ASP.NET Core MVC: adatmodell kialakítása, Entity Framework

Az első gyakorlat keretében elkezdünk egy .NET Core MVC projektet, koncentrálva elsősorban az **Entity Framework Core** használatára és a *code-first* elvű adatbázis létrehozására. Az alkalmazás egy modell rétegből fog állni, ami alapján létrehozzuk az adatbázist, valamint egy szolgáltatásokat tartalmazó osztályból, amellyel az adatbázis-műveleteket végezzük el.

Az alkalmazás tennivalólistákat fog tartalmazni, amelyeket a nevük ír le. A listákhoz tetszőleges számú elem tartozhat, amelyek névvel, határidővel és opcionális leírással rendelkeznek.

1 Fejlesztői környezet

Telepítsük a Visual Studio 2019-et, vagy módosítsuk a korábbi telepítést (ha szükséges), a Visual Studio Installer futtatásával. A kurzushoz a .NET desktop development mellett az ASP.NET and web development workload-ok telepítésére lesz szükség.



Figure 1: Szükséges workloadok.

2 Projekt létrehozása

A File \rightarrow New \rightarrow Project menüben válasszuk ki az ASP.NET Core Web Application típusú projektet! Használhatjuk a keresőt segítségül. Fontos, hogy .NET Core és ne .NET Framework projekt kerüljön létrehozásra!

Create a new pr	oject	
Create a new pr	ojeci	ASP × Clear all
Recent project templates		All languages - All platforms - All project types -
ASP.NET Core Web Application ASP.LET Core Web Application Class Library (.NET Core)	C#	ASP.NET Core Web Application Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux. Ce Linux macOS Windows Cloud Service Web
		 ASP.NET Web Application (.NET Framework) Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET. Visual Basic Windows Cloud Web
		ASP.NET Web Application (.NET Framework) Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET. C= Windows Cloud Web
		ASP.NET Core Web Application Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Anyular, React, or React + Redux. F# Linux macOS Windows Web
		A Blazor App
		Next

Figure 2: Projekttípus kiválasztása Visual Studioban.

A következő lépésben megadhatjuk a projekt nevét és helyét. Ha ez is megvan, akkor a következő lépésben ügyeljünk arra, hogy mindenképp legalább a .NET Core 3.1-es verziót válasszuk ki. Ha valaki szeretné, a legújabb Visual Studioval tudja használni a .NET Core 5.0-át is, de a kurzus a 3.1-es verziót fogja alkalmazni.

		 0.000	Service	1100		
TodoList						
Location						
C:\Users\roxanaprovender\source\repos			·			
Solution						
Create new solution			•			
Solution name 🚺						
TodoList						
Place solution and project in the same direct	tory					

Figure 3: Projekt útvonalának megadása.

ET Coi	re v ASP.NET Core 3.1 v		
	Empty An empty project template for creating an ASP.NET Core application. This template does not have any content in it.		Authentication No Authentication
⊷ 0	API A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service.	1	Change
⊕_	This template can also be used for ASP.NET Core MVC Views and Controllers. Web Application	l	Advanced ✓ Configure for HTTPS
Æ	A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content. Web Application (Model-View-Controller)	l	Enable Docker Support (Requires Docker Desktop)
	A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTIUI HTTP services.		Linux -
A	Angular A project template for creating an ASP.NET Core application with Angular		
*	React.js		Author: Microsoft

Figure 4: .NET Core verzió és sablon (*Empty*) kiválasztása.

3 Modell réteg

A modell az adatbázistáblák leképezéseit, az entitás osztályokat fogja tartalmazni, valamint a leképezést és az adatbázisszerverhez való kapcsolódást létrehozó kontextusból. Ebben a rétegben készítünk még mintaadatokat, amelyekkel létrehozáskor töltjük fel az adatbázist.

3.1 Lista entitás

Készítsük el a listákat reprezentáló entitás osztályt!

- 1) Hozzuk létre a entitás osztályokat tartalmazó mappát a projektben!
- 2) Adjunk egy új osztályt a modellhez: List.cs
- 3) Tegyük publikussá az osztályt, ha nem lenne az!
- 4) Hozzuk létre az osztályban a következő propertyket:
 - Azonosító (Id, típusa 32 bites egész szám): a tábla elsődleges kulcsa lesz. Ezt a Key annotációval jelezzük!
 - Név (Name, típusa szöveg): a Required annotációval jelezzük, hogy új lista hozzáadásakor kötelező a mezőt kitölteni, illetve garantáljuk a MaxLength annotációval, hogy a név nem lehet 30 karakternél hosszabb.

3.2 Listaelemek

Készítsük el a listaelemeket reprezentáló entitás osztályt! A elemek sok-az-egyhez kapcsolatban állnak a listákat tartalmazó táblával (egy listához több elem tartozhat, de egy elem csak egy listához).

- 1. Adjunk egy új osztályt a modellhez: Item.cs
- 2. Tegyük publikussá az osztályt, ha nem lenne az!
- 3. Hozzuk létre az osztályban a következő propertyket:
 - Azonosító (Id, típusa 32 bites egész szám, elsődleges kulcs)
 - Név (Name, típusa szöveg, kötelező mező, maximális hossza 30 karakter)

- Leírás (Description, típusa szöveg): a DataType annotáció MultilineText típusával garantáljuk, hogy többsoros lehessen a leírás!
- Határidő (Deadline, típusa DateTime, kötelező mező)
- A elemet tartalmazó lista azonosítója (ListId, típusa 32 bites egész szám, kötelező mező)
- Az elemet tartalmazó lista (List, típusa List, virtuális mező).

Megjegyzés: Az Item osztályban a List típusú adattagot navigációs propertynek nevezik. Szerepe, hogy ez alapján az Entity Framework felismerje a kapcsolatot a List és az Item táblák között, és megfelelően beállítsa a Foreign Key constraintet a ListId-ra.

3.2.1 Nullable típusok

A C# esetén az érték szerinti típusok (mint például a primitív típusok, illetve minden olyan típus, amely a struct kulcsszóval kerül definiálásra) alapesetben nem vehetnek fel null értéket. Minden ilyen típusnak van azonban *nullable* változata, amelyet a ? operátor segítségével használhatunk (pl. az int nullable típusa az int?, teljes nevén Nullable<int>)

Az ilyen típusok a megszokott értékeiken túl a null értéket is felvehetik (pl. bool? esetén a változó értéke lehet true, false vagy null). A nullable értékek default értéke mindig a null.

Az adatbázisra való leképezéskor a nem nullable-ként definiált típusok az adatbázisban NOT NULL megszorítással fognak rendelkezni, míg a nullable típusok felvehetik a NULL értéket is. Ha azt szeretnénk tehát, hogy egy értéknek a megadása opcionális legyen, használjunk nullable típust.

A referencia szerinti típusok (pl. string vagy a class kulcsszóval definiált típusok) alapból nullable típusok, ezért nincs kérdőjeles változatuk. Ezeknek a típusoknak a default értéke szintén null. Ha azt szeretnénk, hogy az ilyen típusú propertyk ne vehessenek fel NULL értéket az adatbázisban, akkor használjuk a [Required] annotációt, ami beállítja számunkra a NOT NULL *constraintet*.

Megjegyzés: Az eddig tárgyalt nullable típusokat érték szerinti nullable típusoknak nevezzük. A C# 8.0-s verziója óta (kiadás éve: 2019) lehetőségünk van a referencia szerinti típusokból is *nullable* és *non-nullable* változatokat használni, amely így lehetővé teszi a referencia szerinti típusok inicializáltságának statikus (fordítási idejeű) ellenőrzését. Ennek használatához szükséges a projektben az ún. *nullable context* engedélyezése, ezzel most nem foglalkozunk.

3.3 Adatbázis-kontextus létrehozása

Az adatbázis-kontextus felel az adatbázishoz való kapcsolódás
ért, valamint a C# kód és az adatbázistáblák egymásra való leképezésé
ért.

- 1. Hozzunk létre egy osztályt a kontextusnak! (TodoListDbContext)
- 2. Az osztály származzon a DbContext osztályból, amely a Microsoft.EntityFrameworkCore NuGet package letöltésével érhető el a csomaggal megegyező nevű névtérben.

Megjegyzés: Ugyan hivatalosan az 5.0-ás package-k is kompatibilisek a 3.1-es .NET Core verzióval, célszerű az ugyanolyan verziószámmal rendelkező package-t használni, hogy később, több NuGet függőség esetén az esetleges verziókövetelmények közötti konfliktusokat elkerülhessük. (Azaz 3.1 .NET Core esetén az EF csomagból is a 3.1.*-et válasszuk, ha pedig az 5.0-ás .NET Core verzió mellett döntöttünk, akkor az EF-ből is mehet az 5.0.)

- 3. Hozzuk létre a listák leképezését a kontextusban! Ezt egy DbSet típusú objektumok példányosításával tehetjük meg, amelyek típusparamétere a List és Item.
- 4. Írjuk felül a DbContext-ből származó OnConfiguring metódust! Egy DbContextOptionsBuilder típusú paramétert fog várni (optionsBuilder). Az optionsBuilder UseSqlServer metódusának meghívásával konfigurálhatjuk az adatbázist*. A metódus egy connection stringet vár, amelyet ebben a feladatban konstans stringként adunk meg a metódus hívásakor, pl.:

Data Source=(localdb)\\MSSQLLocalDB;

Initial Catalog=TodoListCore3;Trusted_Connection=True;MultipleActiveResultSets=True

A connection string fontos része a Data Source, amely megadja, hogy milyen szerverhez kapcsolódva hozzuk létre az adatbázist, valamint az initial catalog, amelyben az adatbázis nevét adjuk meg. Ebben a lépésben Microsoft SQL Serverrel dolgozunk.

* A UseSqlServer metódust a Microsoft.EntityFrameworkCore.SqlServer NuGet package telepítésével érhetjük el.

3.4 Lusta betöltés

Lusta betöltéssel (*lazy loading*) a navigációs propertyken keresztül az adatok akkor kerülnek betöltésre, amikor szükség van rájuk (amikor először kiértékelésre kerülnek). Használatához telepítsük a Microsoft.EntityFrameworkCore.Proxies NuGet csomagot, majd a TodoListDbContext osztály OnConfiguring eljárásában a DbContextOptionsBuilder típusú objektumot egészítsük ki a UseLazyLoadingProxies eljárás hívással.

Ne feledkezzünk meg, hogy ilyen esetben a navigációs propertyket lássuk el a virtual kulcsszóval.

Megjegyzés: Nem feltétlenül mindig a lusta betöltés a leghatékonyabb megoldás a feladatra.

3.5 Adatbázis létrehozása migrációval

A migrációk szerepe, hogy a kontextusban beállított osztályok, illetve azok változásai az adatbázisban is megtörténjenek. A migrációk segítségével folyamatosan bővíthetjük/módosíthatjuk az adatbázisunk sémáját anélkül, hogy ki kellene törölnünk azt, ezáltal esetlegesen adatvesztést okozva.

(*Megjegyzés:*: A következő leírás a legfontosabb lépéseket mutatja be, a migrációk részletes bemutatása a munkafüzethez tartozó videóban tekinthető meg.)

- 1. Telepítsük a Microsoft.EntityFrameworkCore.Tools NuGet package-et! Ebben elérhetőek az adatbázis *Package Manager Console*-ból való manipulációjához szükséges parancsok.
- 2. Nyissuk meg a View \rightarrow Package Manager Console-t!
- 3. Hozzunk létre migrációt a következő paranccsal: Add-Migration {tetszőleges név}.
- 4. Az előző lépéssel létrejöttek az adatbázist leíró C# kódok, de a változtatások még nem kerültek mentésre az adatbázisba. Az adatbázis tényleges létrehozásához futtassuk az Update-Database parancsot!
- 5. Az adatbázis tartalmát a View \rightarrow SQL Server Object Explorer ablakban tekinthetjük meg.

Megjegyzés: az Entity Framework Core parancsait nem csak a Visual Studio *Package Manager Console*-jából érhetjük el, hanem az operációs rendszer termináljából is (pl. Linux operációs rendszer alatt), amennyiben telepítettük az EF Core konzolos eszközeit, amelyet megtehetünk lokálisan a projekthez, vagy globálisan a felhasználói fiókunkhoz. Javasolt az utóbbi, hiszen gyakran lehet szükségünk ezekre az eszközökre:

dotnet tool install --global dotnet-ef

Ilyenkor az előbbi két parancs megfelelője:

- dotnet ef migrations add {tetszőleges név}
- dotnet ef database update

3.6 Az adatbázis feltöltése mintaadatokkal

- 1. Hozzunk létre egy publikus, statikus osztályt DbInitializer néven!
- 2. Az osztálynak legyen egy publikus, statikus Initialize nevű metódusa, amely nem ad vissza semmit, és egy adatbázis-kontextust vár paraméterül.
- 3. Az Initialize metóduson belül győződjünk meg róla, hogy az adatbázis létezik és az összes migrációt alkalmaztuk (Database.Migrate).
- 4. Amennyiben az adatbázisban már vannak adatok (pl. a List tábla nem üres), térjük vissza.

- 5. Hozzunk létre 1-2 új listát néhány elemmel. Minden kötelező adatot adjunk meg.
- 6. A listákat adjuk hozzá az adatbázis List táblájához, amelyet az adatbázis-kontextuson keresztül érhetünk el.
- 7. Mentsük el a kontextus változtatásait (SaveChanges)!

4 Szolgáltatás (*service*) osztály létrehozása

Ebben az interface-ben definiálunk műveleteket, amelyek az adatbázis-manipulációt végzik. Szigorúan véve a modell réteg része, nem képez külön réteget.

- 1. Adjunk a projekthez egy Services nevű mappát!
- 2. Készítsünk egy új osztályt a Services-ben (TodoListService)!
- 3. Tegyük publikussá az osztályt!
- 4. Az osztály konstruktora paraméterként kapjon egy adatbázis-kontextus példányt, amin keresztül az osztály metódusai el tudják érni az adatbázist.

4.1 CRUD műveletek

Definiáljunk néhány metódust, amelyek alapvető műveleteket végeznek az entitásokon! Négy művelethez definiálunk metódusokat: adatok hozzáadása, lekérése, módosítása és törlése.

1. Create

- Legyen egy AddItemToList metódusunk, amely egy listaelemet vár paraméterül, és nem ad vissza semmit. Ha az elem nem null, adjuk hozzá az Items entitáshoz! Ne felejtsük el elmenteni a változtatást.
- 2. Read
 - Készítsünk egy GetLists nevű metódust, amely listázza a megadott stringet tartalmazó nevű listákat. A metódus egy alapértelmezetten üres stringet vár paraméterül, és listaentitások egy listáját adja vissza. Egy Linq lekérdezéssel kérjük le az adatbázisból azokat a listákat, amelyek nevében szerepel a paraméterként kapott string! A lista legyen a listanevek szerint növekvő sorrendbe rendezve.
 - Készítsünk egy GetListByID nevű metódust, amely egy egész számot vár paraméterként (egy azonosítót), és egy modellbeli listát ad vissza. Egy Linq lekérdezéssel kérjük le az adatbázisból azt a listát és elemeit, amelynek az azonosítója megegyezik a paraméterként kapott számmal!
 - Definiáljunk egy GetItemsByListID nevű metódust, amely a megadott azonosítójú lista elemeit adja vissza! Egy Linq lekérdezéssel keressük ki az azonosítónak megfelelő listát, és adjuk vissza a hozzá tartozó elemeket!
- 3. Update
 - Definiáljunk egy ChangeListName nevű metódust, amellyel megváltoztathatjuk egy lista nevét! A metódus egy listaazonosítót és egy új nevet vár paraméterül, és nem ad vissza semmit. Egy Linq lekérdezéssel keressük meg az azonosítónak megfelelő listát, és változtassuk meg a nevét! Mentsük el a változtatást.
- 4. Delete
 - Definiáljunk egy RemoveItemByName nevű metódust, amely kitörli a megadott listából a keresett nevű listaelemet! A metódus egy lista azonosítót és elem nevet vár paraméterül, és nem ad vissza semmit. A név alapján egy Linq lekérdezéssel keressük ki a megadott nevű elemet, majd töröljük az Items entitásból! Mentsük el a változtatást.

A TodoListService-ben definiált műveleteket hívjuk meg a Program.cs osztály Main metódusában! Szükség lesz egy példányra a TodoListDbContext-ből, amivel meghívhatjuk az adatbázist feltöltő statikus metódust, valamint a service ezen keresztül fogja elérni az adatbázist.

5 SQLite támogatás

Tegyük cross-platformmá az alkalmazást: biztosítsunk lehetőséget SQLite szerverhez való kapcsolódásra!

- 1. A modellben hozzunk létre egy fájlt DbType.cs néven!
- 2. A fájlban legyen egy publikus, $\tt DbType$ nevű enum, amelynek legyen két értéke: SqlServer és Sqlite.
- 3. Adjunk a projekthez egy fájlt appsettings.json néven! Ebbe a fájlba emeljük át az MSSQL szerverhez tartozó connection stringet (a szekció neve legyen ConnectionStrings), valamint adjunk hozzá egy új connection stringet, amivel egy SQLite adatbázishoz lehet majd kapcsolódni! Pl. "Data Source=TodoListCore3.db"
- 4. Legyen még egy kulcs-érték párunk a fájlban, ahol a kulcs DbType, az érték (SQLite támogatás esetén) Sqlite.
- 5. Adjuk a projekthez a Microsoft.Extensions.Configuration.Json NuGet csomagot, amellyel könnyedén strukturáltan beolvashatunk JSON konfigurációs állományokat.
- 6. Az OnConfiguring metódust egészítsük ki úgy, hogy SQLite szerverhez is tudjunk kapcsolódni (UseSqlite)! Ehhez a Microsoft.EntityFrameworkCore.Sqlite NuGet package-re lesz szükség. A metódust egészítsük ki egy ConfigurationBuilder típusú objektummal, amely beállítja a projektkönyvtárat alapértelmezett útvonalnak (SetBasePath(Directory.CurrentDirectory())), és itt keresi a connection stringeket tartalmazó JSON fájlt (AddJsonFile("appsettings.json"))!
- 7. Készítsünk egy IConfigurationRoot típusú objektumot úgy, hogy meghívjuk az előző lépésben létrehozott objektum Build parancsát.
- 8. Ezen az objektumon keresztül elérhetjük az appsettins.json DbType kulcsához tartozó értéket (GetValue), valamint az ennek megfelelő connection stringet (GetConnectionString), amivel helyettesíthetjük az eddigi beégetett értéket.

5.1 Ajánlás SQLite adatbázis kezelőre

A SQLite adatbázisok megtekintéséhez (esetleges szerkesztéséhez) szükséges valamilyen programot telepítenünk. A Visual Studio kiegészítőjeként elérhető például az SQLite/SQL Server Compact Toolbox.