használatával könnyebben megtalálhatjuk a lekérdezés által generált SQL utasítást a naplóban: a függvénynek megadott szöveg közvetlenül a generált utasítás elé kerül.

Ismét figyeljük meg a naplóban, mikor fut le és milyen lekérdezés. Itt is látszik a késleltetett kiértékelés és a lekérdezések össze lesznek fűzve, egy lekérdezés hajtódik végre.

Lekérdezések összefűzése

Ez rámutat az EF egy nagy előnyére: bonyolult lekérdezéseket megírhatunk kisebb, egyszerűbb részletekben, az EF pedig összevonja, sőt optimalizálhatja is a teljes lekérdezést.

Próbáljuk ki, `var q =` helyett `IEnumerable<Product> q =` -val is, ilyenkor nem fűzi össze a lekérdezést. A `q2` műveletei már memóriában fognak lefutni, hiszen a `q2` adatforrásként csak egy `IEnumerable` -t lát.

Próbáljuk ki, `var q =` helyett `IQueryable<Product> q =` -val is, ilyenkor megint összefűzi a lekérdezést.

Itt is érdemes összevetni a `where` operátor definícióját a két lekérdezésrészben.

> ⚠ **`IQueryable` és `IEnumberable` különbségei újra**
>
> Nem lehet elégszer hangsúlyozni az `IQueryable` és az `IEnumerable` közti különbségeket. Az `IQueryable` kifejezések SQL-lé fordulnak (amikor le tudnak), míg az `IEnumerable`-en végzett műveletek minden esetben memóriában hajtódnak végre.
>
> Ha nem akarunk véletlenül memóriabeli kiértékelésre váltani, az implicit típus (`var`) alkalmazása jó szolgálatot tehet.

2.6.9 Beszúrás több-többes kapcsolatba

Azokat a termékeket szeretnénk megrendelni, amiknek a nevében van egy adott betű. Használjuk fel újra az előző, hasonló lekérdezésünket.

```
var products = ctx.Products
   .Where(p => p.Name.Contains("r"))
   .ToList();

var order = new Order { OrderDate = DateTime.Now };

foreach (var p in products)
{
   order.OrderItems.Add(
      new OrderItem { Product = p, Order = order, Quantity=2 }
   );
}

ctx.Orders.Add(order);
ctx.SaveChanges();
```

Ismét figyeljük, hogy milyen SQL generálódik. Az `Order` létrehozása után nekünk még egy új `OrderItem` entitást is létre kell hoznunk, amit a több-több kapcsolatra használunk fel. Figyeljük meg, hogy nem kellett minden `OrderItem`-et külön-külön hozzáadnunk a kontextushoz, az `Order` hozzáadásával minden `OrderItem` is bekerült a kontextusba, majd el is mentődött az adatbázisba.

2.6.10 Kapcsolódó entitások betöltése

Írjuk ki minden termék neve mellé a kategóriáját is.

```
var products = ctx.Products.ToList();

foreach (var p in products)
{
   Console.WriteLine($"{p.Name} ({p.Category.Name})");
}
```

Figyeljük meg, hogy a fenti lekérdezésben a kategória navigációs property `null` értékű és kivétel is keletkezik, pedig biztosan tartozik a termékhez kategória az adatbázisban. Ennek oka, hogy az EF alapból **nem** tölti be a navigációs property-k értékeit, ezt egy külön `Include` metódushívással tudjuk megtenni az `IQueryable` típuson. Ez az ún. *eager loading*.

```
var products = ctx.Products
   .Include(p => p.Category)
   .ToList();
```

Ismét figyeljük, hogy mikor mi fut le az adatbázisszerveren: ez egy `JOIN` segítségével egy füst alatt beránt minden adatot mindkét táblából.

Ha a kapcsolódó `Order` listát is szeretnénk kitöltetni, akkor ott egyrészt a `ProductOrders` listát is be kell `Include`-olni, másrészt pedig még egy kapcsolattal továbbmenve a `OrderItem` `Order` tulajdonságát is be kell töltetni. Az ilyen többszintes hivatkozást az `Include` és `ThenInclude` használatával lehet elérni:

```
var products = ctx.Products
   .Include(p => p.Category)
   .Include(p => p.ProductOrders)
      .ThenInclude(po => po.Order)
   .ToList();

foreach (var p in products)
{
   Console.WriteLine($"{p.Name} ({p.Category.Name})");
   foreach (var po in p.ProductOrders)
   {
      Console.WriteLine($"\tRendelés: {po.Order.OrderDate}");
   }
}
```

Ha nem akarunk minden oszlopot lekérdezni az összes érintett táblából, akkor a projekciós (`select`) részt úgy is megírhatjuk, hogy csak a szükséges adatokat kérdezze le, ez az ún. *query result shaping*.

```
var products = ctx.Products
   .Select(p => new
   {
      ProductName = p.Name,
      CategoryName = p.Category.Name,
      OrderDates = p.ProductOrders
         .Select(po => po.Order.OrderDate)
         .ToList(),
   })
   .ToList();

foreach (var p in products)
{
   Console.WriteLine($"{p.ProductName} ({p.CategoryName})");
   foreach (var po in p.OrderDates)
   {
      Console.WriteLine($"\tRendelés: {po}");
   }
}
```

Figyeljük meg, hogy a generálódó `SELECT` projekciós része így jóval rövidebb.

Lazy Loading

További ritkábban alkalmazott / korábbi verziókban elterjedt módszerek: *explicit loading*, *lazy loading*.

2.6.11 Több-többes kapcsolat közvetlen navigálása

Lehetőség van több-többes kapcsolat navigálásakor a kapcsolótábla átugrására. Ehhez vegyünk fel ennek megfelelő property-ket a kapcsolat két oldalán. Az `Order`-be:

```
public ICollection<Product> Products { get; } = new List<Product>();
```

A `Product`-ba:

```
public ICollection<Order> Orders { get; } = new List<Order>();
```

A kontext `OnModelCreating` -jében konfigurálnunk kell a több-többes kapcsolatban részt vevő minden property-t, hogy az EF tudja, hogy ez az összes property ugyanazon kapcsolathoz tartozik:

```
modelBuilder.Entity<Product>()
    .HasMany(p => p.Orders)
    .WithMany(o => o.Products)
    .UsingEntity<OrderItem>(
        j => j
            .HasOne(oi => oi.Order)
            .WithMany(o => o.OrderItems)
            .HasForeignKey(oi => oi.OrderId),
        j => j
            .HasOne(oi => oi.Product)
            .WithMany(p => p.ProductOrders)
            .HasForeignKey(oi => oi.ProductId),
        j =>
        {
            j.HasKey(oi => oi.Id);
        });
```

Bonyolultnak tűnik, de inkább csak hosszú, míg mind a 9 érintett property szerepét beállítjuk.

Mindennek nem szabadna adatbázis változást okoznia, hiszen nem lett több kapcsolat, csak egy logikai útrövidítést vettünk fel. Ellenőrizzük le:

```
Add-Migration NxN
```

Ha mindent jól csináltunk, ennek egy üres migrációt kell generálnia. Töröljük is.

```
Remove-Migration
```

Ezután a korábbi lekérdezésünknél elhagyhatjuk az `OrderItem` betöltését.

```
var products = ctx.Products
    .Include(p => p.Category)
    .Include(p => p.Orders)
    .ToList();

foreach (var p in products)
{
    Console.WriteLine($"{p.Name} ({p.Category.Name})");
    foreach (var po in p.ProductOrders)
    {
        Console.WriteLine($"\tRendelés: {po.Order.OrderDate}");
    }
}
```

> **Warning**
>
> Ettől nem feltétlenül lesz egyszerűbb vagy gyorsabb a generált lekérdezés, csak a kódunk lesz egyszerűbb.

2.6.12 Módosítás, Find

Nézzünk példát egyszerű módosításra.

```
var pFirst = ctx.Products.Find(1);
if (pFirst != null)
{
    Console.WriteLine(ctx.Entry(pFirst).State);
```

```
    pFirst.UnitPrice *= 2;
    Console.WriteLine(ctx.Entry(pFirst).State);
    ctx.SaveChanges();
    Console.WriteLine(ctx.Entry(pFirst).State);
}
```

Debuggerrel sorról sorra lépkedve kövessük végig az EF változáskövető működését. A lekérdezések eredménye alapértelmezetten bekerül a változáskövetőbe (change tracker). Ezután az osztályon végezhetünk adatváltoztató műveletet, mindig könnyen eldönthető, hogy volt-e változás, ha összevetjük az aktuális állapotot (current value) a bekerüléskorival (original value). Figyeljük meg, hogyan kezeli az EF a hozzá tartozó objektumok állapotát.

Change tracker Entry adatai

Az `Entry` által adott osztályból megtudhatjuk az aktuális és a bekerüléskori értékeket az `OriginalValues` és `CurrentValues` propertyk által, amit akár mi is felhasználhatunk saját change tracking logikához: pl.: audit napló.

Find

A `Find` az elsődleges kulcs alapján keres ki egy entitást. Nem kell ismernünk az elsődleges kulcs property nevét. Ha a változáskövetőbe már korábban bekerült a keresett entitás, akkor onnan kapjuk vissza, ilyenkor adatbázishozzáférés nem történik.

2.6.13 Törlés

Töröljük ki az adatbázisból az egyik megrendelést.

```
var orderToRemove = ctx.Orders
    .OrderBy(o => o.OrderDate)
    .First();

ctx.Orders.Remove(orderToRemove);
ctx.SaveChanges();
```

Figyeljük meg az adatbázis adatai között, hogy az `Order` törlésével a kapcsolódó `OrderItem` bejegyzések is törlődtek, mivel alapértelmezetten a sémán be van kapcsolva a **kaszkád** törlés. Ez ebben az esetben indokolt is lenne, de sokszor nem szeretnénk, ha a kapcsolódó rekordok is törlődnének. Ennek megakadályozására vegyük fel explicit a konfigurációban az `Order-OrderItem` kapcsolatot és kapcsoljuk ki rajta a kaszkád törlést az `OnModelCreating`-ben.

```
j => j
    .HasOne(oi => oi.Order)
    .WithMany(o => o.OrderItems)
    .HasForeignKey(oi => oi.OrderId) // , törölve
    .OnDelete(DeleteBehavior.Restrict),
```

A törölt `Order`-t és a szükséges kapcsoló rekordokat vegyük fel migráció által beszúrt adatként. Az `OnModelCreating` végére:

```
modelBuilder.Entity<Order>().HasData(
    new Order { Id = 1, OrderDate = new DateTime(2019, 02, 01) }
);

modelBuilder.Entity<OrderItem>().HasData(
    new OrderItem { Id = 1, OrderId = 1, ProductId = 1 },
```

```
    new OrderItem { Id = 2, OrderId = 1, ProductId = 2 }
);
```

Fordítás után ne felejtsük el migrációval átvezetni az adatbázis sémájába is a változásokat, mivel a kaszkád törlés egy MSSQL funkció nem EF viselkedés.

```
Add-Migration ProductOrderRestrictDelete
Update-Database 0
Update-Database
```

Futtassuk újra a törlő kódot - kivételt kapunk, mivel az `OrderItem` rekord nem törlődött kaszkád módon, így az egy már nem létező `Order`-re hivatkozik, viszont ez a külső kulcs kényszert megsérti. Emiatt az egész törlési művelet meghiúsul.

> **Soft delete**
>
> Adatkezelő alkalmazásokban az adatbázisbeli törlés (SQL `DELETE` utasítás) helyett gyakran inkább logikai törlést (*soft delete*) alkalmaznak. A logikai törlés megvalósításával ezen gyakorlat keretében nem foglalkozunk, de egyszerűen megvalósítható a SaveChanges felüldefiniálásával, abban a change tracker figyelésével és lekérdezéskor Global Query Filter használatával.

2.6.14 Felsorolt típus, értékkonvertálók

Az EF alapértelmezetten képes a felsorolt típusokat is leképezni. Hozzunk létre új felsorolt típust a `Product` osztály mellé `ShipmentRegion` néven.

```
[Flags]
public enum ShipmentRegion
{
    EU = 1,
    NorthAmerica = 2,
    Asia = 4,
    Australia = 8
}
```

A `Flags` attribútummal azt jelezzük, hogy szeretnénk a bitműveleteket is alkalmazni a felsorolt értékére, így egy `ShipmentRegion` típusú változó egyszerre több értéket is felvehet (pl.: 3-as érték egyszerre tartalmazza az EU-t és Észak-Amerikát is).

Vegyünk fel a `Product` osztályba egy új property-t az új felsorolt típussal.

```
public ShipmentRegion? ShipmentRegion { get; set; }
```

Módosítsuk és bővítsük a kezdeti `Product`-ok listáját szállítási információkkal:

```
modelBuilder.Entity<Product>().HasData(
    new Product()
    {
        Id = 1,
        Name = "Sör",
        UnitPrice = 50,
        CategoryId = 1,
        ShipmentRegion = ShipmentRegion.Asia
    },
    new Product() { Id = 2, Name = "Bor", UnitPrice = 550, CategoryId = 1 },
    new Product() { Id = 3, Name = "Tej", UnitPrice = 260, CategoryId = 1 },
```

```
    new Product()
    {
        Id = 4,
        Name = "Whiskey",
        UnitPrice = 960,
        CategoryId = 1,
        ShipmentRegion = ShipmentRegion.Australia
    },
    new Product()
    {
        Id = 5,
        Name = "Rum"
        UnitPrice = 960,
        CategoryId = 1,
        ShipmentRegion = ShipmentRegion.EU | ShipmentRegion.NorthAmerica
    }
);
```

Figyeljük meg a generált migrációban, hogy milyen ügyesen lekezeli az EF a korábbi migrációban beszúrt elem (1-es `Id`) változását, módosító kódot generál hozzá.

Változott a modell, frissítsük az adatbázist.

```
Add-Migration ProductShipmentRegion
Update-Database
```

Figyeljük meg, hogy az új oszlop egész számként tárolja a felsorolt típus értékeit. Ha ez nem tetszik nekünk, mert például szövegesen szeretnénk az adatbázisban látni az értékeket, használhatjuk az értékkonvertálókat (*value converter*), melyek az adatbázis- és az objektummodell között képesek oda-vissza konvertálni a leképezett elemek értékeit. Számos beépített konvertáló van az EF-ben, melyek közül a leggyakoribbakat automatikusan alkalmaz is az EF, elég csak a céltípust megadnunk. Az **felsorolt típus** - **szöveg** átalakító is ilyen. Az `OnModelCreating`-be:

```
modelBuilder
  .Entity<Product>()
  .Property(e => e.ShipmentRegion)
  .HasConversion<string>();
```

Változott a modell, frissítsük az adatbázist.

```
Add-Migration ProductShipmentRegionAsString
Update-Database
```

> **Migrációk helyességének ellenőrzése**
>
> Ahogy a migráció generálásakor a figyelmeztetés is írja, ellenőrizzük a migrációt, mert olyan oszlop típusát változtatjuk, amiben vannak már adatok, ez pedig különös körültekintést igényel. A generált migráció nem is tökéletes, a `Down` részben előbb állítja a migráció `nvarchar`-ról `int`-re az oszlop típusát, minthogy a szöveges értéket számra (pontosabban számot tartalmazó szövegre) cserélné - így lefelé migráláskor SQL hibát kapunk. Az `UpdateData` hívás `AlterColumn` hívás elé helyezésével ezt javíthatjuk.

Ellenőrizzük a termékek táblájában, hogy sikerült-e az átalakítás. Kipróbálhatjuk, hogy működik-e a konverzió objektummodell szinten is. A legfelső szintű kódban kérjük el az összes terméket:

```
var prods = ctx.Products.ToList();
```

Vizsgáljuk meg a listában lévő termékeket debuggerrel: látható, hogy a szállítási területek megfelelő értékűek.

Értékkonvertálók

Explicit is megadhatjuk az alkalmazandó konvertert, ami leggyakrabban a számos beépített konverter közül kerül ki. Saját konvertereket is írhatunk, ha a beépítettek között nem találunk megfelelőt.

2.6.15 Tranzakciók

Az EF az egyes `SaveChanges` hívásokat egy tranzakcióban futtatja (ha az adatbázis provider támogatja azt). Viszont gyakran megesik az, hogy több `SaveChanges` hívást kellene egy tranzakcióban kezelnünk. Tehát ha az egyik sikertelenül fut le, akkor a többit sem szabad érvényre juttatni.

Nézzünk példát a tranzakciókezelésre. Szúrjunk be több terméket az adatbázisba több `SaveChanges` hívással.

```
int cid = ctx.Categories.First().Id;
try
{
  using (var transaction = ctx.Database.BeginTransaction())
  {
    ctx.Products.Add(new Product()
    {
      CategoryId = cid,
      Name = "Coca Cola",
    });
    ctx.SaveChanges();

    ctx.Products.Add(new Product()
    {
      CategoryId = cid,
      Name = "Pepsi",
    });
    ctx.SaveChanges();

    transaction.Commit();
  }
}
catch (Exception)
{
}
```

A tranzakciók kezdete-végével kapcsolatos események csak a debug szintű naplóban jelennek meg. Állítsuk át a naplózási szintet a `LogTo` függvényben:

```
.LogTo(Console.WriteLine, LogLevel.Debug); // LogLevel módosítva
```

A tranzakción `Commit`-ot hívunk, ha sikeresen lefutott mindegyik `SaveChanges`, ha valamelyik hibára futott, akkor a using blokkból való kilépésig nem fog `Commit` hívódni. Ha bármilyen ok miatt a `Commit` nem hívódik meg, legkésőbb a using blokk vége `Rollback`-kel **lezárja a tranzakciót**.

Próbáljuk ki! Ezesetben helyesen fut le a beszúrásunk. Figyeljük meg a konzolon a tranzakciókezeléssel kapcsolatos üzeneteket.

Teszteljük a hibás ágat is azáltal, hogy a második terméket egy nem létező kategóriába próbáljuk meg beszúrni.

```
using (var transaction = ctx.Database.BeginTransaction())
{
  ctx.Products.Add(new Product()
  {
    Name = "Cider", // új név
    CategoryId = cid,
  });
```

```
    ctx.SaveChanges();
    ctx.Products.Add(new Product()
    {
        Name = "Kőműves Actimel", //új név
        CategoryId = 100, //nem létező CategoryId
    });
    ctx.SaveChanges();
    transaction.Commit();
}
```

Figyeljük meg, hogy ilyenkor nem kerül beszúrásra az első termék sem. Úgyszintén figyeljük meg a konzolon a tranzakciókezeléssel kapcsolatos üzeneteket.

2.7 ASP.NET Core webszolgáltatások I.-II.

2.7.1 Kiinduló projektek beüzemelése

Klónozzuk le a publikus kiinduló projektet a GitHub-ról az alábbi paranccsal:

```
git clone https://github.com/bmeviauav23/WebApiLab-kiindulo.git
```

A kiinduló solution két .NET 8 osztálykönyvtárat foglal magába, melyek egy N-rétegű architektúra egy-egy rétegét valósítják meg:

- **WebApiLab.Dal**: lényegében az Entity Framework gyakorlatok anyagát tartalmazza, ez az adatelérési rétegünk.
  - entitásdefiníciók
  - kontext, modellkonfigurációval, kezdeti adatokkal
  - connection string kezelés és SQL naplózás a korábbi gyakorlatok alapján
  - migráció (még) nincs
- **WebApiLab.Bll**: ezt szánjuk az üzleti logikai rétegnek. Fő feladata, hogy a DAL-ra építve végrehajtsa az `Interfaces` mappában definiált műveleteket.
  - Interfaces - ez a BLL réteg specifikációja
  - Services - ide kerülnek majd az üzleti logikát, ill. az interfészeket megvalósító osztály(ok)
  - Dtos - csak később lesz szerepük, egyelőre nincsenek használva
  - Exceptions - saját kivétel osztály, egyelőre nincs használva

Adjunk hozzá a solution-höz egy új C# nyelvű web API projektet (ASP.NET Core Web **API**, nem pedig Web App), a neve legyen `WebApiLab.Api` .

A következő dialógusablakban válasszuk ki a *.NET 8* opciót. Az extrák közül **ne** kérjük ezeket: HTTPS, Docker, authentikáció. Viszont hagyjuk bepipálva a Controller és az OpenAPI támogatást. A generált projektből törölhetjük a minta API fájljait, azaz a `Weather` kezdetű fájlokat a projekt gyökeréből és a `Controllers` mappából.

Adjuk hozzá függőségként:

- a *BLL* projektet (menu:projekten jobbklikk[Dependencies > Add Project Reference…])
- a *Microsoft.EntityFrameworkCore.Tools* NuGet csomagot. Válasszunk olyan verziót, ami egyezik a DAL projekt Entity Framework Core függőségének verziójával.

> **Warning**
>
> Olyan csomagoknál, ahol a verziószámozás követi az alap keretrendszer verziószámozását, törekedjünk arra, hogy a csomagok verziói konzisztensek legyenek egymással és a keretrendszer verziójával is - akkor is, ha egyébként a függőségi szabályok engednék a verziók keverését. Ha a projektünk például .NET 8-os keretrendszert használ, akkor az Entity Framework Core és egyéb extra ASP.NET Core csomagok közül is olyan verziót válasszunk, ahol legalább a főverzió egyezik, tehát valamilyen 8.x verziót. Ez nem azt jelenti, hogy az inkonzisztens verziók mindig hibát eredményeznek, inkább a projekt általában stabilabb, ha a főverziók közötti váltást egyszerre, külön migrációs folyamat (példa) keretében végezzük.

2.7.2 Az EF bekötése az ASP.NET Core DI, naplózó, konfiguráló rendszereibe

A kontext konfigurálása az EF gyakorlat során - mivel ott egy sima konzol alkalmazást írtunk - a kontext `OnConfiguring` függvényében történt. Mivel az ASP.NET Core projekt beépítetten DI konténert is ad a számunkra, érdemes a kontextet a DI rendszerbe regisztrálni, hogy a projekten belül a modulok/osztályok függőségként tudják használni. A regisztrálás a legfelső szintű kódban történik (lásd ASP.NET Core bevezető gyakorlatot).

A kontext regisztrálása a legfelső szintű kódban a DI konténerbe:

```
builder.Services.AddDbContext<AppDbContext>(o =>
  o.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Az EF naplózást az ASP.NET Core naplózó rendszere végzi, amit a kiinduló builder már inicializál, így ezzel kapcsolatban nincs teendőnk. Viszont egy új kontext konstruktorra lesz szükségünk, ami `DbContextOptions<AppDbContext>` -et vár.

A kontext `OnConfiguring` -jára pedig nincs szükség, úgyhogy töröljük ki, helyére tegyük az új konstruktort:

```
public AppDbContext(DbContextOptions<AppDbContext> options)
  : base(options)
{
}
```

Az Entity Framework gyakorlat alapján hozzunk létre egy új LocalDB adatbázist egy választott névvel, pl. neptun kód, northwind, stb. Az SQL Server Object Explorer-ből a connection string-et lopjuk el. (menu:nyissuk le az adatbáziskapcsolatot[jobbklikk az adatbázison > Properties > a Properties ablakból a *Connection String* értéke]).

Az *appsettings.Development.json*-ba vegyük fel a connection string-et és a generált SQL megfigyeléséhez a *Microsoft* kategóriájú naplók minimum szintjét csökkentsük *Information*-re.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Information",
    }
  }, //vessző bekerült
  "ConnectionStrings": {
    "DefaultConnection": "<connection string>"
  }
}
```

Escapelt karakterek

Kukac (@) ilyenkor nem kell a connection string elé, mert ez JSON.

A connection string különleges karaktereit a beillesztés után a VS alapesetben automatikusan escape-eli. Ha az automatikus escape-elés mégsem történik meg, manuálisan kell ezt megtennünk, különben *A network-related or instance-specific error occurred while establishing a connection to SQL Server* hibát kaphatunk.

Az adatbáziskapcsolatot azért kellhet lenyitni, hogy az SQL Server Object Explorer csatlakozzon is az új adatbázishoz, ezután tudjuk megszerezni a connection stringet.

Adatbázis inicializálása Code-First migrációval

Fordítsuk a teljes solution-t, állítsuk be indítandó (startup) projektnek az új Web API projektet (menu:jobbklikk a projekten[Set as Startup Project]). A *Package Manager Console*-t nyissuk meg, és állítsuk be Default Project-ként a DAL projektet.

.NET 8-ban a migrációk csak akkor működnek rendesen, ha az API projekten kikapcsoljuk az InvariantGlobalization beállítást. Ezt a *WebApiLab.Api.csproj* fájlban tehetjük meg:

```
<InvariantGlobalization>false</InvariantGlobalization>
```

Készíttessük el a migrációt és futtassuk is le:

```
Add-Migration Init
Update-Database
```

> **Projektek a migrációhoz**
>
> Fontos, hogy a fenti parancs két projektet ismerjen: azt, amelyikben a kontext van, ill. a kontextet használó futtatható projektet. A VS Package Manager Console-jában futtatva alapértelmezésben az előbbit a Default Project értéke adja meg, utóbbit az indítandó projekt. Továbbá ezeket a projekteket meg lehet adni paraméterként is.

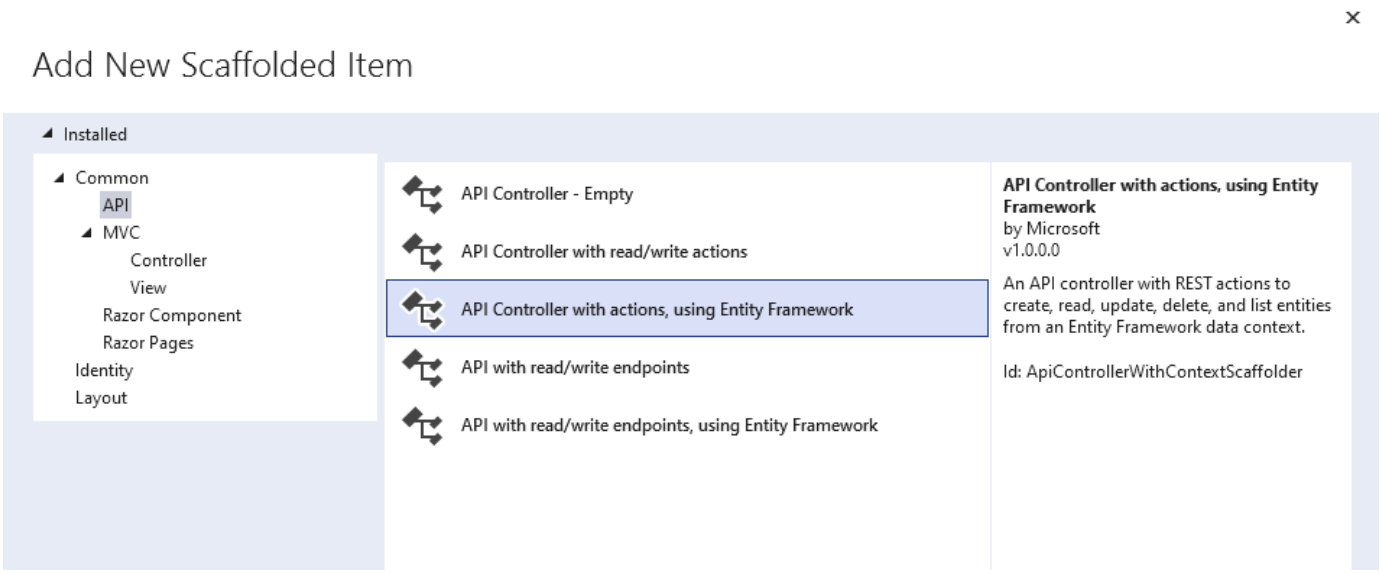
> **Migráció során lefut a Program.cs is?**
>
> Igen, itt mutatkozik meg, hogy a migráció lényegében egy teljes alkalmazásindítást jelent a `Program` osztályon keresztül: inicializálódik a DI konténer, a konfigurációs objektum stb.

Ellenőrizzük az SQL Server Object Explorer-ben, hogy rendben lefutott-e a migráció, létrejöttek-e az adatbázis objektumok, feltöltődtek-e a táblák.

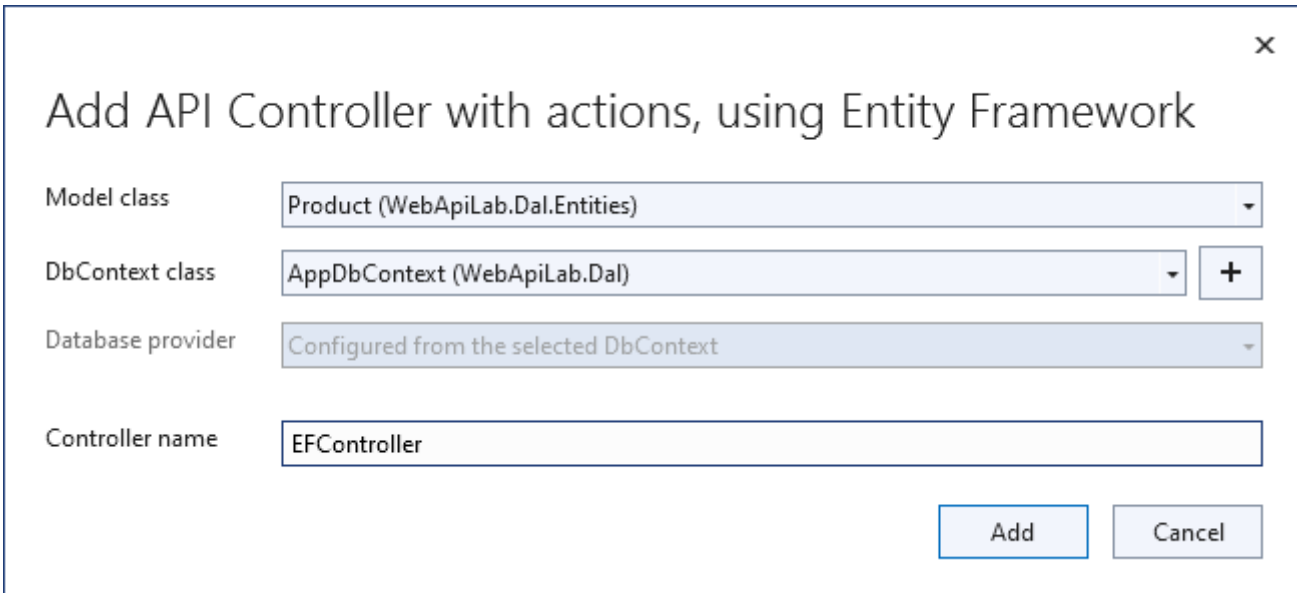
2.7.3 EF entitások használata az API felületen

Bár architektúra szempontból nem a legszebb, a BLL réteget gyakorlatilag mellőzve közvetlenül is használhatjuk az EF entitásokat a kontrollerek megvalósításánál. Ehhez használhatjuk a Visual Studio Entity Framework-ös Controller sablonjait, amit most csak azért használunk, hogy gyorsan legyen egy működő API felületünk.

Adjunk hozzá egy új Controllert a *Controllers* mappába (menu:Add[Controller > bal fában Common > API > jobb oldalon API Controller with read/write actions]) `EFProductController` néven.



Válasszuk ki az `AppDbContext`-t, és a `Product` entitást. Az új kontrollerben már láthatjuk is a scaffoldolt CRUD műveleteket.



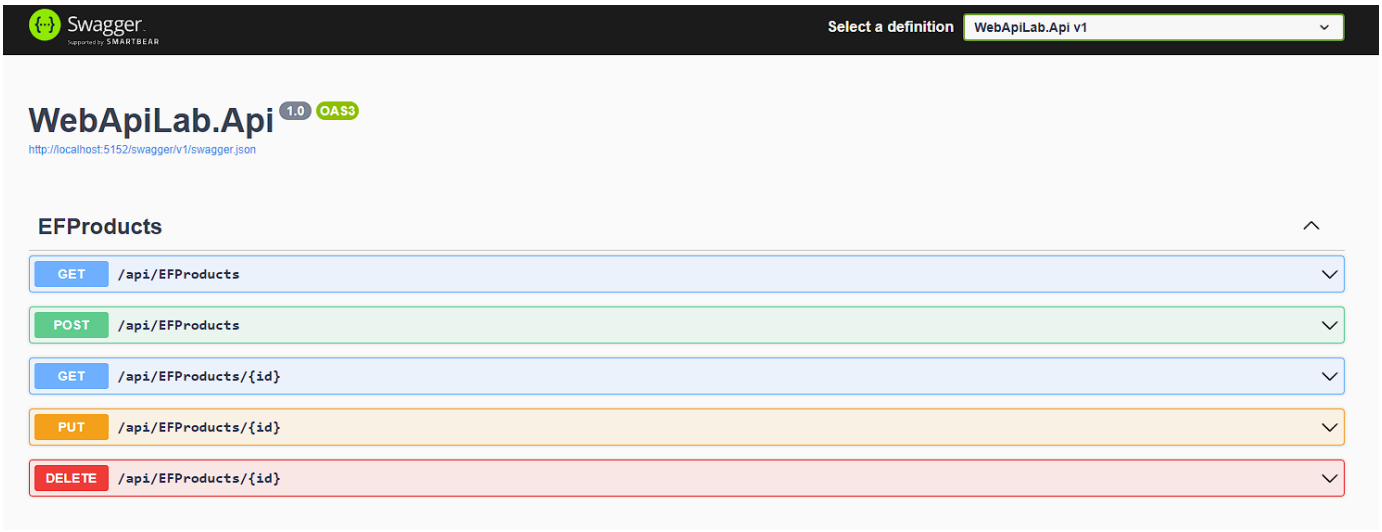
névterek

Figyeljünk rá, hogy **ne** a `Dtos` névtérből adjuk meg a DTO típust a tényleges entitástípus helyett.

Generálás hibára fut

A generálás során *Unable to create an object of type* `AppDbContext`. hibát kaphatunk. A hiba a kódgeneráló eszközben keresendő, a kapcsolódó GitHub issue-ban találunk egy lehetséges megoldást is a problémára, ami elő van készítve a kiinduló projektben is.

A legenerálódó kontroller már használható is. Állítsuk át a zöld nyíl mellett az indítási konfigurációt a projektnevesre, hogy ne IIS Express induljon és így lássuk a konzolon a naplót. Indítsuk a projektet és próbáljuk például lekérni az összes terméket az **api/efproduct** címről vagy a Swagger felületről.



Böngésző választása debugoláshoz

Érdemes a zöld nyíl melletti lenyíló menüben olyan böngészőt megadni (Chrome, Firefox), ami értelmes formában meg tudja jeleníteni a nyers JSON adatokat, ha nem Swagger felületről tesztelünk.

Alapértelmezett URL útvonal

Az alapértelmezésben megnyitandó URL útvonalat a projekt tulajdonságok között adhatjuk meg: menu:zöld nyíl melletti legördülő menü[\<Projektnév> Debug Properties]. Ide egy a gyökércímhez képesti relatív útvonalrészt kell beírni. (pl. *api/efproduct*)

Figyeljük meg, hogy a controller a konstruktorban igényli meg a DI-tól az EF kontextet, amit a szokásos módon osztályváltozóban tárol el.

2.7.4 Köztes réteg alkalmazása

A rétegezett architektúra elveit követve gyakori eljárás, hogy a kontroller nem éri el közvetlenül az EF kontextet, hanem csak egy extra rétegen keresztül. A kontroller projekt így függetleníthető az EF modelltől.

Ehhez a megoldáshoz készítsünk külön kontroller változatot. A *Controllers* mappába hozzunk létre egy kontrollert (menu:Add[Controller > bal fában Common > API > jobb oldalon API Controller with read/write actions]) `ProductsController` néven.

A BLL projekt Services mappájába hozzunk létre egy új osztályt `ProductService` néven. Az új osztály kontroller számára nyújtandó funkcióit az `IProductService` adja meg.

Implementáljuk ezt az interfészt, a kiinduló implementációt generáltassuk a Visual Studio-val. Konstruktorban várja a függőségként a kontextet. A kontext segítségével implementáljuk normálisan a `GetProducts` függvényt. Eager Loading* használatával az egyes termékekhez a kapcsolódó kategóriát és megrendeléseket is adjuk vissza.

```
public class ProductService : IProductService
{
   private readonly AppDbContext _context;

   public ProductService(AppDbContext context)
   {
      _context = context;
   }

   public List<Product> GetProducts()
   {
      var products = _context.Products
         .Include(p => p.Category)
         .Include(p => p.ProductOrders)
            .ThenInclude(po => po.Order)
         .ToList();

      return products;
   }
   /*Többi függvény generált implementációja*/
}
```

Injektáljunk `IProductService`-t a `ProductsController`-be.

```
private readonly IProductService _productService;

public ProductsController(IProductService productService)
{
   _productService = productService;
}
```

Adjuk meg a DI alrendszernek, hogy hogyan kell egy `IProductService` típusú függőséget létrehozni. A legfelső szintű kódba:

```
builder.Services.AddTransient<IProductService, ProductService>();
```

A függőséginjektálás úgy működik, hogy a kontrollereket is a központi DI komponens példányosítja, és ilyenkor megvizsgálja a konstruktor paramétereket. Ha a konténerben talál alkalmas beregisztrált osztályt, akkor azt létrehozza és átadja a konstruktornak. Ezt hívjuk konstruktor injektálásnak. Ha a létrehozandó függőségnek is vannak konstruktor paraméterei, akkor azokat is megpróbálja feloldani, így rekurzívan a teljes függőségi objektum hierarchiát le tudja kezelni (ha abban nincs irányított kör). Ezt hívjuk autowiring-nek.

A regisztráció során több lehetőségünk is van. Egyrészt nem kötelező interfészt megadni egy osztály beregisztrálásához, az osztályt önmagában is be lehet regisztrálni, ilyenkor a konstruktorban is osztályként kell elkérni a függőségeket.

Háromféle példányosítási stratégiával regisztrálhatjuk be az osztályainkat:

- **Transient**: minden egyes injektálás során új példány jön létre
- **Scoped**: HTTP kérésenként egy példány kerül létrehozásra és a kérésen belül mindenkinek ez lesz injektálva
- **Singleton**: mindenkinek ugyanaz az egy példány kerül átadásra kéréstől függetlenül

Írjunk új `Get()` változatot az eredeti helyett a `ProductsController`-be az `IProductService` függőséget felhasználva:

```
[HttpGet]
public IEnumerable<Product> Get()
{
   return _productService.GetProducts();
}
```

Próbáljuk ki (`api/products`).

Hibát kapunk, mert a `ProductService` lekérdező függvénye eager loading-gal (`Include`) navigációs property-ket is kitölt, így könnyen hivatkozási kör jön létre, amit a JSON sorosító alapértelmezésben kivétellel jutalmaz. A sorosítást a keretrendszer végzi, a kontrollerfüggvény visszatérési értékét sorosítja a HTTP tartalomegyeztetési szabályok szerint. Böngésző kliens esetén alapesetben a JSON formátum lesz a befutó. Persze a sorosítás ennél közvetlenebbül is konfigurálható, ha szükséges.

A kontrollerek által használt JSON sorosítót konfigurálhatjuk a legfelső szintű kódban, például beállíthatjuk, hogy ha egy objektumot már korábban sorosított, akkor csak hivatkozzon rá és ne sorosítsa újra.

```
builder.Services.AddControllers() //; törölve
   .AddJsonOptions(o => o.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.Preserve);
```

Így már sikerülni fog a sorosítás, egy elég furcsa JSON-t láthatunk, ahol az első elem egy nagyobb objektumgráfot leíró rész, a többi elem pedig csak hivatkozás. Ennek a megoldásnak a hátránya, hogy a kliensoldali sorosítónak is támogatnia kell ezt a sorosítási logikát, a JSON-on belüli kereszthivatkozások kezelését. Emiatt kommentezzük is ki ezt a beállítást, keressünk más megoldást.

2.7.5 DTO osztályok

Láthattuk, hogy az entitástípusok közvetlen sorosítása gyakran nehézségekbe ütközik. A modell kifejezetten az EF számára lett megalkotva, illetve hogy a lekérdező műveleteket minél kényelmesebben végezhessük. A kliensoldal számára érdemes külön modellt megalkotni, egy ún. DTO (*Data Transfer Object*) modellt, ami a kliensoldal igényeit veszi figyelembe: **pontosan** annyi adatot és **olyan szerkezetben** tartalmaz, amire a kliensnek szüksége van.

A *BLL* projektben jelenleg egy nagyon egyszerű DTO modell található a *Dtos* mappában:

- rekord típusok alkotják a modellt
- nincs benne minden navigációs property, pl. `Category.Products`
- nincs benne a kapcsolótáblát reprezentáló entitás
- a termékből közvetlenül elérhetők a megrendelések

A különféle modellek közötti leképezésnél jól jönnek az ún. *object mapper*-ek, melyek segítenek elkerülni a leképezésnél nagyon gyakori repetitív kódokat, mint amilyen az `x.Prop = y.Prop` jellegű propertyérték-másolgatás.

Adjuk hozzá az API és a BLL projekthez az *AutoMapper* csomagot.

> **Tranzitív nuget referenciák**
>
> Alap esetben elég lenne csak a BLL projekthez felvenni a nuget csomagot, mivel az API projekt hivatkozik a BLL-re, az az ott behivatkozott csomagokat is láthatja (mint ahogy az EF-et is hivatkoztuk fentebbi rétegekből). Az AutoMapper-t viszont explicit be kell hivatkoznunk az API projektbe is, mert a BLL-ben behivatkozott, class library-val kompatibilis csomag nem tartalmazza az ASP.NET Core-hez szükséges konfigurációs függvényeket.

A leképezési konfigurációkat profilokba szervezve adhatjuk meg. Adjunk hozzá a BLL projekthez egy új osztályt `WebApiProfile` néven a *Dtos* mappába. Az AutoMapper konvenció alapon működik, tehát a DTO-entitás párokon kívül nem kell megadni például egyesével a property- vagy konstruktorparaméter-leképezéseket, ha a nevek alapján a leképezés kikövetkeztethető. Külön konfigurálásra csak a nem-triviális esetekben van szükség.

```
using AutoMapper;

namespace WebApiLab.Bll.Dtos;

public class WebApiProfile : Profile
{
    public WebApiProfile()
    {
        CreateMap<Dal.Entities.Product, Product>().ReverseMap();
        CreateMap<Dal.Entities.Order, Order>().ReverseMap();
        CreateMap<Dal.Entities.Category, Category>().ReverseMap();
    }
}
```

A DI konténerhez adjuk hozzá és konfiguráljuk a leképezési szolgáltatást.

```
builder.Services.AddAutoMapper(typeof(WebApiProfile));
```

Típusparaméter

Az AutoMapper az `AddAutoMapper` paramétereként megadott típust definiáló szerelvényben fogja a profilt keresni. A konkrét típusnak nincs más jelentősége, nem kell feltétlenül profilnak lenni.

Injektáéjuk be a leképzőt reprezentáló `IMapper` típusú objektumot a `ProductService`-be.

```
private readonly AppDbContext _context;
private readonly IMapper _mapper;

public ProductService(AppDbContext context, IMapper mapper)
{
    _context = context;
    _mapper = mapper;
}
```

A `ProductsController`-ben, az `IProductService`-ben és a `ProductService`-ben az entitásokra mutató névteret cseréljük ki a DTO-kra mutatóra:

```
//using WebApiLab.Dal.Entities;
using WebApiLab.Bll.Dtos;
```

Írjuk át a lekérdezést a `ProductService`-ben a leképzőt alkalmazva:

```
public List<Product> GetProducts()
{
    var products = _context.Products
        .ProjectTo<Product>(_mapper.ConfigurationProvider)
        .ToList();
    return products;
}
```

Hogy ne zavarjanak be a Swaggernek az `EFProductController`-ben használt entitás osztályok, töröljük ki a Controllers mappából az `EFProductController`-t!

Próbáljuk ismét meghívni böngészőből, figyeljük meg a naplóban, hogy milyen SQL lekérdezés fut le.

Modell rétegek

A többrétegű architektúránál elméletben minden rétegnek külön objektummodellje kellene, hogy legyen DAL: EF entitások, BLL: domain objektumok, Kontroller: DTO-k, viszont ha a domain objektumok nem visznek plusz funkciót a rendszerbe, akkor el szoktuk hagyni.

A DTO leképezést más rétegben is végezhetnénk. Egyes megközelítések szerint a kontroller réteg feladata lenne, azonban, ha az EF lekérdezésekkel összevonva végezzük a leképezést, akkor kiaknázhatjuk a *query result shaping* előnyeit, azaz csak azt kérdezzük le az adatbázisból, amire a leképezésnek szüksége van. Az AutoMapper `ProjectTo` függvénye ráadásul mindezt el is intézi helyettünk a leképezési konfiguráció alapján.

A `ProjectTo` metódust felfoghatjuk a továbbiakban egy LINQ-s `Select()` operátornak, annyi különbséggel, hogy az AutoMapper generálja azt az `Expression` -t, ami alapján előáll majd az eredmény.

ProjectTo

A `ProjectTo` speciálisan `IQueryable` -en működik. Ha csak simán memóriabeli objektumok között szeretnénk leképezni, akkor az `IMapper` `Map<>` függvényét hívjuk. A memóriabeli leképezésnek hátránya, hogy EF szinten gondoskodnunk kell róla, hogy `Include` hívásokkal a leképezéshez szükséges kapcsolódó entitásokat is lekérdezzük. A `ProjectTo` ezt is elintézi helyettünk.

2.7.6 BLL funkciók implementációja

Egy elem lekérdezése

Valósítsunk meg további interfész által előírt funkciókat a `ProductService` osztályban:

```
public Product GetProduct(int productId)
{
    return _context.Products
        .ProjectTo<Product>(_mapper.ConfigurationProvider)
        .SingleOrDefault(p => p.Id == productId)
        ?? throw new EntityNotFoundException("Nem található a termék", productId);
}
```

Szintén a `ProjectTo` -t használva, de most a `SingleOrDefault` LINQ operátorral kérdezzük le az egyetlen elemet, aminek az `Id` mezője egyezik a paraméterben kapott `productId` -val. Ha nem találjuk meg az elemet, akkor egy saját kivételt dobunk, ami majd a későbbiekben lekezelünk.

SingleOrDefault vagy FirstOrDefault

Ha biztosak vagyunk benne, hogy csak egy elemet találhatunk, akkor a `SingleOrDefault` használata javasolt, mert ha több elemet talál, akkor kivételt dob. Ha több elemet is várható egy lekérdezés eredményeként, de biztosak vagyunk benne a követelményeink alapján, hogy az első elem az, amit keresünk, akkor a `FirstOrDefault` használata javasolt.

Beszúrás

Ez hasonló az EF gyakorlaton látottakhoz, csak itt nem kell legyártanunk az új `Product` példányt, paraméterként kapjuk és memóriában leképezzük az enititásra. A `SaveChanges` hívás után a kulcs értéke már ki lesz töltve (adatbázis osztja ki a kulcsot).

```
public Product InsertProduct(Product newProduct)
{
    var efProduct = _mapper.Map<Dal.Entities.Product>(newProduct);
    _context.Products.Add(efProduct);
    _context.SaveChanges();
    return GetProduct(efProduct.Id);
}
```

Módosítás

Módosításhoz lekérdezzük az adott elemet, majd a `Map` függvénnyel a DTO-ból az entitásba mappeljük az új adatokat. Mentés után pedig visszaadjuk a módosított elemet.

```
public Product UpdateProduct(int productId, Product updatedProduct)
{
    var efProduct = _context.Products.SingleOrDefault(p => p.Id == productId)
        ?? throw new EntityNotFoundException("Nem található a termék", productId);
    _mapper.Map(updatedProduct, efProduct);
    _context.SaveChanges();
    return GetProduct(efProduct.Id);
}
```

Alternatív módosítás

Alternatíva, hogy a `Map` helyett a `Attach` függvényt használjuk, amivel az EF kontextusba visszatöltjük az entitást, majd a `Entry` függvénnyel jelöljük módosítottként. Ilyenkor spórolunk egy lekérdezést, de a SaveChanges hibával térhet vissza ha nem létező elemet próbálunk módosítani.

Törlés

Hasonlóan az előzőekhez, csak itt a `Remove` függvényt hívjuk meg a kontextuson.

```
public void DeleteProduct(int productId)
{
    var efProduct = _context.Products.SingleOrDefault(p => p.Id == productId)
        ?? throw new EntityNotFoundException("Nem található a termék", productId);
    _context.Products.Remove(efProduct);
    _context.SaveChanges();
}
```

Törlés lekérdezés nélkül

Egy trükkel elkerülhetjük, hogy le kelljen kérdezni a törlendő terméket. Az azonosító alapján előállítunk memóriában egy példányt a megfelelő kulccsal, majd `Remove` függvénnyel hozzáadjuk a kontexthez. A `Remove` törlendőnek jelöli a példányt, de itt is hibaágakra kell készülni, ha nem létező elemet próbálunk törölni.

```
public void DeleteProduct(int productId)
{
   _context.Products.Remove(new Dal.Entities.Product(null!) { Id = productId });
   _context.SaveChanges();
}
```

2.7.7 REST konvenciók alkalmazása

A REST megközelítés nem csak átviteli közegnek tekinti a HTTP-t, hanem a protokoll részeit felhasználja, hogy kiegészítő információkat vigyen át. Emiatt előnyös lenne, ha nagyobb ellenőrzésünk lenne a HTTP válasz felett - szerencsére az ASP.NET Core biztosítja ehhez a megfelelő API-kat.

GET - 200 OK

Egyik legegyszerűbb ilyen irányelv, hogy a lekérdezések eredményeként, ha megtaláltuk és visszaadtuk a kért adatokat, akkor **200 (OK)** HTTP válaszkódot adjunk.

HTTP és REST irányelvek

A HTTP kérést érintő irányelvekről egy jó összefoglaló elérhető itt.

Az eddig megírt `Get()` függvényünk most is **200 (OK)**-ot ad, ezt le is ellenőrizhetjük a böngészőnk hálózati monitorozó eszközében.

HTTP monitorozás

A HTTP kommunikáció megfigyelésére használhatjuk a böngészők beépített eszközeit, mint amilyen a Firefox Developer Tools, illetve Chrome DevTools. Általában az F12 billentyűvel aktiválhatók. Emellett, ha egy teljesértékű HTTP kliensre van szükségünk, amivel például könnyen tudunk nem csak GET kéréseket küldeni, akkor a Postman és a Fiddler Classic külön telepítendő eszközök ajánlhatók. A Fiddler mint proxy megoldás egy Windows gépen folyó HTTP kommunikáció megfigyelésére is alkalmas.

Első körben a két lekérdező függvényt írjuk át úgy, hogy a HTTP válaszkódokat explicit megadjuk. A jelenlegi ajánlás ehhez az `ActionResult<>` használata. Elég `T`-t visszaadnunk a függvényben, automatikusan `ActionResult<T>` típussá konvertálódik.

```
[HttpGet]
public ActionResult<IEnumerable<Product>> Get()
{
   return _productService.GetProducts();
}
```

Írjuk meg ugyanígy a másik `Get` függvényt is:

```
[HttpGet("{id}")]
public ActionResult<Product> Get(int id)
{
    return _productService.GetProduct(id);
}
```

Próbáljuk ki mindkét kontroller függvényt (*api/products*, *api/products/1*), ellenőrizzük a státuszkódokat is.

Ami fura, hogy még mindig nem állítottunk explicit státuszkódokat. A logikánk most még nagyon egyszerű, csak a hibamentes ágat kezeltük, így eddig az `ActionResult` alapértelmezései megoldották, hogy **200 (OK)**-ot kapjunk.

POST - 201 Created

Most viszont következzen egy létrehozó művelet:

```
[HttpPost]
public ActionResult<Product> Post([FromBody] Product product)
{
    var created = _productService.InsertProduct(product);
    return CreatedAtAction(nameof(Get), new { id = created.Id }, created);
}
```

Itt már látszik az `ActionResult` haszna. A konvenciónak megfelelően 201-es kódot akarunk visszaadni. Ehhez a `ControllerBase` ősosztály biztosít segédfüggvényt. A segédfüggvény olyan `ActionResult` leszármazottat ad vissza, ami 201-es kódot szolgáltat a kliensnek. Másik konvenció, hogy a *Location* HTTP fejlécben legyen egy URL az új termék lekérdező műveletének meghívásához. Ezt az URL-t rakjuk össze a `CreatedAtAction` paraméterei révén.

Gyakori, hogy a lefele irányú kommunikáció során (kliens felé) bővebb adathalmaz kerül leküldésre, mint amit egy létrehozáskor vagy módosításkor várunk. Esetünkben is az `Orders` és a `Category` propertyk létrehozáskor feleslegesek. Erre a célra jobb egy külön DTO-t létrehozni, ami csak a megfelelő adatokat tartalmazza. Most ideiglenesen tegyük nullozhatóvá ezt a két propertyt.

Product.cs

```
public Category? Category { get; init; } //? módosító bekerült
public List<Order>? Orders { get; init; } //? módosító bekerült
```

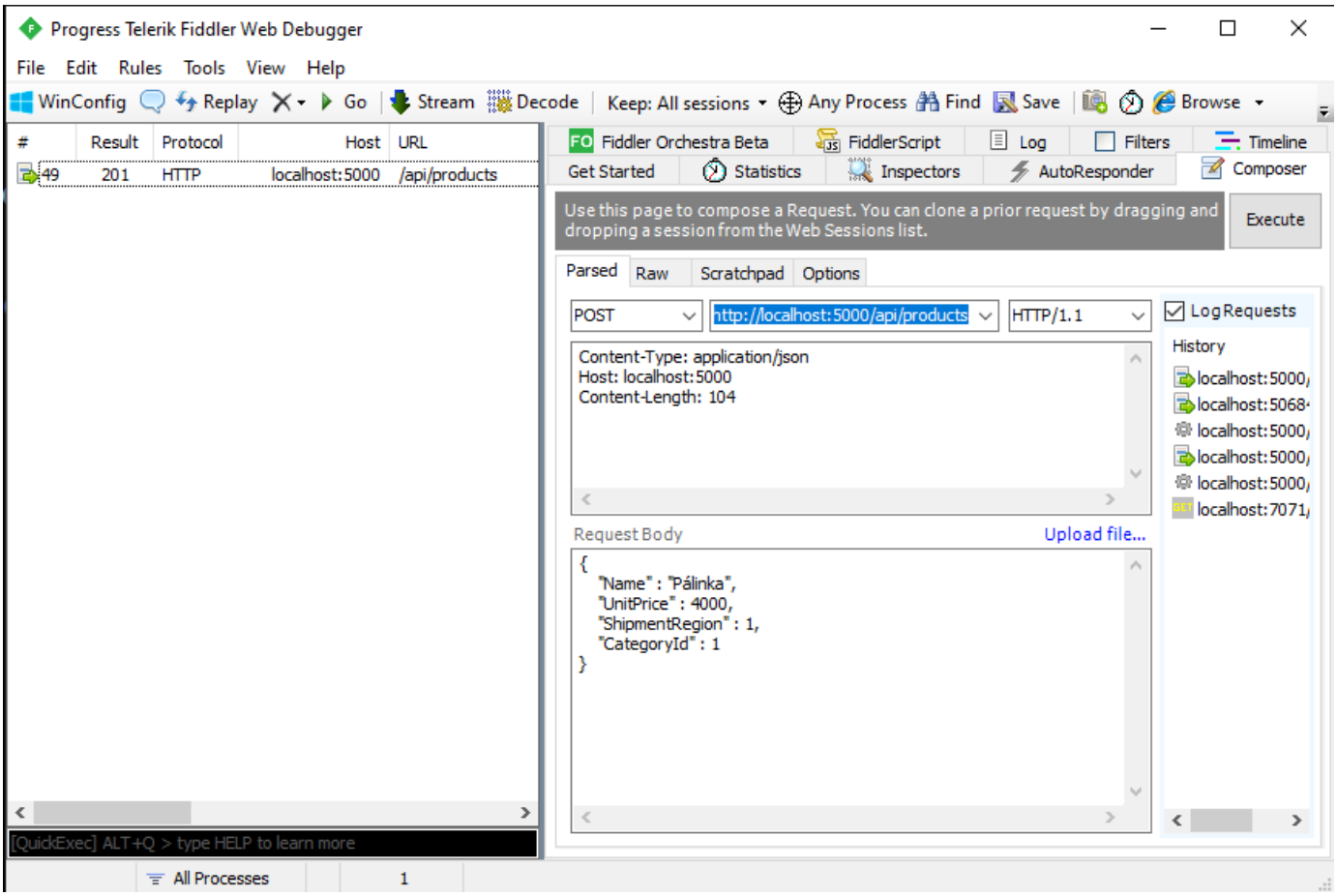
Próbáljuk ki a műveletet Swagger felületről. Egy `Product`-ot kell felküldenünk. Erre egy példa érték:

```
{
  "Name" : "Pálinka",
  "UnitPrice" : 4000,
  "ShipmentRegion" : 1,
  "CategoryId" : 1
}
```

> ⚠ **Content-Type**
>
> Ha Fiddlerből vagy Postmanből tesztelünk, ne felejtsük el a *Content-Type* fejlécet **application/json**-re állítani!

Figyeljük meg a kapott választ. A válaszból másoljuk ki a *Location* fejlécből az URL-t és hívjuk meg böngészőből.

Fiddler Classic példa POST hívásra:



PUT - 200 OK

A módosítás a konvenció szerint **200 (OK)** választ ad, mert a kliens a módosított erőforrást kapja vissza.

```
[HttpPut("{id}")]
public ActionResult<Product> Put(int id, [FromBody] Product product)
{
    return _productService.UpdateProduct(id, product);
}
```

PUT és PATCH

PUT mellett a módosításhoz használatos a PATCH is. A PUT konvenció szerint teljes, míg a PATCH részleges felülírásnál használatos. PATCH esetén általában valamilyen patch formátumú adatot küld a kliens, pl. RFC 6902 - JSON Patch. A JSON Patch formátumot jelenleg csak a JSON korábbi sorosító (*Newtonsoft.Json*) támogatja.

204 No Content

Módosítás esetében a konvenció megengedi, hogy üres törzsű (body) választ adjunk, ilyenkor a válaszkód **204 (No Content)**.

DELETE - 204 No Content

A törlő műveleteknél a konvenció megengedi, hogy üres törzsű (body) választ adjunk, ilyenkor a válaszkód **204 (No Content)**. Ilyesfajta válasz előállításához is van segédfüggvény, illetve elég csak az `ActionResult` típust megadni visszatérési típusnak:

```
[HttpDelete("{id}")]
public ActionResult Delete(int id)
{
    _productService.DeleteProduct(id);
    return NoContent();
}
```

Próbáljuk kitörölni az újonnan felvett terméket Swaggerből/Fiddler-ből/Postman-ből (*DELETE* igés kérés az `api/products/<új id>` címre, üres törzzsel). Sikerülnie kell, mert még nincs rá idegen kulcs hivatkozás.

2.7.8 Hibakezelés

Eddig főleg csak a hibamentes ágakat (happy path) néztük. A REST konvenciók rendelkeznek arról is, hogy bizonyos hibahelyezetekben milyen HTTP választ illik adni, például ha a kérésben hivatkozott azonosító nem létezik - 404-es hiba a bevett eljárás. Státuszkódok szempontjából a korábban idézett oldal ad segítséget, a válasz törzsében a hibaüzenet szerkezete tekintetében az RFC 7807 ad iránymutatást az ún. *Problem Details* típusú válaszok bevezetésével. Az ASP.NET Core támogatja a *Problem Details* válaszokat, és általában automatikusan ilyen válaszokat küld.

400 Bad Request

Kezdjük a kliens által küldött nem helyes adatokkal. Ez a hibakód nem összekeverendő a 415-tel, ahol az adat formátuma nem megfelelő (XML vagy JSON): ezt általában nem kell kézzel lekezeljük, mivel ezt az ASP.NET Core megteszi helyettünk. 400-zal olyan hibákat szoktunk lekezelni, ahol a küldött adat formátuma megfelelő, de valamilyen saját validációs logikának nem felel meg a kapott objektum, pl.: egységár nem lehet negatív stb.

Itt használjuk fel a .NET ún. *Data Annotation* attribútumait, amiket a DTO-kon érvényesíthetünk, és az ASP.NET Core figyelembe vesz a művelet végrehajtása során. Vegyünk fel a `Product` DTO osztályban néhány megkötést attribútumok formájában.

```
[Required(ErrorMessage = "Product name is required.", AllowEmptyStrings = false)]
public string Name { get; init; } = null!;

[Range(1, int.MaxValue, ErrorMessage = "Unit price must be higher than 0.")]
public int UnitPrice { get; init; }
```

Próbáljuk ki egy **POST /api/Products** művelet meghívásával. Paraméterként kiindulhatunk a felület által adott minta JSON-ból, csak töröljük ki a navigációs property-ket és sértsük meg valamelyik (vagy mindkét) fenti szabályt. Egy példa törzs:

```
{
  "Name" : "",
  "UnitPrice" : 0,
  "ShipmentRegion" : 1,
  "CategoryId" : 1
}
```

A válasz 400-as kód és valami hasonló, RFC 7807-nek megfelelő törzs lesz:

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "|2f35d378-4420cbafb80aec04.",
  "errors": {
    "Name": [
      "Product name is required."
    ],
    "UnitPrice": [
      "Unit price must be higher than 0."
    ]
  }
}
```

Összetettebb validáció

Az egyszerűbb eseteknél a *Data Annotation* attribútumok elegendőek, de ha összetettebb validációra van szükség, akkor érdemes a FluentValidation csomagot használni.

404 Not Found - kontroller szinten

Konvenció szerint 404-es hibát kellene adnunk, ha a keresett azonosítóval nem található erőforrás - esetünkben termék. Jelenleg a `ProductService` `EntityNotFoundException`-t dob, és amennyiben Development módban futtatjuk az alkalmazást, a cifra hibaoldal jelenik meg, amit a *DeveloperExceptionPage* middleware generál. Ha kivesszük a middleware-t (vagy nem Development módban indítjuk, de ekkor gondoskodnunk kell connection string-ről, ami eddig csak a Development konfigurációban volt beállítva), akkor 500-as hibát kapunk vissza.

Exception Shielding

A kezeletlen kivételek általában 500-as hibakód formájában kerülnek vissza a kliensre, mindenfajta egyéb információ nélkül (üres oldalként jelenik meg). Ez a jobbik eset, ahhoz képest, ha a teljes kivételszöveg és stack trace is visszakerülne. Az átlagos felhasználók nem tudják értelmezni, viszont a támadó szándékúaknak értékes információt jelenthet, így ajánlott elkerülni, hogy a kivétel ilyen módon kijusson. Ez az elkerülés az úgynevezett *exception shielding* technika, és az ASP.NET Core alapértelmezetten alkalmazza.

Legegyszerűbb módszer a kontroller műveletben érvényesíteni a konvenciót egy try-catch blokkal:

```
[HttpGet("{id}")]
public ActionResult<Product> Get(int id)
{
    try
    {
        return _productService.GetProduct(id);
    }
    catch (EntityNotFoundException)
    {
        return NotFound();
    }
}
```

null érték

Alternatív megoldás, hogy a `ProductService` egy `null` értékkel jelezné, hogy nincs találat. Ezesetben a fenti kódban a `null` értékre kellene vizsgálni, pl. `if` szerkezettel. A későbbiekben látjuk majd, hogy a kivételeket egyszerűbb központi helyen kezelni.

Próbáljuk ki, hogy 404-es státuszkódot és annak megfelelő *problem details*-t kapunk-e, ha egy nem létező termékazonosítóval hívjuk a fenti műveletet.

Ha saját *problem details*-t szeretnénk a 404-es kód mellé, akkor kézzel összerakhatjuk és visszaküldhetjük.

```
catch (EntityNotFoundException)
{
   ProblemDetails details= new ProblemDetails
   {
      Title = "Invalid ID",
      Status = StatusCodes.Status404NotFound,
      Detail = $"No product with ID {id}"
   };
   return NotFound(details);
}
```

Így is próbáljuk ki. Az általunk megadott üzenetet kell visszakapjuk.

404 Not Found - központi hibakezeléssel

A rendhagyó válaszok előállításánál előnyös lehet, ha az alacsonyabb rétegekből specifikus kivételeket dobunk, mert ezeket egy központi helyen szisztematikusan átalakíthatjuk konvenciónak megfelelő HTTP válaszokká. Ezt az ASP.NET Core 8 óta beépítetten meg tudjuk tenni. Erre a célra több kiterjesztési pontja is van a keretrendszernek 1 2, de nekünk most elég a legmagasabb szinten a `ProblemDetails` választ testreszabni.

Az `AddProblemDetails` konfigurációjában a saját kivételtípusunkat képezzük le 404-es hibakódra és a válasz tartalmát módosítsuk.

```
builder.Services.AddProblemDetails(options =>
      options.CustomizeProblemDetails = context =>
      {
         if (context.HttpContext.Features.Get<IExceptionHandlerFeature>()?.Error is EntityNotFoundException ex)
         {
            context.HttpContext.Response.StatusCode = StatusCodes.Status404NotFound;
            context.ProblemDetails.Title = "Invalid ID";
            context.ProblemDetails.Status = StatusCodes.Status404NotFound;
            context.ProblemDetails.Detail = $"No product with ID {ex.Id}";
         }
      }
   );
```

A middleware pipeline-ba az alábbi beépített middleware-eket kell felvenni a csővezeték elejére:

```
var app = builder.Build();

app.UseExceptionHandler();
app.UseStatusCodePages();
```

Térjünk vissza a korábbi, nem kivétel-elkapós változatra:

```
[HttpGet("{id}")]
public ActionResult<Product> Get(int id)
```

```
{
    return _productService.GetProduct(id);
}
```

Próbáljuk ki: hasonlóan kell működjön, mint a kontroller szintű változat, de ez általánosabb, bármely műveletből `EntityNotFoundException` érkezik, azt kezeli, nem kell minden műveletben megírni a kezelő logikát.

500 Internal Server Error

Beépítetten a fenti megoldás minden egyéb kezeletlen hibára 500-as hibakódot ad vissza, és egy általános ProblemDetails tartalommal tér vissza, ami nem tartalmazza a kivétel szövegét és stack trace-jét.

Az *exception shielding* elv miatt csak olyan kivételeknél alkalmazzuk, ahol a felhasználók számára hasznos, de nem technikai jellegű információt tartalmaz a kivétel szövege.

Delete idempotens működése

Jelenleg a delete műveletünk hibával tér vissza másodjára, ha 2x egymás után meghívnánk azonos azonosítóval.

Egy másik megközelítés szerint a DELETE műveletnek idempotensnek kellene lennie, tehát egymás után többször végrehajtva is sikeres eredményt kell kapjunk. Ez azt is jelenti, hogy 404-es hiba helyet 204 No Content státuszkódot kell küldenünk akkor is, ha nem található adott ID-val entitás. Ezt a jelenlegi kódban egyszerűen implementálhatjuk, hogy nem dobunk kivételt a megfelelő ágban.

Próbáljuk ki, hogy az egy termék lekérdezésénél, a módosításnál és a törlésnél is a rossz azonosító egységesen működik-e: 404-es hibát ad vissza, a Problem Details-ben a kivétel szövegével.

2.7.9 Aszinkron műveletek

Aszinkron műveletek alkalmazásával hatékonyságjavulást érhetünk el: nem feltétlenül az egyes műveleteink lesznek gyorsabbak, hanem időegység alatt több műveletet tudunk kiszolgálni. Ennek oka, hogy az `await`-nél (például egy adatbázis művelet elküldésekor) a várakozási idejére történő kiugrásnál, ha vissza tudunk ugrálni egészen az ASP.NET engine szintjéig, akkor a végrehajtó környezet a kiszolgáló szálat a várakozás idejére más kérés kiszolgálására felhasználhatja.

Aszinkronitás végigvezetése a kódban

Ökölszabály, hogy ha elköteleztük magunkat az aszinkronitás mellett, akkor ha megoldható, az aszinkronitást vezessük végig a kontrollertől az adatbázis művelet végrehajtásáig minden rétegben. Ha egy API-nak van *TAP* jellegű változata, akkor azt részesítsük előnyben (pl. `SaveChanges` helyett `SaveChangesAsync`). Ha aszinkronból szinkronba váltunk, csökkentjük a hatékonyságot, rosszabb esetben deadlock-ot is előidézhetünk.

Vezessük végig az aszinkronitást egy művelet teljes végrehajtásán:

IProductService.cs

```
public Task<Product> UpdateProductAsync(int productId, Product updatedProduct);
```

ProductService.cs

```
public async Task<Product> UpdateProductAsync(int productId, Product updatedProduct)
{
    var efProduct = await _context.Products.SingleOrDefaultAsync(p => p.Id == productId)
        ?? throw new EntityNotFoundException("Nem található a termék", productId);
    _mapper.Map(updatedProduct, efProduct);
    await _context.SaveChangesAsync();
    return await GetProductAsync(efProduct.Id);
}
```

ProductController.cs

```
[HttpPut("{id}")]
public async Task<ActionResult<Product>> Put(int id, [FromBody] Product product)
{
    return await _productService.UpdateProductAsync(id, product);
}
```

> **Async végződés és kontroller műveletek**
>
> Az *Async* végződés alkalmazása kontroller műveletek nevében jelenleg nem ajánlott, mert könnyen hibákba futhatunk.

Próbáljuk ki, hogy továbbra is működik a módosított művelet.

2.7.10 Végállapot

A végállapot elérhető a kapcsolódó GitHub repo megoldas ágán is.

2.8 ASP.NET Core webszolgáltatások III.

2.9 Kiegészítő anyagok, segédeszközök

- kapcsolódó repo: https://github.com/bmeviauav23/WebApiLab-kiindulo
- NSwag Studio - itt is elég csak a legfrissebb zip verziót az **Assets** részről letölteni
- Postman HTTP kérések küldéséhez

2.9.1 Kiinduló projektek beüzemelése

Klónozzuk le a kiinduló projekt `lab-kiindulo-240502` ágát, ez az előző gyakorlat folytatása - a kódot ismerjük. Ha nincs már meg az adatbázisunk, akkor az előző gyakorlat alapján hozzuk létre az adatbázist Code-First migrációval (`Update-Database`).

```
git clone https://github.com/bmeviauav23/WebApiLab-kiindulo -b lab-kiindulo-240502
```

2.9.2 Egyszerű kliens

A tárgy tematikájának ugyan nem része a kliensoldal, de demonstrációs céllal egy egyszerű kliensoldalról indított hívást implementálunk. A webes API-khoz nagyon sokféle technikával írhatunk klienst, mivel gyakorlatilag csak két képességgel kell rendelkezni:

- HTTP alapú kommunikáció, HTTP kérések küldése, a válasz feldolgozása
- JSON sorosítás

A fentiekhez szinte minden manapság használt kliensoldali technológia ad támogatást. Mi most egy sima, .NET alapú konzol alkalmazást írunk kliens gyanánt.

A két képességet könnyen lefedhetjük a **System.Net.Http** (HTTP kommunikáció) és a **System.Text.Json** (JSON sorosítás) csomagokkal. Mindkettő a **Microsoft.NetCore.App** shared framework része, így általában nem kell külön beszereznünk őket.

Adjunk a solution-höz egy konzolos projektet (Console App (.NET 8), **nem .NET Framework!**) *WebApiLab.Client* néven. A *Program.cs*-ben írjuk meg az egy terméket lekérdező függvényt (`GetProductAsync`) és hívjuk meg.

```
Console.Write("ProductId: ");
var id = Console.ReadLine();
if(id != null)
{
   await GetProductAsync(int.Parse(id));
}

Console.ReadKey();

static async Task GetProductAsync(int id)
{
   using var client = new HttpClient();

   // Ha eltér, a portot írjuk át a szervernek megfelelően
   var response = await client.GetAsync(new Uri($"http://localhost:5184/api/Products/{id}"));
   response.EnsureSuccessStatusCode();
```

```
    var jsonStream = await response.Content.ReadAsStreamAsync();
    var json = await JsonDocument.ParseAsync(jsonStream);
    Console.WriteLine(
      $"{json.RootElement.GetProperty("name")}:" +
      $"{json.RootElement.GetProperty("unitPrice")}");
}
```

Ez most jelenleg csak egy egyszerű példa DTO osztályok nélkül, a későbbiekben egyszerűsíteni fogjuk a JSON feldolgozást.

!!! tip .NET 8 kliensen is Az elterjedtebb .NET alapú kliensek, a WinForms, WPF alkalmazások a legutóbbi időkig .NET Framework alapúak voltak, viszont már egy ideje a .NET 6-os verziótól felfelé is támogatja a WinForms, WPF, WinUI, MAUI (régi Xamarin) alkalmazásokat. Célszerű ezeket választani a régi .NET Framework alapú változatok helyett.

Állítsuk be, hogy a szerver és a kliensoldal is elinduljon (menu:solutionön jobbklikk[Set startup projects…]), majd próbáljuk ki, hogy a megadott azonosítójú termék neve és ára megjelenik-e a konzolon.

Jelenleg csak alapszintű (nem típusos) JSON sorosítást alkalmazunk. A következő lépés az lenne, hogy a JSON alapján visszasorosítanánk egy konkrétabb objektumba. Ehhez kliensoldalon is kellene lennie egy `Product` DTO-nak megfelelő osztálynak. Hogyan jöhetnek létre a kliensoldali modellosztályok?

- kézzel létrehozzuk őket a JSON alapján - macerás, bár vannak rá eszközök, amik segítenek
- a DTO-kat osztálykönyvtárba szervezzük és mindkét alkalmazás hivatkozza

  csak akkor működik, ha mindkét oldal .NET-es, ráadásul könnyen kaphat az osztálykönyvtár olyan függőséget, ami igazából az egyik oldalnak kell csak, így meg mindkét oldal meg fogja kapni
- generáltatjuk valamilyen eszközzel a szerveroldal alapján - ezt próbáljuk most ki

Állítsuk be, hogy csak a szerveroldal (*Api* projekt) induljon.

2.9.3 OpenAPI/Swagger Szerver oldal

Az *OpenAPI* (eredeti nevén: *Swagger*) eszközkészlet segítségével egy JSON alapú leírását tudjuk előállítani a szerveroldali API-nknak. A leírás alapján generálhatunk dokumentációt, sőt kliensoldali kódot is a kliensoldali fejlesztők számára. Jelenleg a legfrissebb specifikáció az OpenAPI v3-as (OAS v3). Az egyes verziók dokumentációja elérhető itt.

Az OpenAPI nem .NET specifikus, különféle nyelven írt szervert és klienst is támogat. Ugyanakkor készültek kifejezetten a .NET-hez is OpenAPI eszközök, ezek közül használunk párat most. .NET környezetben a legelterjedtebb eszközkészletek:

- NSwag - leíró-, szerver-, és kliensoldali generálás is. Részleges OAS v3 támogatás.
- Swashbuckle - csak leíró generálás. OAS v3 támogatott.
- AutoRest - *npm* csomag .NET függőséggel, csak kliensoldali kódgeneráláshoz. Részleges OAS v3 támogatás.
- Swagger codegen - java alapú kliensoldali generátor. C# támogatás csak OpenAPI v2-höz
- Kiota - új, Microsoft fejlesztésű C# alapú kliensoldali generátor. OAS v3 támogatott.

Leíró generálás

Első lépésként a szerveroldali kódunk alapján Swagger leírást generálunk *NSwag* segítségével.

Adjuk hozzá a projekthez az **NSwag.AspNetCore** csomagot a `Package Manager Console` -ból vagy az API projekt Manage NuGet packages UI-on, és töröljük ki a **Swashbuckle.AspNetCore** csomagot.

Konfiguráljuk a szükséges szolgáltatásokat a DI rendszerbe.

```
//builder.Services.AddEndpointsApiExplorer();
//builder.Services.AddSwaggerGen();
builder.Services.AddOpenApiDocument();
```

Az OpenAPI leíró, illetve a dokumentációs felület kiszolgálására regisztráljunk egy-egy NSwag middleware-t az **Endpoint MW elé**. Az eddigi Swagger támogatással kapcsolatos kódok törölhetők.

```
if (app.Environment.IsDevelopment())
{
    //app.UseSwagger();
    //app.UseSwaggerUI();
    app.UseOpenApi();
    app.UseSwaggerUi();
}
```

A Swagger UI a `/swagger` útvonalon lesz elérhető. Próbáljuk ki, hogy működik-e a dokumentációs felület a **/swagger** útvonalon, illetve a leíró elérhető-e a **/swagger/v1/swagger.json** útvonalon.

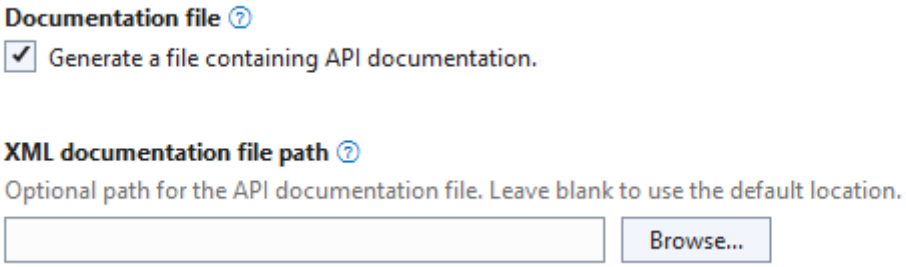
> **Tip**
>
> A Swagger leíró linkje megtalálható a dokumentációs felület címsora alatt.

Testreszabás - XML kommentek

Az NSwag képes a kódunk XML kommentjeit hasznosítani a dokumentációs felületen. Írjuk meg egy művelet XML kommentjét.

```
/// <summary>
/// Get a specific product with the given identifier
/// </summary>
/// <param name="id">Product's identifier</param>
/// <returns>Returns a specific product with the given identifier</returns>
/// <response code="200">Listing successful</response>
[HttpGet("{id}")]
public async Task<ActionResult<Product>> Get(int id){/*...*/}
```

A Swagger komponensünk az XML kommenteket nem a forráskódból, hanem egy generált állományból képes kiolvasni. Állítsuk be ennek a generálását a projekt build beállításai között (menu:Build[XML documentation file]). Az alatta lévő textbox-ot üresen hagyhatjuk.



Projektbeállítások (Build lap) - XML dokumentációs fájl generálása

Testreszabás - Felsorolt típusok sorosítása szövegként

Következő kis testreszabási lehetőség, amit kipróbálunk, a felsorolt típusok szövegként való generálása (az egész számos kódolás helyett). Ez általában a bevált módszer, mivel a kliensek számára kifejezőbb. A DI-ban a JSON sorosítást konfiguráljuk:

```
builder.Services.AddControllers() //; törölve
   .AddJsonOptions(o =>
   {
      o.JsonSerializerOptions.Converters.Add(new JsonStringEnumConverter());
   });
```

Próbáljuk ki, hogy az XML kommentünk megjelenik-e a megfelelő műveletnél, illetve a válaszban a `Product.ShipmentRegion` szöveges értékeket vesz-e fel.

Testreszabás - HTTP státuszkódok dokumentálása

Gyakori testreszabási feladat, hogy az egyes műveletek esetén a válasz pontos HTTP státuszkódját is dokumentálni szeretnénk, illetve ha több különböző kódú válasz is lehetséges, akkor mindegyiket.

Ehhez elég egy (vagy több) `ProducesResponseType` attribútumot felrakni a műveletre.

```
/// <summary>
/// Creates a new product
/// </summary>
/// <param name="product">The product to create</param>
/// <returns>Returns the product inserted</returns>
/// <response code="201">Insert successful</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
public async Task<ActionResult<Product>> Post([FromBody] Product product)
{/*...*/}

[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
public async Task<ActionResult> Delete(int id)
{/*...*/}
```

Ellenőrizzük, hogy a dokumentációs felületen a fentieknek megfelelő státuszkódok jelennek-e meg.

A hibaágakra is felvehetjük a megfelelő `ProducesResponseType` attribútumot, ahol annak generikus paramétere a hiba típusa.

```
[HttpGet]
[ProducesResponseType(StatusCodes.Status200OK)]
public async Task<ActionResult<IEnumerable<Product>>> Get()
{
   return await _productService.GetProductsAsync();
}

[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<ProblemDetails>(StatusCodes.Status404NotFound)]
public async Task<ActionResult<Product>> Get(int id)
{
   return await _productService.GetProductAsync(id);
}

[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
```

```
[ProducesResponseType<ProblemDetails>(StatusCodes.Status404NotFound)]
[ProducesResponseType<ValidationProblemDetails>(StatusCodes.Status400BadRequest)]
public async Task<ActionResult<Product>> Post([FromBody] Product product)
{
    var created = await _productService.InsertProductAsync(product);
    return CreatedAtAction(nameof(Get), new { id = created.Id }, created);
}

[HttpPut("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<ProblemDetails>(StatusCodes.Status404NotFound)]
[ProducesResponseType<ValidationProblemDetails>(StatusCodes.Status400BadRequest)]
public async Task<ActionResult<Product>> Put(int id, [FromBody] Product product)
{
    return await _productService.UpdateProductAsync(id, product);
}

[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType<ProblemDetails>(StatusCodes.Status404NotFound)]
public async Task<ActionResult> Delete(int id)
{
    await _productService.DeleteProductAsync(id);
    return NoContent();
}
```

Vegyük észre az alábbiakat:

- Ha nem adunk meg a `ProducesResponseType` attribútumnak típus paramétert, akkor a metódus visszatérési értékéből próbálja kitalálni a modellt.
- Amíg nem adtunk meg semmilyen `ProducesResponseType` attribútumot, addig a Swagger UI az egyenes ágra mindig 200-as státuszkódot fog feltételezni
- Ezt a feltételezést elveszítjük, ha bármilyen `ProducesResponseType` attribútumot felvesszük pl.: `ProducesResponseType(StatusCodes.Status404NotFound)` esetében szükséges már kiírni a 200-as státuszkódot is, ha az is lehetséges válasz.
- A hibaeseteket akár közössé is tehetnénk, ha a `ProducesResponseType` attribútumokat nem a metódusra, hanem a kontrollerre raknánk. Ilyenkor viszont hibásan a listás végpontra is azt nyilatkoznánk, hogy jöhet 404-es státuszkód, pedig ez nem igaz.

2.9.4 OpenAPI/Swagger kliensoldal

A kliensoldalt az *NSwag Studio* eszközzel generáltatjuk. Ez a generátor egy egyszerűen használható, de mégis sok beállítást támogató eszköz, azonban van pár hiányossága:

- egyetlen fájlt generál
- csak részlegesen támogatja az új JSON sorosítót, csak a régebbit

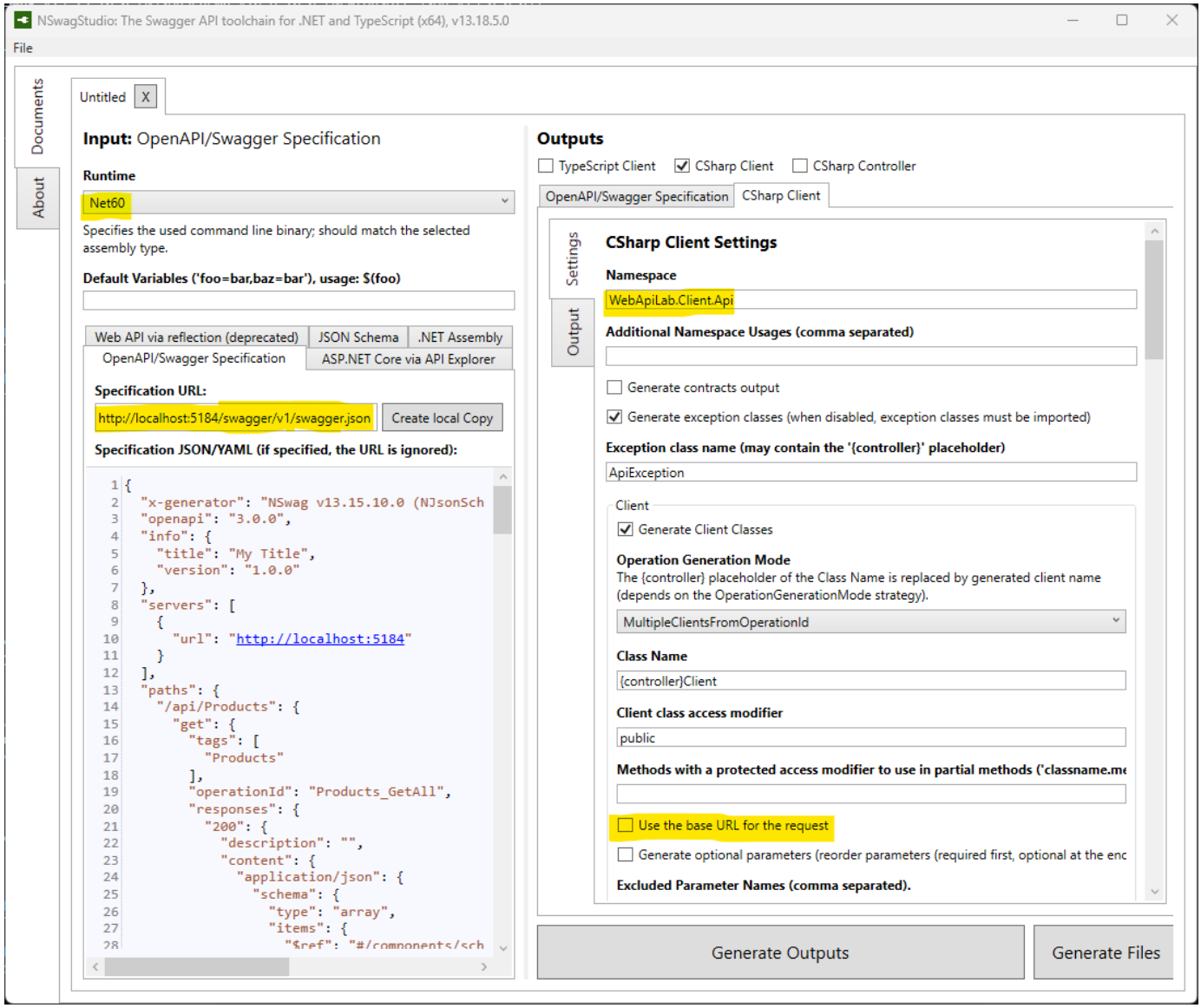
Előkészítésként adjuk a Client projekthez az alábbiakat:

- *Newtonsoft.Json* NuGet csomagot
- egy osztályt `ApiClients` néven

Indítsuk el a projektünket (a szerveroldalra lesz most szükség) és az NSwag Studio-t, és adjuk meg az alábbi beállításokat:

- Input rész (bal oldal): válasszuk az *OpenAPI/Swagger Specification* fület és adjuk meg a OpenAPI leírónk címét (pl.: http://localhost:5000/swagger/v1/swagger.json). Nyomjuk meg a **Create local Copy** gombot.
- Input rész (bal oldal) - Runtime: Net80

- Output rész (jobb oldal) - jelöljük be a *CSharp Client* jelölőt
- Output rész (jobb oldal) - *CSharp Client* fül - Settings alfül: fölül a *Namespace* mezőben adjunk meg egy névteret, pl. *WebApiLab.Client.Api*, lentebb a *Use the base URL for the request* ne legyen bepipálva



NSwag Studio beállítások

Jobb oldalt alul a *Generate Outputs* gombbal generáltathatjuk a kliensoldali kódot.

A generált kóddal írjuk felül az *ApiClients.cs* tartalmát (ehhez le kell állítani a futtatást). Ezután a projektnek fordulnia kell. Írjuk meg a *Program.cs*-ben a `GetProduct` új változatát:

```
static async Task<Product> GetProduct2Async(int id)
{
    using var httpClient = new HttpClient()
        { BaseAddress = new Uri("http://localhost:5184/") };
    var client = new ProductsClient(httpClient);
    return await client.GetAsync(id);
}
```

Használjuk az új változatot.

```
if (id != null)
{
    //await GetProductAsync(int.Parse(id));
    var p = await GetProduct2Async(int.Parse(id));
    Console.WriteLine($"{p.Name}: {p.UnitPrice}.-");
}
```

Állítsuk be, hogy a szerver és a kliensoldal is elinduljon, majd próbáljuk ki, hogy megjelenik-e a kért termék neve és ára.

> **NSwag beállításai**
>
> Ez csak egy minimálpélda volt, az NSwag nagyon sok beállítással rendelkezik.

A kliensre innentől nem lesz szükség, beállíthatjuk, hogy csak a szerver induljon.

> **Helyes státuszkódok**
>
> A generált kliens helyes működéséhez a műveletek minden nem hibát jelző státuszkódjait (2xx) dokumentálnunk kellene Swagger-ben a `ProducesResponseType` attribútummal, különben helyes szerver oldali lefutás után is kliensoldalon *nem várt státuszkód* hibát kaphatunk.

2.9.5 Hibakezelés II.

409 Conflict - konkurenciakezelés

Konfiguráljuk fel a `Product` **entitást** úgy, hogy az esetleges konkurenciahelyzeteket is felismerje a frissítés során. Jelöljünk ki egy kitüntetett mezőt (`RowVersion`), amit minden update műveletkor frissítünk, így ez az egész rekordra vonatkozó konkurenciatokenként is felfogható.

Ehhez vegyünk fel egy `byte[]` -t a `Product` entitás osztályba `RowVersion` néven.

```
public class Product
{
    //...
    public byte[] RowVersion { get; set; } = null!;
}
```

Állítsuk be az EF kontextben (`OnModelCreating`), hogy minden módosításnál frissítse ezt a mezőt és ez legyen a konkurenciatoken, az `IsRowVersion` függvény ezt egyben el is intézi:

```
modelBuilder.Entity<Product>()
    .Property(p => p.RowVersion)
    .IsRowVersion();
```

> **Mi történik a háttérben?**
>
> A háttérben az EF a módosítás során egy plusz feltételt csempész az UPDATE SQL utasításba, mégpedig, hogy az adatbázisban lévő `RowVersion` mező adatbázisbeli értéke az ugyanaz-e mint, amit ő ismert (a kliens által látott) értéke. Ha ez a feltétel sérül, akkor konkurenciahelyzet áll fent, mivel valaki már megváltoztatta az adatbázisban lévő értéket.

Migrálnunk kell, mert megjelent egy új mező a `Products` táblánkban. Ne felejtsük el a szokásos módon beállítani a Default Project-et a DAL-ra a Package Manager Console-ban!

```
Add-Migration ProductRowVersion
Update-Database
```

Még a `Product` DTO osztályba is fel kell vegyük a `RowVersion` tulajdonságot és legyen ez is kötelező.

```
public record Product
{
    //...
    [Required(ErrorMessage = "RowVersion is required")]
    public byte[] RowVersion { get; init; } = null!;
}
```

Konkurenciahelyzet esetén a 409-es hibakóddal szokás visszatérni, illetve **PUT** művelet során a válasz azt is tartalmazhatja, hogy melyek voltak az ütköző mezők. Az ütközés feloldása tipikusan nem feladatunk ilyenkor.

Készítsünk egy saját `ProblemDetails` leszármazottat. Hozzunk létre egy új mappát **ErrorHandling** néven az **Api** projektben és bele egy új osztályt `ConcurrencyProblemDetails` néven, az alábbi implementációval:

```
public record Conflict(object? CurrentValue, object? SentValue);

public class ConcurrencyProblemDetails : ProblemDetails
{
    public Dictionary<string, Conflict> Conflicts { get; }

    public ConcurrencyProblemDetails(DbUpdateConcurrencyException ex)
    {
        Status = 409;

        Conflicts = new Dictionary<string, Conflict>();
        var entry = ex.Entries[0];
        var props = entry.Properties
            .Where(p => !p.Metadata.IsConcurrencyToken).ToArray();
        var currentValues = props.ToDictionary(
            p => p.Metadata.Name, p => p.CurrentValue);

        entry.Reload();

        foreach (var property in props)
        {
            if (!Equals(currentValues[property.Metadata.Name], property.CurrentValue))
            {
                Conflicts[property.Metadata.Name] = new Conflict
                (
                    property.CurrentValue,
                    currentValues[property.Metadata.Name]
                );
            }
        }
    }
}
```

A fenti megvalósítás összeszedi az egyes property-khez (a `Dictionary` kulcsa) a jelenlegi (`CurrentValue`) és a kliens által küldött (`SentValue`) értéket. Adjunk egy újabb leképezést a hibakezelő MW-hez a legfelső szintű kódban:

```
builder.Services.AddProblemDetails(options =>
    options.CustomizeProblemDetails = context =>
    {
        if (context.HttpContext.Features.Get<IExceptionHandlerFeature>()?.Error is EntityNotFoundException ex)
        {
            // ...
        }
        else if (context.HttpContext.Features.Get<IExceptionHandlerFeature>()?.Error is DbUpdateConcurrencyException ex)
        {
```

```
            context.HttpContext.Response.StatusCode = StatusCodes.Status409Conflict;
            context.ProblemDetails = new ConcurrencyProblemDetails(ex);
        }
    }
);
```

Ezzel kész is az implementációnk, amit Postman-ből fogjuk kipróbálni. A kész kód elérhető a *net8-client-megoldas* ágon.

> **Konkurencia mező beszúrás esetében**
>
> A kötelezően kitöltendő konkurencia mező beszúrásnál kellemetlen, hiszen kliensoldalon még nem tudható a token kezdeti értéke. Ilyenkor használhatunk bármilyen értéket, az adatbázis fogja a kezdeti token értéket beállítani.
>
> Ezt feloldhatnánk úgy, hogy a különböző CRUD műveletekhez külön DTO-kat használunk, ahol a `RowVersion` mező csak a frissítésnél kötelező. Ezt a megoldást azonban nem fogjuk most bemutatni.

2.9.6 Postman használata

Postman segítségével összeállítunk egy olyan hívási sorozatot, ami két felhasználó átlapolódó módosító műveletét szimulálja. A két felhasználó ugyanazt a terméket (tej) fogja módosítani, ezzel konkurenciahelyzetet előidézve.

Kollekció generálás OpenAPI leíró alapján

A Postman képes az OpenAPI leíró alapján példahívásokat generálni. Ehhez indítsuk el a szerveralkalmazásunkat és a Postman-t is. A Postman-ben fölül az **Import** gombot választva adjuk meg az OpenAPI leíró swagger.json URL-jét (amit az elindított BE /swagger oldalán a címsor alatt találunk). A felugró ablakban csak a **Generate collection from imported APIs** opciót válasszuk. Ezután megjelenik egy új Postman API és egy új kollekció is **My Title** néven - ezeket nevezzük át **WebApiLab**-ra (menu:jobbklikk a néven[Rename]).

> **Importálás**
>
> További segítség a dokumentációban.

A kollekcióban mind az öt műveletre található példahívás.

Változók

A változókat a kéréseken belüli és a kérések közötti adatátadásra használhatjuk. Több hatókör (scope) közül választhatunk, amikor definiálunk egy változót: globális, kollekción belüli, környezeten belüli, kérésen belüli lokális. Sőt, egy adott nevű változót is definiálhatunk több szinten is - ilyenkor a specifikusabb felülírja az általánosabbat. Ebben a példában mi most csak a kollekció szintet fogjuk használni.

A kollekciót kiválasztva egy új fül jelenik meg, itt a **Variables** fülön állíthatjuk a változókat, illetve megnézhetjük az aktuális értéküket.

Változók

További segítség a kollekció változók felvételéhez a dokumentációban.

Vegyük fel az alábbi változókat:

- `u1_allprods` - az első felhasználó által lekérdezett összes termék adata
- `u1_tejid` - az előző listából az első felhasználó által kiválasztott termék (tej) azonosítója
- `u1_tej` - az előbbi azonosító alapján lekérdezett termék adata
- `u1_tej_deluxe` - az előbbi termék módosított termékadata, amit a felhasználó menteni kíván

Ne felejtsük el elmenteni a kollekció változtatásait a **Save** (Ctrl + S) gombbal.

Mentés

A Postman nem ment automatikusan, ezért lehetőleg mindig mentsünk (Ctrl + S), amikor egy másik hívás, kollekció szerkesztésére térünk át.

Mappák

A kéréseinket külön mappákba szervezve elkülöníthetjük a kollekción belül az egyes (rész)folyamatokat. Mappákat a kollekció extra menüjén (a kollekció neve mellett a **...** ikont megnyomva) belül az **Add Folder** menüpont segítségével vehetünk fel.

Vegyünk fel a kollekciónkba egy új mappát **Update Tej** néven.

Mappák

További segítség új mappa felvételéhez a dokumentációban.

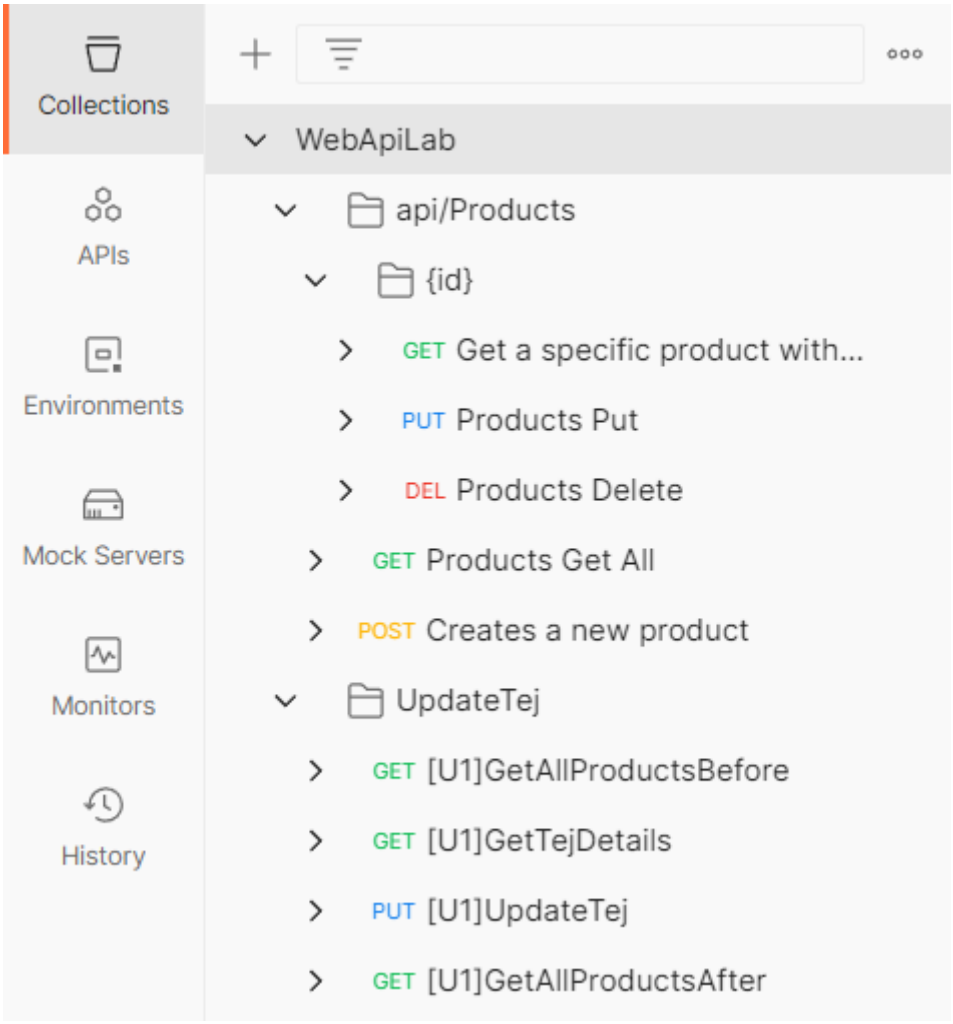
Egy felhasználó folyamata

Egy tipikus módosító folyamat felhasználói szempontból az alábbi lépésekből áll - az egyes lépésekhez szerveroldali API műveletek kapcsolódnak, ezeket a listaelemekhez hozzá is rendelhetjük:

- összes termék megjelenítése - API: összes termék lekérdezése
- módosítani kívánt termék kiválasztása - API: **nincs teendő, tisztán kliensoldali művelet**
- a módosítani kívánt termék részletes adatainak megjelenítése - API: egy termék adatainak lekérdezése
- a kívánt módosítás(ok) bevitele - API: **nincs, tisztán kliensoldali művelet**
- mentés - API: adott termék módosítása
- (vissza) navigáció + aktuális (frissített) állapot megjelenítése - API: összes termék lekérdezése

A négy API hívást klónozzuk (Ctrl + D) a generált példahívásokból. Egy adott hívásra csináljunk egy klónt (jobbklikk **Duplicate**), drag-and-drop-pal húzzuk rá az új mappánkra, végül nevezzük át (Ctrl + E). Ezekre a hívásokra csináljuk meg:

- összes termék lekérdezése (módosítás előtt), azaz **Products Get All** példahívás, nevezzük át erre: **[U1]GetAllProductsBefore**
- egy termék adatainak lekérdezése, azaz az `{id}` mappán belüli **Get a specific product with the given identifier** példahívás, nevezzük át erre **[U1]GetTejDetails**
- adott termék módosítása, azaz az `{id}` mappán belüli **Products Put** példahívás, nevezzük át erre **[U1]UpdateTej**
- összes termék lekérdezése (módosítás után), azaz **Products Get All** példahívás, nevezzük át erre: **[U1]GetAllProductsAfter**



Postman hívások - egy felhasználó folyamata

> **Warning**
>
> Vegyük észre, hogy az elnevezések az OpenAPI leíró alapján generálódnak, tehát ha máshogy dokumentáltuk az API-nkat, akkor más lesz a példahívások neve is.

Összes termék lekérdezése, saját vizualizáció és adattárolás változóba

Az **[U1]GetAllProductsBefore** hívás már most is kipróbálható külön a **Send** gombbal és az alsó **Body** részen látható az eredmény formázott (**Pretty**) és nyers (**Raw**) nézetben.

Saját vizualizációt is írhatunk, ehhez a kérés **Tests** fülét használhatjuk. Az ide írt *JavaScript* nyelvű kód a kérés után fog lefutni. Általában a válaszra vonatkozó teszteket szoktuk ide írni.

Írjuk be a kérés **Tests** fülén lévő szövegdobozba az alábbi kódot, ami egy táblázatos formába formázza a válasz JSON fontosabb adatait:

```
const template = `
  <table bgcolor="#FFFFFF">
    <tr>
      <th>Name</th>
      <th>Unit price</th>
      <th>[Hidden]Concurrency token</th>
    </tr>

    {{#each response}}
      <tr>
        <td>{{name}}</td>
        <td>{{unitPrice}}</td>
        <td>{{rowVersion}}</td>
      </tr>
    {{/each}}
  </table>
`;
const respJson = pm.response.json();
pm.visualizer.set(template, {
  response: respJson
});
```

Vizualizáció

További segítség a vizualizációkhoz a dokumentációban.

A visszakapott adatokra a későbbi lépéseknek is szükségük lesz, ezért mentsük el az `u1_allprods` változóba.

```
pm.visualizer.set(template, {
  response: respJson
});

pm.collectionVariables.set("u1_allprods", JSON.stringify(respJson));
```

Sorosítás változókba

Változóba mindig sorosított (pl. egyszerű szöveg típusú) adatot mentsünk, ne közvetlenül a JavaScript változókat. Ezzel elkerülhetjük a JavaScript típusok és a Postman változók közötti konverziós problémákat.

Próbáljuk ki így a kérést, alul a **Body** fül **Visualize** alfülén táblázatos megjelenítésnek kell megjelennie, illetve a kollekció változókezelő felületén az `u1_allprods` értékbe be kellett íródnia a teljes válasz törzsnek.

Változók

Nem kötelező előzetesen felvenni a változókat, a `set` hívás hatására létrejön, ha még nem létezik.

> **Scriptek**
>
> További segítség szkriptek írásához a dokumentációban.

Egy termék részletes adatainak lekérdezése, változók felhasználása

A forgatókönyvünk szerint a felhasználó a termékek listájából kiválaszt egy terméket (a *Tej* nevűt). Ezt a lépést szkriptből szimuláljuk, mint az **[U1]GetTejDetails** hívás előtt lefutó szkript. A hívás előtt futó szkripteket a hívás **Pre-request Script** fülén lévő szövegdobozba írhatjuk:

```
const allProds = JSON.parse(pm.collectionVariables.get("u1_allprods"));
const tejid = allProds.find(({ name }) => name.startsWith('Tej')).id;
pm.collectionVariables.set("u1_tejid", tejid);
```

Tehát kiolvassuk az elmentett terméklistát, kikeressük a *Tej* nevű elemet, vesszük annak azonosítóját, amit elmentünk az `u1_tejid` változóba. Ezt a változót már fel is használjuk a kérés paramétereként: a **Params** fülön az `id` nevű URL paraméter (**Path Variable**) értéke legyen `{{u1_tejid}}`

A kérés lefutása után mentsük el a válasz törzsét az `u1_tej` változóba. A **Tests** fülön lévő szövegdobozba:

```
pm.collectionVariables.set("u1_tej", pm.response.text());
```

> **Részletes adatok**
>
> Ezt a fázist ki is lehetne hagyni, mert a listában már minden szükséges adat benne volt a módosításhoz, de általánosságban gyakori, hogy egy részletes nézeten lehet a módosítást elvégezni, ami a részletes adatok lekérdezésével jár.

Módosított termék mentése

Mielőtt a módosított terméket elküldenénk a szervernek, szimuláljuk magát a felhasználói módosítást. Az **[U1]UpdateTej** hívás **Pre-request Script**-je legyen ez:

```
const tej = JSON.parse(pm.collectionVariables.get("u1_tej"));
tej.unitPrice++;
pm.collectionVariables.set("u1_tej_deluxe", JSON.stringify(tej));
```

Látható, hogy a módosított termékadatot egy új változóba (`u1_tej_deluxe`) mentjük. Ennél a hívásnál is a **Params** fülön az `id` nevű URL paraméter (**Path Variable**) értéke legyen `{{u1_tejid}}` . Viszont itt már a kérés törzsét is ki kell tölteni a módosított termékadattal. Mivel ez meg is van változóban, így elég a **Body** fül szövegdobozába (**Raw** nézetben) csak ennyit beírni: `{{u1_tej_deluxe}}` .

Frissített terméklista lekérdezése, folyamat futtatása

Az utolsó folyamatlépésnél már nincs sok teendő, ha akarunk vizualizációt, akkor a **Tests** fül szövegdobozába másoljuk át a fentebbi vizualizációs szkriptet.

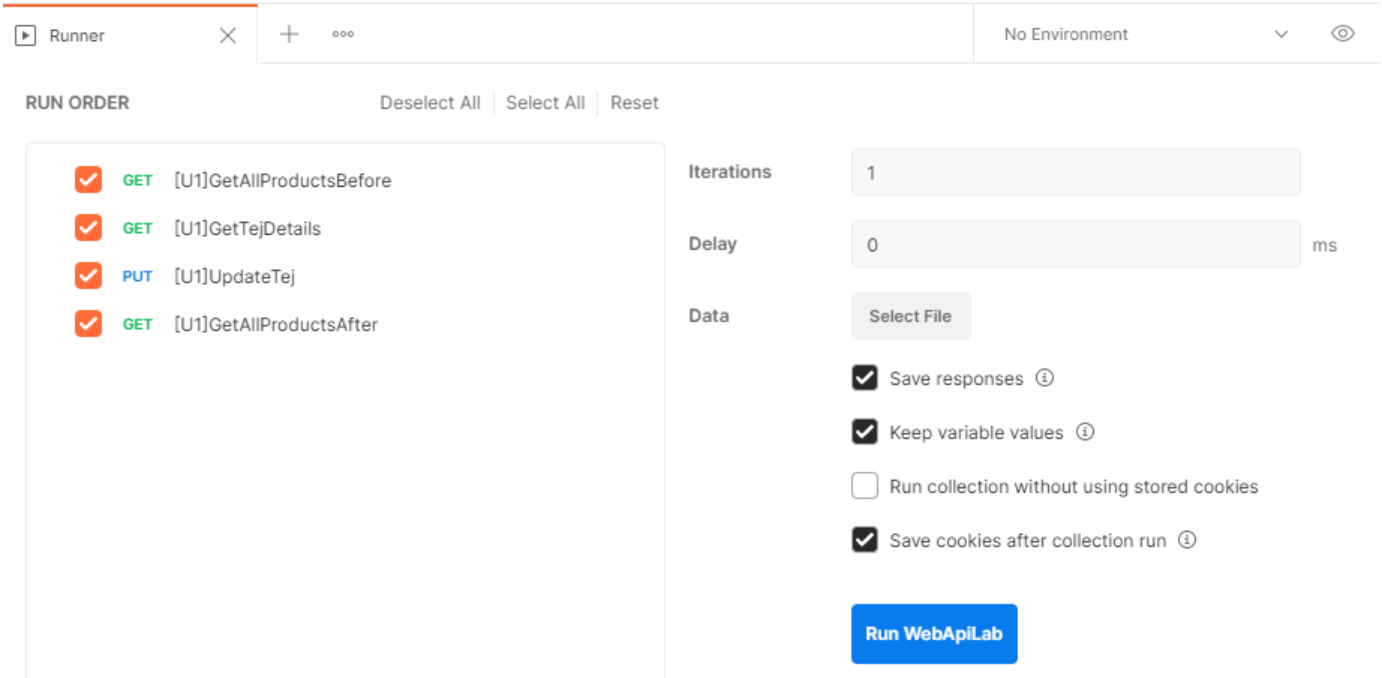
Egy kéréssorozat futtatásához használható a **Collection Runner** funkció, ami a kollekció vagy egy almappájának oldaláról (ami a kollekció/almappa kiválasztásakor jelenik meg) a jobb szélen a **Save** melletti **Run** gombra nyomva

hozható elő. A megjelenő ablak bal oldalán megjelennek a választott kollekció/mappa alatti hívások, amiket szűrhetünk (a hívások előtti jelölődobozzal), illetve sorrendezhetünk (a sor legelején lévő fogantyúval).

Futtatás

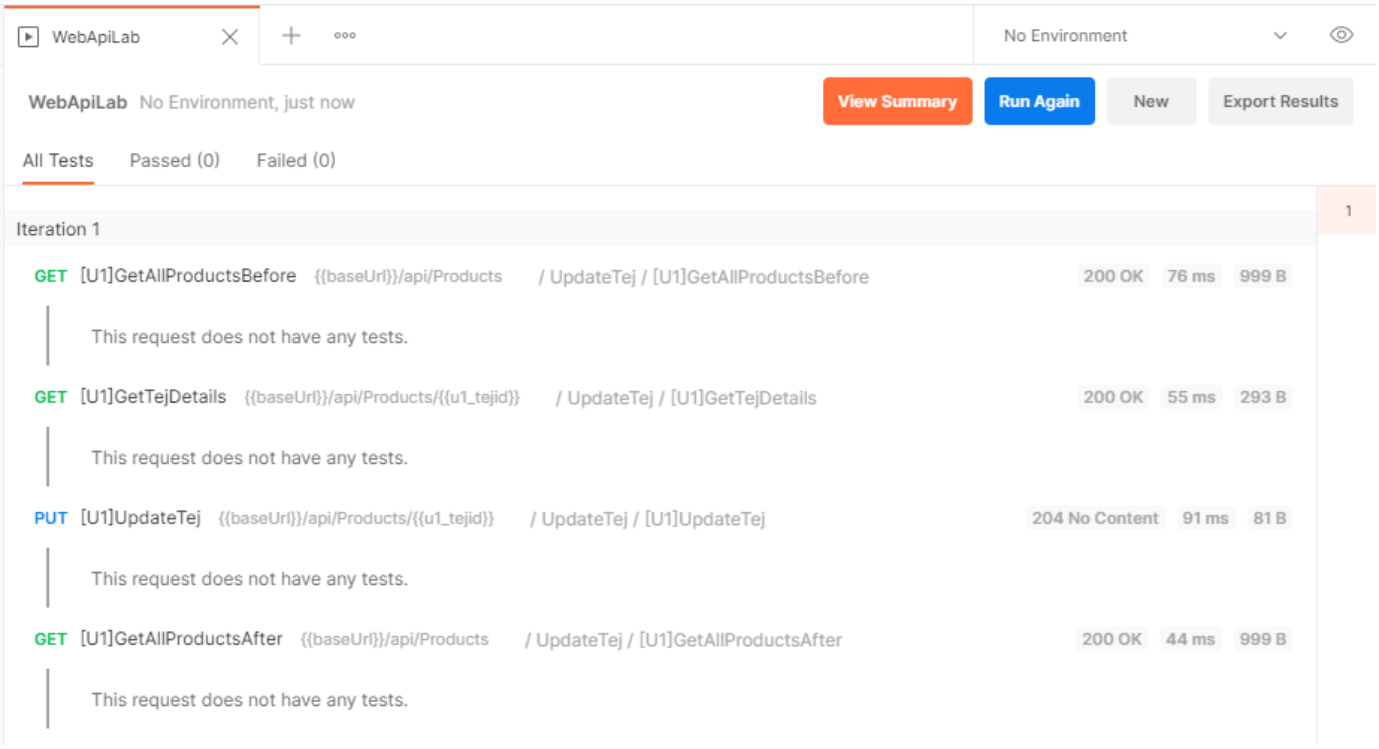
További segítség kollekciók futtatásához a dokumentációban.

Az eddig elkészült folyamatunk futtatásához válasszuk ki az **Update Tej** mappát. Érdemes beállítani a jobb részen a **Save responses** jelölőt, így a lefutás után megvizsgálhatjuk az egyes kérésekre jött válaszokat.



Postman Runner konfigurálása egy felhasználó folyamatának futtatásához

Próbáljuk lefuttatni a folyamatot, a lefutás után a válaszokban ellenőrizzük a termékadatokat (kattintsuk meg a hívást, majd a felugró ablakocskában válasszuk a **Response Body** részt), különösen az utolsó hívás utánit - a tej árának meg kellett változnia az első híváshoz képest.



Postman Runner - egy felhasználó folyamatának lefutása

A második felhasználó folyamata

Az alábbi lépésekkel állítsuk elő a második felhasználó folyamatát:

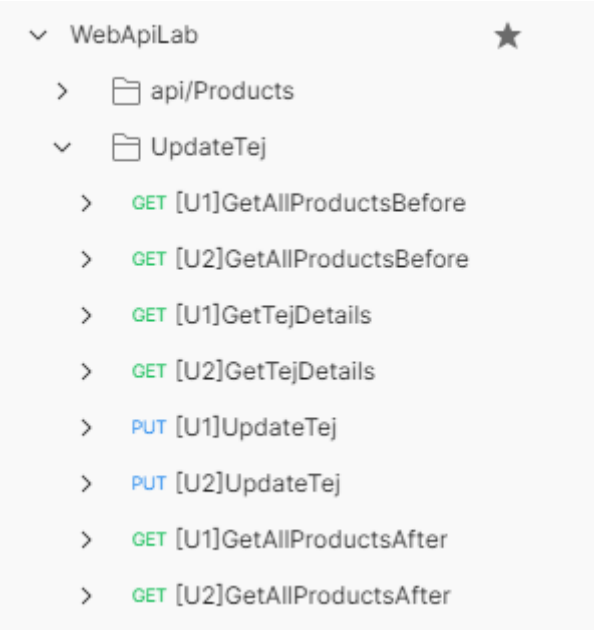
- vegyünk fel minden `u1` változó alapján új változót `u2` névkezdettel
- duplikáljunk minden **[U1]** hívást, a klónok neve legyen ugyanaz, mint az eredetié, de kezdődjön **[U2]**-vel
- a klónok minden szkriptjében, illetve paraméterében írjunk át **minden** `u1`-es változónevet `u2`-esre

  az **[U2]GetAllProductsBefore** hívásban a **Tests** fülön egy helyen

  az **[U2]GetTejDetails** hívásban a **Pre-request Script** fülön két helyen, a **Tests** fülön egy helyen, illetve a **Params** fülön egy helyen

  az **[U2]UpdateTej** hívásban a **Pre-request Script** fülön két helyen, a **Body** fülön egy helyen, illetve a **Params** fülön egy helyen
- az **[U2]UpdateTej** hívás **Pre-request Script** módosító utasítását írjuk át a lenti kódra. A termék nevét módosítjuk, nem az árát, a konkurenciahelyzetet ugyanis akkor is érzékelni kell, ha a két felhasználó nem ugyanazt az adatmezőt módosítja (ugyanazon terméken belül).

```
tej.name = "Tej " + new Date().getTime();
```



Postman hívások - mindkét felhasználó folyamata

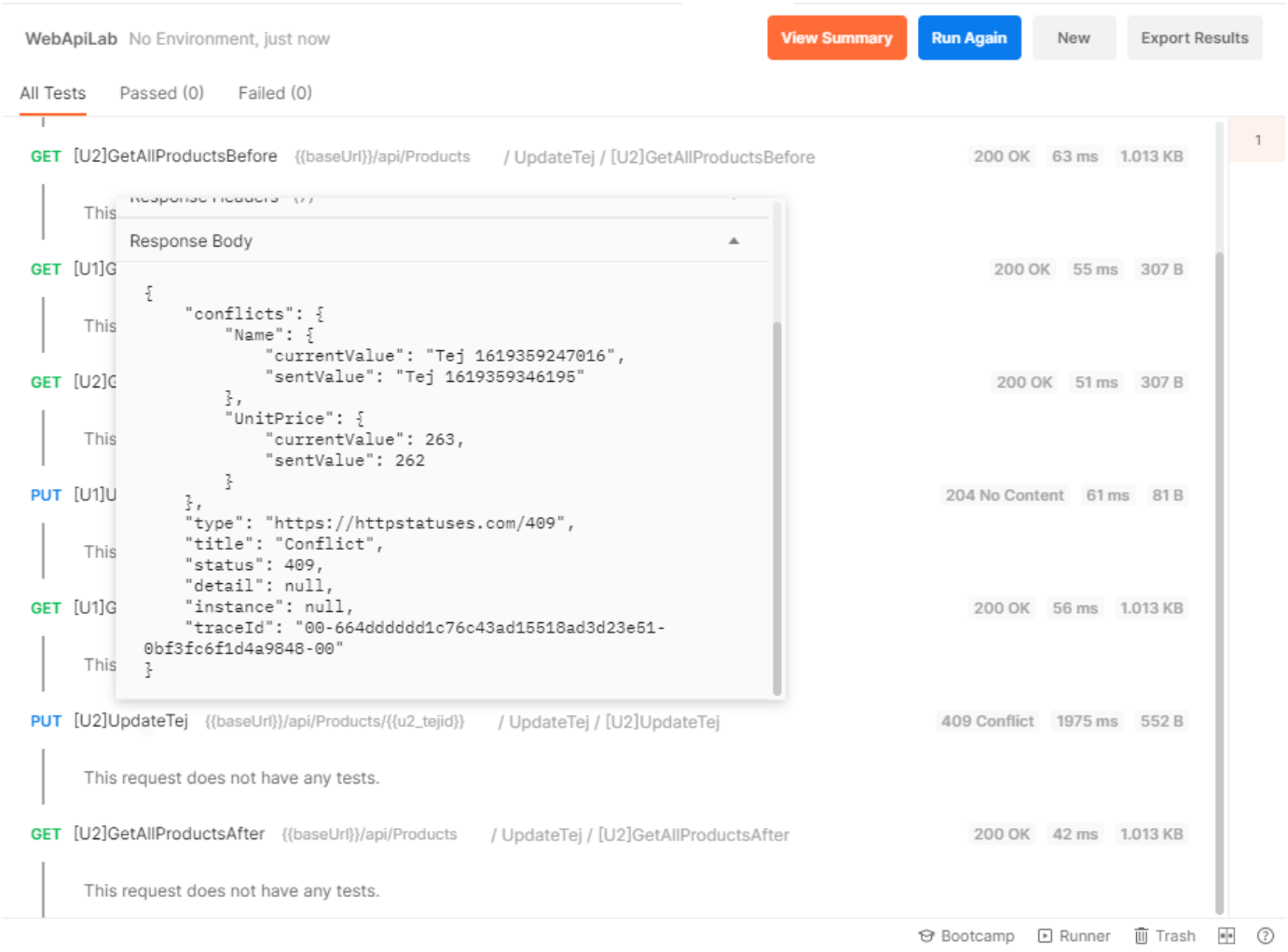
Ezzel elkészült a második felhasználó folyamata. Attól függően, hogy hogyan lapoltatjuk át a négy-négy hívást, kapunk vagy nem kapunk 409-es válaszkódot futtatáskor. Az alábbi sorrend nem ad hibát, hiszen a második felhasználó azután kéri le a terméket, hogy az első felhasználó már módosított:

1. **[U1]GetAllProductsBefore**
2. **[U2]GetAllProductsBefore**
3. **[U1]GetTejDetails**
4. **[U1]UpdateTej**
5. **[U1]GetAllProductsAfter**
6. **[U2]GetTejDetails**
7. **[U2]UpdateTej**
8. **[U2]GetAllProductsAfter**

Az utolsó hívás után a tej ára és neve is megváltozott.

Az alábbi sorrend viszont hibát ad, hiszen a második felhasználó már elavult `RowVersion`-t fog mentéskor elküldeni:

1. **[U1]GetAllProductsBefore**
2. **[U2]GetAllProductsBefore**
3. **[U1]GetTejDetails**
4. **[U2]GetTejDetails**
5. **[U1]UpdateTej**
6. **[U1]GetAllProductsAfter**
7. **[U2]UpdateTej**
8. **[U2]GetAllProductsAfter**



Postman Runner lefutás konkurenciahelyzettel

Konkurenciakezelés során felhasználható adatok

Érdemes megvizsgálni a 409-es hibakódú válasz törzsét és benne a változott mezők eredeti és megváltozott értékét.

Konkurenciakezelés szabályai

Ha igazi klienst írunk, figyeljünk arra, hogy a konkurenciatokent mindig küldjük le a kliensnek, a kliens változatlanul küldje vissza a szerverre, és a szerver pedig a módosítás során **a klienstől kapott** tokent szerepeltesse a módosítandó entitásban. A legtöbb hibás implementáció arra vezethető vissza, hogy nem követjük ezeket az elveket. Szerencsére az adatelérési kódunkban ezeknek a problémáknak a nagy részét megoldja az EF.

Postman Runner

Hívásokból álló folyamatokat nem csak **Runnerben** állíthatunk össze, hanem szkriptből is. Ha épp ellenkezőleg, kevesebb szkriptelést szeretnénk, akkor a Postman Flows ajánlott.

Az elkészült teljes Postman kollekció importálható erről a linkről az OpenAPI importáláshoz hasonló módon. A kollekció szinten ne felejtsük el beállítani a `baseUrl` változót a szerveralkalmazásunk alap URL-jére.

2.10 Automatizált tesztelés

2.10.1 Segédeszközök

- kapcsolódó GitHub repo: https://github.com/bmeviauav23/WebApiLab-kiindulo

2.10.2 Bevezetés

Az automatizált tesztelés az alkalmazásfejlesztés egyik fontos lépése, mivel ezzel tudunk meggyőződni arról, hogy egy-egy funkció akkor is helyesen működik, ha az alkalmazás egy másik részén valamit módosítunk. Hogy ezt az ellenőrzést ne kelljen minden egyes alkalommal manuálisan végrehajtani az alkalmazáson, programozott teszteket szoktunk írni, amelyek futtatását CI/CD folyamatokban automatizálhatjuk.

A tesztek több típusát ismerhetjük:

- **Unit test (egységteszt)** célja, hogy egy adott osztály egy metódusának a viselkedését önmagába vizsgáljuk úgy, hogy a függőségeit mock/fake objektumokkal helyettesítjük, hogy azok a tesztesetnek megfelelően viselkedjenek vagy megfigyelhetőek legyenek.
- **Integrációs teszt / End-2-end teszt / funkcionális teszt** esetében a célunk, hogy a teljes rendszert meghajtsuk úgy, hogy az integrációk (SQL kapcsolat, egyéb szolgáltatások) is tesztelésre kerülnek, illetve a BE szempontjából vizsgáljuk azt is, hogy a rendszer interfésze helyesen válaszol-e a különböző kérésekre.
- **UI teszt** esetében azt vizsgáljuk, hogy a felhasználói felület a különböző felhasználói interakciókra, eseményekre helyesen rajzolja-e ki az elvárt felületeket.

A fenti tesztelési módok mindegyike fontos, de érdemes egy olyan egészséges egyensúlyt megtalálni, ahol a lehető legjobban lefedhetőek a legfontosabb funkcionalitások különböző tesztesetekkel.

2.10.3 Automatizált tesztelés .NET környezetben

Automatizált tesztelésre több keretrendszer is használható .NET környezetben, de ASP.NET Core alkalmazások esetében a legelterjedtebb ilyen könyvtár az **xUnit**. Ebben a keretrendszerben lehetőségünk van tesztesetek definiálására, akár a bemenetek variálásával is, illetve kellően rugalmas, ahhoz, hogy a tesztek feldolgozási mechanizmusa kiterjeszthető legyen.

Unit tesztek esetében az osztályok függőségeit le kell cseréljük, amire több library is lehetőséget nyújt. A legelterjedtebbek a **Moq** és az **NSubstitute**.

Gyakran szükséges funkció, hogy a bemenő adatok előállítása során szeretnénk a valóságra hasonlító véletlenszerű/generált példaadatokat megadni. Ehhez egy bevált osztálykönyvtár a **Bogus**.

A tesztesetek elvárt eredményének a vizsgálatát asszertálásnak nevezzük (*assert*), aminek az írásához nagy segítséget tud nyújtani a **Fluent Assertions** könyvtár. Ez nem csak a szintaktikát teszi olvashatóbbá fluent szintakszissal, hanem több olyan beépített segédlogikát tartalmaz, amivel tömörebbé tehető az *assert* logika (pl.: objektumok mélységi összehasonlítása érték szerint).

2.10.4 Integrációs tesztelés

Ezen gyakorlat keretében csak integrációs teszteket fogunk készíteni.

Teszt projekt

Vegyünk fel a solutionbe egy új xUnit (.NET 8) típusú projektet `WebApiLab.Tests` néven. A létrejövő tesztosztályt és fájlját nevezzük át `ProductControllerTests` névre. Ide fogjuk a `ProductController` -hez kapcsolódó műveletekre vonatkozó integrációs teszteket készíteni.

Vegyük fel az alábbi NuGet csomagokat a teszt projektbe. A *Bogus*ról és a *Fluent Assertions*ről már volt szó. A *Microsoft.AspNetCore.Mvc.Testing* csomag olyan segédszolgáltatásokat nyújt, amivel integrációs tesztekhez egy in-process teszt szervert tudunk futtatni, és ennek a meghívásában is segítséget nyújt. A projektfájlban a többi `PackageReference` mellé (menu:a projekten jobbklikk[Edit Project File]):

```
<PackageReference Include="Bogus" Version="35.5.1" />
<PackageReference Include="FluentAssertions" Version="6.12.0" />
<PackageReference Include="Microsoft.AspNetCore.Mvc.Testing" Version="8.0.4" />
```

Vegyük fel az **Api** projektet projekt referenciaként a teszt projektbe. A projektfájlban egy másik `ItemGroup` mellé:

```
<ItemGroup>
  <ProjectReference Include="..\WebApiLab.Api\WebApiLab.Api.csproj" />
</ItemGroup>
```

Teszt szerver

A tesztszervernek meg kell tudnunk mondani, hogy melyik osztály adja az alkalmazásunk belépési pontját. Viszont mivel top level statement szintaktikájú a `Program` osztályunk, annak láthatósága internal, ami a tesztelés szempontjából nem szerencsés (a hasonló esetekben alkalmazott `InternalsVisibleTo` sem lenne ebben az esetben megoldás). Helyette tegyük a `Program` osztályt publikussá egy `partial` deklarációval. Vegyük fel az alábbi partial kiegészítést az API projektben a legfelső szintű kód végére:

```
public partial class Program { }
```

Az integrációs tesztünkhöz az in-process teszt szervert egy `WebApplicationFactory<TEntryPoint>` leszármazott osztály fogja létrehozni. Ez a segéd ősosztály a fenti Microsoft.AspNetCore.Mvc.Testing csomagból jön. Itt lehetőségünk van a teszt szerverünket konfigurálni, így akár a DI konfigurációt is.

Hozzunk létre egy osztályt a teszt projektbe `CustomWebApplicationFactory` néven, ami származzon a `WebApplicationFactory<Program>` osztályból és definiáljuk felül a `CreateHost` metódusát.

```
public class CustomWebApplicationFactory : WebApplicationFactory<Program>
{
    protected override IHost CreateHost(IHostBuilder builder)
    {
        builder.UseEnvironment("Development");
        builder.ConfigureServices(services =>
        {
            services.AddScoped(sp => new DbContextOptionsBuilder<AppDbContext>()
                .UseSqlServer(@"connection string")
                .UseApplicationServiceProvider(sp)
                .Options);
        });

        var host = base.CreateHost(builder);

        using var scope = host.Services.CreateScope();
        scope.ServiceProvider.GetRequiredService<AppDbContext>()
            .Database.EnsureCreated();

        return host;
```

```
    }
}
```

Megfigyelhetjük, hogy itt is LocalDB-t használunk (mivel integrációs teszt), de a connection stringet lecseréjük a DI konfigurációban. A connection string alapvetően egyezhet a tesztelendő projektben használttal, csak az adatbázisnevet változtassuk meg. Az adatbázis automatikusan létrejön és a migrációk is lefutnak az `EnsureCreated` meghívásával - az első lefutáskor.

> **DI Scope létrehozása**
>
> Mivel az `AppDbContext` Scoped életciklussal van regisztrálva a DI-ba, szükséges létrehozni egy scope-ot, hogy el tudjuk kérni a DI konténertől. Ezt természetesen ha HTTP kérés közben lennénk az ASP.NET Core automatikusan megtenné.

Kontrollertesztek előkészítése

Alakítsuk át a `ProductControllerTests` osztályt. Az osztály valósítsa meg az `IClassFixture<CustomWebApplicationFactory>` interfészt, amivel azt tudjuk jelezni az xUnit-nak, hogy kezelje a `CustomWebApplicationFactory` életciklusát (tesztek között megosztott objektum lesz), illetve pluszban lehetőségünk van ezt a tesztosztályokban konstruktoron keresztül elkérni.

```
public partial class ProductControllerTests : IClassFixture<CustomWebApplicationFactory>
{
    private readonly WebApplicationFactory<Program> _appFactory;

    public ProductControllerTests(CustomWebApplicationFactory appFactory)
    {
        _appFactory = appFactory;
    }
}
```

> **xUnit Fixture**
>
> Az xUnit nem tartalmaz DI konténert. Csak azok a konstruktorparaméterek töltődnek ki, amelyek a dokumentációban megtalálhatók. A `CustomWebApplicationFactory` típusú paraméter azért töltődik ki, mert az osztály az interfészében jelzi, hogy megosztott kontextusként `CustomWebApplicationFactory`-t vár.

Hozzunk létre a Bogus könyvtárral egy olyan `Faker<Product>` objektumot, amivel az API-nak küldendő DTO objektum generálását végezzük el. Azonosítóként küldjünk 0 értéket, mivel a létrehozás műveletet fogjuk tesztelni, kategória esetében pedig az 1-et, mivel a migráció által létrehozott 1-es kategóriát fogjuk tudni csak használni. A többi esetben használjuk a Bogus beépített lehetőségeit a név és a szám értékek random generálásához.

```
// ...
private readonly Faker<Product> _dtoFaker;

public ProductControllerTests(CustomWebApplicationFactory appFactory)
{
    // ...
    _dtoFaker = new Faker<Product>()
        .RuleFor(p => p.Id, 0)
        .RuleFor(p => p.Name, f => f.Commerce.Product())
        .RuleFor(p => p.UnitPrice, f => f.Random.Int(200, 20000))
        .RuleFor(p => p.ShipmentRegion,
              f => f.PickRandom<Dal.Entities.ShipmentRegion>())
        .RuleFor(p => p.CategoryId, 1)
```

```
        .RuleFor(p => p.RowVersion, f => f.Random.Bytes(5));
}
```

A kliensoldali JSON sorosítást a szerveroldallal kompatibilisen kell megtegyük. Ehhez készítsünk egy `JsonSerializerOptions` objektumot, amibe beállítjuk, hogy a felsorolt típusokat szöveges értékként kezelje. Mivel ugyanazt a példányt akarjuk használni a tesztekben, ezért a példányt a `CustomWebApplicationFactory` (mint tesztek közötti megosztott objektum) készítse el és ajánlja ki.

```
public JsonSerializerOptions SerializerOptions { get; }

public CustomWebApplicationFactory()
{
    JsonSerializerOptions jso = new(JsonSerializerDefaults.Web);
    jso.Converters.Add(new JsonStringEnumConverter());
    SerializerOptions = jso;
}
```

A `ProductControllerTests` a kiajánlott `JsonSerializerOptions`-t vegye át.

```
// ...
private readonly JsonSerializerOptions _serializerOptions;

public ProductControllerTests(CustomWebApplicationFactory appFactory)
{
    // ...
    _serializerOptions = appFactory.SerializerOptions;
}
```

> **Sorosítás beállításai**
>
> Sajnos ezt a `JsonSerializerOptions` példányt minden sorosítást igénylő műveletnél majd át kell adnunk, mivel az alapértelmezett JSON sorosítónak nincs publikusan elérhető API-ja alapértelmezett sorosítási beállítások megadásához. Ugyanakkor fontos, hogy kerüljük a `JsonSerializerOptions` felesleges példányosítását. Ugyanolyan beállításokat igénylő műveletek lehetőleg ugyanazt a példányt használják. Ezt most az XUnit megosztott kontextusával oldottuk meg.

POST művelet alapműködés tesztelése

Készítsük el az első tesztünket a `ProductController` `Post` műveletéhez. Érdemes azt az osztálystruktúrát követni, hogy minden művelethez / függvényhez külön teszt osztályokat hozunk létre, ami akár több tesztesetet is tartalmazhat. Ez a teszt osztályt beágyazott osztályként (`Post`) hozzuk létre egy külön partial fájlban (**ProductIntegrationTests.Post.cs**) a nagyobb egységhez tartozó tesztosztályon belül. Ezzel szépen strukturáltan tudjuk tartani a **Test Explorerben** (lásd később) is a teszteseteinket. Pluszban még származtassuk le a tartalmazó osztályból, hogy a tesztesetek elérhessék a fentebb létrehozott osztályváltozókat.

> **Láthatóság beágyazott osztályoknál**
>
> Érdekesség, hogy nem kell `protected` láthatóságúaknak lenniük a fenti osztályváltozóknak, ha beágyazott osztály akarja elérni azokat.

```
public partial class ProductControllerTests
{
    //...
```

```
    public class Post : ProductControllerTests
    {
        public Post(CustomWebApplicationFactory appFactory)
            : base(appFactory)
        {
        }
    }
}
```

A tesztesetek a teszt osztályban metódusok fogják reprezentálni, amelyek `[Fact]` vagy `[Theory]` attribútummal rendelkeznek. A fő különbég az, hogy a `Fact` egy statikus tesztesetet reprezentál, míg a `Theory` bemenő paraméterekkel rendelkezhet.

Elsőként az egyenes ágat teszteljük le, hogy a beszúrás helyesen lefut-e, és a megfelelő HTTP válaszkódot, a *location* HTTP fejlécet, és válasz DTO-t adja-e vissza. Hozzunk létre egy függvényt `Fact` attribútummal `Should_Succeded_With_Created` néven.

A teszteset az AAA (Arrange, Act, Assert) mintát követi, ahol 3 részre tagoljuk magát a tesztesetet.

1. Az *Arrange* fázisban előkészítjük a teszteset körülményeit.
2. Az *Act* fázisban elvégezzük a tesztelendő műveletet.
3. Az *Assert* fázisban pedig megvizsgáljuk a végrehajtott művelet eredményeit, mellékhatásait.

```
[Fact]
public async Task Should_Succeded_With_Created()
{
    // Arrange

    // Act

    // Assert
}
```

Az *Arrage*-ben kérjünk el egy a teszt szerverhez kapcsolódó `HttpClient` objektumot, illetve hozzunk létre egy felküldendő DTO-t.

```
// Arrange
var client = _appFactory.CreateClient();
var dto = _dtoFaker.Generate();
```

Az *Act* fázisban küldjünk el egy POST kérést a megfelelő végpontra a megfelelő sorosítási beállításokkal és olvassuk ki a választ.

```
// Act
var response = await client.PostAsJsonAsync("/api/products", dto, _serializerOptions);
var p = await response.Content.ReadFromJsonAsync<Product>(_serializerOptions);
```

Az *Assert* fázisban pedig fogalmazzuk meg a FluentValidation könyvtár segítségével az elvárt eredmény szabályait. Gondoljunk arra is, hogy a `Category`, `Order`, `Id` és `RowVersion` property-k esetében nem az az elvárt válasz, amit felküldünk a szerverre, ezért ezeket szűrjük le az összehasonlításból és vizsgáljuk őket külön szabállyal.

```
// Assert
response.StatusCode.Should().Be(HttpStatusCode.Created);
response.Headers.Location
    .Should().Be(
        new Uri(_appFactory.Server.BaseAddress, $"/api/Products/{p.Id}")
    );

p.Should().BeEquivalentTo(
    dto,
```

```
    opt => opt.Excluding(x => x.Category)
        .Excluding(x => x.Orders)
        .Excluding(x => x.Id)
        .Excluding(x => x.RowVersion));
p.Category.Should().NotBeNull();
p.Category.Id.Should().Be(dto.CategoryId);
p.Orders.Should().BeEmpty();
p.Id.Should().BeGreaterThan(0);
p.RowVersion.Should().NotBeEmpty();
```

> **Fluent Assertions és Nullable Reference Types**
>
> A Fluent Assertions (nem preview verziója) jelenleg még nem működik együtt a nem nullozható referencia típusokkal kapcsolatos ellenőrzési logikákkal, így az *Assert* részen kaphatunk ennek kapcsán figyelmeztetéseket `Should().NotBeNull()` hívások után is.

A POST művelet megváltoztatná az adatbázis állapotát, amit célszerű lenne elkerülni. Ezt legegyszerűbben úgy érhetjük el, hogy nyitunk egy tranzakciót a tesztben, amit nem commitolunk a teszt lefutása során. Ehhez vegyük fel az alábbi utasításokat az *Arrange* fázisban.

```
// Arrange
_appFactory.Server.PreserveExecutionContext = true;
using var tran = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled);

var client = _appFactory.CreateClient();
var dto = _dtoFaker.Generate();
```

Tranzakciót a .NET `TransactionScope` osztállyal fogunk most nyitni, amin engedélyezzük az aszinkron támogatást is. Ahhoz pedig, hogy a tesztben létrehozott tranzakció érvényre jusson a teszt szerveren is, a `PreserveExecutionContext` tulajdonságot be kell kapcsoljuk.

Próbáljuk ki a menu:Test[Run All Test] menüpont segítségével. A Test Explorerben figyeljük meg az eredményt.

POST művelet hibaág tesztelése

Készítsünk egy tesztesetet, ami a hibás terméknév ágat teszteli le. Mivel ez két esetet is magában foglal (null, üres string), használjunk paraméterezhető tesztesetet, tehát `Theory`-t. A teszteset bemenő paramétereit többféleképpen is meg lehet adni. Mi most válasszuk az `InlineData` megközelítést, ahol attribútumokkal a teszteset fölött közvetlenül megadhatóak a bemenő paraméter értékei. Ilyen esetben az attribútumban megadott értékeket a teszt metódus paraméterlistáján kell elkérjük. Esetünkben a név hibás értékeit várjuk első paraméterként, második paraméterként pedig az elvárt hibaüzenetet.

```
[Theory]
[InlineData("", "Product name is required.")]
[InlineData(null, "Product name is required.")]
public async Task Should_Fail_When_Name_Is_Invalid(string name, string expectedError)
{
    // Arrange

    // Act

    // Assert
}
```

Az előző tesztesethez hasonlóan hozzunk létre a teszt szervert és a DTO-t, de most a nevet a paraméter alapján töltsük fel. Bár elvileg nem lenne szükséges tranzakciókezelés, hiszen nem szabadna adatbázis módosításnak történnie, a biztonság kedvéért implementáljuk itt is a tranzakciókezelést.

```
// Arrange
 _appFactory.Server.PreserveExecutionContext = true;
using var tran = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled);
var client = _appFactory.CreateClient();
var dto = _dtoFaker.RuleFor(x => x.Name, name).Generate();
```

Az *Act* fázisban annyi a különbség, hogy most `ValidationProblemDetails` objektumot várunk a válaszban.

```
// Act
var response = await client.PostAsJsonAsync("/api/products", dto, _serializerOptions);
var p = await response.Content
   .ReadFromJsonAsync<ValidationProblemDetails>(_serializerOptions);
```

Az *Assert* fázisban pedig a HTTP státuszkódot és a `ProblemDetails` tartalmára vizsgáljunk.

```
// Assert
response.StatusCode.Should().Be(HttpStatusCode.BadRequest);

p.Status.Should().Be(400);
p.Errors.Should().HaveCount(1);
p.Errors.Should().ContainKey(nameof(Product.Name));
p.Errors[nameof(Product.Name)].Should().ContainSingle(expectedError);
```

Próbáljuk ki a menu:Test[Run All Test] menüpont segítségével. Figyeljük meg a tesztek hierarchiáját is, a POST művelethez kapcsolódó tesztek egy csoportba lettek összefogva a beágyazott osztály mentén.

Transzakciókezelés kódduplikáció

Észrevehetjük, hogy a tranzakciókezeléssel kapcsolatos kódot duplikáltuk, ennek elkerülésére például például tesztfüggvényre tehető attribútumot vezethetünk be.

2.10.5 Naplózás

A tesztek üzeneteket naplózhatnak egy speciális tesztkimenetre. Ehhez minden tesztosztály példány kap(hat) egy saját `ITestOutputHelper` példányt a konstruktoron keresztül. Vezessük be az új konstruktorparamétert a tesztosztályban és az ősosztályában is.

```
private readonly ITestOutputHelper _testOutput;

public ProductControllerTests(
   CustomWebApplicationFactory appFactory,
   ITestOutputHelper output)
{
   //...
   _testOutput = output;
}
```

Post beágyazott típus konstruktora

```
public Post(CustomWebApplicationFactory appFactory, ITestOutputHelper output)
   : base(appFactory, output)
{ }
```

Próbaképp írjunk ki egy üzenetet a `ProductControllerTests` konstruktorában.

```
output.WriteLine("ProductControllerTests ctor");
```

Ellenőrizzük, hogy a tesztek lefuttatása után *Test Explorer*-ben megjelennek-e az üzenetek a *Test Detail Summary* ablakrész *Standard output* szekciójában. Ebből láthatjuk, hogy minden tesztfüggvény, sőt minden tesztfüggvény változat (a *Theory* minden bemeneti adatsora egy külön változat) meghívásakor lefut a konstruktor.

Ugyanerre a kimenetre kössük rá a szerveroldali naplózást, hogy a tesztek lefutása mellett ezek a naplóüzenetek is megjelenjenek. Ehhez telepítsünk egy segédcsomagot a tesztprojektbe.

```
<PackageReference Include="MartinCostello.Logging.XUnit" Version="0.3.0" />
```

A `ProductControllerTests` konstruktorában kössük össze a két paramétert, a `CustomWebApplicationFactory` és az `ITestOutputHelper` példányt a fenti segédcsomag (`AddXUnit` metódus) segítségével. A tesztszerver naplózó alrendszerének adjuk meg kimenetként az xUnit tesztkimenetét.

```
_appFactory = appFactory
    .WithWebHostBuilder(builder =>
    {
        builder.ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddXUnit(output);
        });
    });
```

Ellenőrizzük, hogy a tesztek lefuttatása után *Test Explorer*-ben megjelennek-e a szerveroldali üzenetek is.

A végállapot elérhető a kapcsolódó GitHub repóban.

3. Zárszó

Jegyzetünk - és azt körülölelő, Szoftverfejlesztés .NET platformra tantárgyunk - célja, hogy a .NET világa iránt érdeklődő hallgatók és fiatal szakemberek mélyebben elsajátíthassák a .NET, Entity Framework Core és ASP.NET Core ismereteiket. Bár a jegyzetben kisebb-nagyobb hibák előfordulhatnak, reméljük, hogy tényszerű tévedéseket nem vétettünk. Publikálásával szeretnénk, ha a .NET iránt érdeklődők nagyobb köre lelné örömét abban, hogy elmélyülhet e lenyűgöző ökoszisztémában.

A jegyzet publikálását korábbi hallgatóink eredményei, sikerei is nagyban inspirálták.