Rendszerterv

Online Gallery

Szoftverarchitektúrák (VIAUMA06) tárgy házi feladat

Markovics Gergely (ME7KKE)

Nagy Gellért (A2PD2Q)

**Tartalom**

[Megvalósítás 3](#_Toc120382096)

[Backend 3](#_Toc120382097)

[DAL réteg 3](#_Toc120382098)

[Entity Framework 3](#_Toc120382099)

[Adatbázis séma 4](#_Toc120382100)

[Felhasználó entitás 6](#_Toc120382101)

[Szerep entitás 6](#_Toc120382102)

[Felhasználó-szerep entitás 7](#_Toc120382103)

[Album entitás 7](#_Toc120382104)

[Kép entitás 8](#_Toc120382105)

[Kedvenc album entitás 8](#_Toc120382106)

[Repository és Unit of work 9](#_Toc120382107)

[Fájlok tárolása 9](#_Toc120382108)

[Konfiguráció 9](#_Toc120382109)

[BLL réteg 9](#_Toc120382110)

[Mediator minta 9](#_Toc120382111)

[Automapper 9](#_Toc120382112)

[Validálás 10](#_Toc120382113)

[API réteg 10](#_Toc120382114)

[Kontrollerek 10](#_Toc120382115)

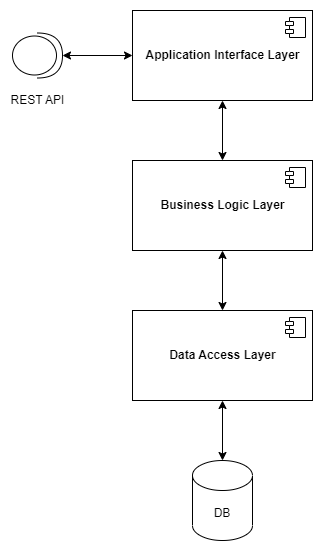
[Autentikálás 10](#_Toc120382116)

[Frontend 10](#_Toc120382117)

# Megvalósítás

## Backend

Az alkalmazás szerveroldali komponensét .NET 6 platformon implementáljuk. Ez egy cross-platform megoldás, tehát lehetőségünk van a program tetszőleges operációs rendszeren való futtatására. Az alkalmazás a háromrétegű architektúra elvét követi: definiálunk adathozzáférési, üzleti logikai és interfész rétegeket, ezek a solutionben külön projektek. Közvetlen módon mindegyik réteg csak az eggyel alacsonyabb szintűtől függ. A szerver a szolgáltatást egy REST API típusú interfészen keresztül publikálja. Az alábbiakban az egyes rétegek szerepeit részletezzük.



A szerver architektúrája

## DAL réteg

#### Entity Framework

Az adatbázis műveletek megvalósításához a .NET Entity Framework keretrendszerét használjuk. Ez egy ORM leképezési réteget biztosít a C# osztályok és az adatbázis táblái között. A lekérdezések megfogalmazása során ezért nem kell sima SQL nyelvet használnunk, elég a C# entitásokon végrehajtani ezeket, az átfordítást az Entity Framework elvégzi számunkra. Lehetőségünk van arra is, hogy a definiált C# osztályok alapján hozzuk létre az adatbázis sémáját (code-first megközelítés). Ekkor a keretrendszer elvégzi a C# és SQL típusok megfeleltetését, az öröklési hierarchiák és a táblák közötti kapcsolatok felderítését. Ez utóbbi névkonvenció alapján történik, ezért fontos, hogy a változók számára megfelelő neveket válasszunk, amennyiben ki akarjuk ezt használni (pl. kulcs esetén Id postfix, lista esetén {entitásnév}s). Ha a modell osztályokban leírt sémát szeretnénk átültetni az adatbázisba, akkor azt egy migráció létrehozásával, majd annak alkalmazásával tehetjük meg. A séma esetleges hibáit (pl. nem felderíthető kapcsolat, ciklikus kaszkád törlés) is ekkor hozza tudomásunkra a rendszer. Amennyiben az entitásokat manuálisan (is) szeretnénk konfigurálni, akkor ezt megtehetjük többek között az adatbázist reprezentáló osztály OnModelCreating metódusában.

#### Adatbázis séma

Az alkalmazás adatbázis sémája az alábbi képen látható. A továbbiakban ismertetjük azokat az entitásokat, amelyeket az alkalmazás használ (a felhasználókezelést megvalósító AspNetCore.Identity könytár adattáblái közül nem az összeset használjuk fel).



Az alkalmazás adatbázis sémája

#### Felhasználó entitás

**Célja:** A regisztrált felhasználókat tároljuk ebben a táblában.

**Leképezése a kódban:** Az ApplicationUser osztály

**Leképezése az adatbázisban:** Az AspNetUsers tábla 1-1 sora

**Tulajdonságai:**

|  |  |  |
| --- | --- | --- |
| **C# mezőnév** | **C# adattípus** | **SQL Server adattípus** |
| Id | Guid | UNIQUEIDENTIFIER PRIMARY KEY |
| FirstName | string | NVARCHAR (MAX) |
| LastName | string | NVARCHAR (MAX) |
| UserName | string | NVARCHAR (256) |
| PasswordHash | string | NVARCHAR (MAX) |
| Email | string | NVARCHAR (256) |
| … | … | … |

A fenti táblázat csak az alkalmazás szempontjából fontos tulajdonságokat sorolja fel.

* Az Id mező tárolja az entitás egyedi azonosítóját. Ez alapján lehet az adatbázisban az entitást azonosítani.
* A FirstName mező tartalmazza a felhasználó keresztnevét.
* A LastName mező tartalmazza a felhasználó vezetéknevét.
* A UserName mező tartalmazza a felhasználó által választott felhasználónevet.
* A PasswordHash mező tartalmazza a fiókhoz tartozó jelszó hashét.
* A felhasználó email címét az Email mező tárolja.

#### Szerep entitás

**Célja:** A felhasználók lehetséges szerepköreit rögzíti. A jogosultságkezeléshez használatos.

**Leképezése a kódban:** Az ApplicationRole osztály

**Leképezése az adatbázisban:** Az AspNetRoles tábla 1-1 sora

**Tulajdonságai:**

|  |  |  |
| --- | --- | --- |
| **C# mezőnév** | **C# adattípus** | **SQL Server adattípus** |
| Id | Guid | UNIQUEIDENTIFIER PRIMARY KEY |
| Description | string | NVARCHAR (MAX) |
| Name | string | NVARCHAR (256) |
| NormalizedName | string | NVARCHAR (256) |
| ConcurrencyStamp | string | NVARCHAR (MAX) |

* Az Id mező tárolja az entitás egyedi azonosítóját. Ez alapján lehet az adatbázisban az entitást azonosítani.
* A Description mező egy rövid leírást tartalmaz az adott szerepkörről.
* A szerepkör nevét a Name mező tartalmazza, ennek segítségével hivatkozunk rá a programban.
* A NormalizedName mező a szerepkör nevét tartalmazza csupa nagybetűvel írva.
* Az esetleges konkurrencia problémákat hivatott megelőzni a ConcurrencyStamp mező.

#### Felhasználó-szerep entitás

**Célja:** Kapcsolótábla, amire a felhasználók és a szerepkörök N-N kapcsolata miatt van szükség. Összerendeli a felhasználókat a hozzá tartozó szerepkörökkel.

**Leképezése a kódban:** Nem hivatkozunk rá expliciten, navigációs tulajdonságként érhető el.

**Leképezése az adatbázisban:** Az AspNetUserRoles tábla 1-1 sora

**Tulajdonságai:**

|  |  |  |
| --- | --- | --- |
| **C# mezőnév** | **C# adattípus** | **SQL Server adattípus** |
| UserId | Guid | UNIQUEIDENTIFIER FOREIGN KEY |
| RoleId | Guid | UNIQUEIDENTIFIER FOREIGN KEY |

* A UserId mező tárolja, hogy mely felhasználót kapcsoljuk hozzá az adott szerepkörhöz.
* A RoleId mező tárolja, hogy mely szerepkört kapcsoljuk hozzá az adott felhasználóhoz.

#### Album entitás

**Célja:** A felhasználók által létrehozott albumokat tárolja.

**Leképezése a kódban:** Az Album osztály

**Leképezése az adatbázisban:** Az Albums tábla 1-1 sora

**Tulajdonságai:**

|  |  |  |
| --- | --- | --- |
| **C# mezőnév** | **C# adattípus** | **SQL Server adattípus** |
| Id | Guid | UNIQUEIDENTIFIER PRIMARY KEY |
| Creator | ApplicationUser | UNIQUEIDENTIFIER FOREIGN KEY |
| Name | string | NVARCHAR (MAX) |
| IsPrivate | bool | BIT |
| LikeCount | int | INT |

* Az Id mező tárolja az entitás egyedi azonosítóját. Ez alapján lehet az adatbázisban az entitást azonosítani.
* A Creator navigációs tulajdonság hivatkozik a létrehozó felhasználó entitásra. Ez adatbázis szinten egy külső kulcs formájában jelenik meg.
* A Name mező tartalmazza az album felhasználó által megadott nevét.
* Az IsPrivate bool flag jelzi, hogy az adott album privát-e.
* A LikeCount mezőben tároljuk, hogy hány ember like-olta az albumot.

#### Kép entitás

**Célja:** A felhasználók által feltöltött egyes képek adatait tároljuk itt.

**Leképezése a kódban:** A Picture osztály

**Leképezése az adatbázisban:** A Pictures tábla 1-1 sora

**Tulajdonságai:**

|  |  |  |
| --- | --- | --- |
| **C# mezőnév** | **C# adattípus** | **SQL Server adattípus** |
| Id | Guid | UNIQUEIDENTIFIER PRIMARY KEY |
| Album | Album | UNIQUEIDENTIFIER FOREIGN KEY |
| PhysicalPath | string | NVARCHAR (MAX) |
| DisplayPath | string | NVARCHAR (MAX) |
| Size | double | FLOAT (53) |
| FileExtension | string | NVARCHAR (MAX) |

* Az Id mező tárolja az entitás egyedi azonosítóját. Ez alapján lehet az adatbázisban az entitást azonosítani.
* A Creator navigációs tulajdonság hivatkozik a tartalmazó albumra. Ez adatbázis szinten egy külső kulcs formájában jelenik meg.
* A kép abszolút fizikai elérési útvonalát a PhysicalPath tulajdonság tárolja.
* A DisplayPath mező a kép relatív útvonalát tárolja, amit az alap URL mögé illesztve elérhetjük azt a szerverről.
* A Size tulajdonság a kép méretét tárolja MB-ban.
* A FileExtension a kép formátumát rögzíti.

#### Kedvenc album entitás

**Célja:** A felhasználók és a kedvenc albumok összerendelő táblázata.

**Leképezése a kódban:** FavoritedBy navigációs tulajdonság az Album osztályban és FavoritedAlbums navigációs osztály az ApplicationUser osztályban

**Leképezése az adatbázisban:** A Favorite tábla 1-1 sora

**Tulajdonságai:**

|  |  |  |
| --- | --- | --- |
| **C# mezőnév** | **C# adattípus** | **SQL Server adattípus** |
| FavoritedAlbumsId | Album | UNIQUEIDENTIFIER FOREIGN KEY |
| FavoritedById | ApplicationUser | UNIQUEIDENTIFIER FOREIGN KEY |

* A FavoritedAlbumsId mező tárolja, hogy mely albumot kapcsoljuk hozzá az adott felhasználóhoz. Ez adatbázis szinten egy külső kulcs formájában jelenik meg.
* A FavoritedById mező tárolja, hogy mely felhasználót kapcsoljuk hozzá az adott albumhoz. Ez adatbázis szinten egy külső kulcs formájában jelenik meg.

#### Repository és Unit of work

Az entitásokkal való műveletek során vannak olyanok, amiket szinte mindig definiálni kell. Ilyen a valamilyen feltétel alapján való keresés, törlés, módosítás és létrehozás. Azért, hogy ezeket a műveleteket csak egyszer definiáljuk, létrehozunk egy generikus repository osztályt, és ezeket a közös műveleteket itt implementáljuk.

Ha egy művelet végrehajtásához sok különböző entitással kell dolgozunk, akkor az ezekkel paraméterezett repository osztályokat egyesével kéne létrehoznunk. Annak érdekében, hogy ezt megspóroljuk és az adatelérést absztraháljuk, létrehozunk egy Unit of work osztályt, ami tartalmazza az összes entitáshoz tartozó repository osztályt. Ekkor bármilyen adatbázis művelet végrehajtása esetén elég csak ebből az osztályból létrehozni egy példányt, így elérve az összes definiált entitást.

#### Fájlok tárolása

A képek tárolásához az operációs rendszer fájlrendszerét használjuk. A pontos fizikai útvonal az appsettings.json fájlban található értéktől függ. A feltöltött fájlokat felhasználók szerint rendezzük: egy adott kép a feltöltő azonosítójával elnevezett mappába kerül.

Ahhoz, hogy a webszerver kiszolgálja ezeket a kéréseket, el kell helyeznünk a megfelelő middleware-t a HTTP kérés pipelineban. Ennek a konfigurálásánál megadunk egy fizikai útvonalat és egy kérés útvonalat. Ezután a kérés útvonalához hozzáillesztve a fájl elérésének relatív útvonalát az adott mappából kapunk egy arra mutató linket.

#### Konfiguráció

Az alkalmazás futása során szükségünk lehet bizonyos konfigurációs értékekre, amik bizonyos beállításokat tartalmaznak (pl. mi legyen a maximális feltölthető fájlméret). Ezeket az appsettings.json fájlban tároljuk, azonban mi szeretnénk ezekhez az értékekhez plusz üzleti logikát rendelni, ezért bevezetünk egy konfigurációs szolgáltatást. Az alkalmazás egyéb részei ezen keresztül férnek hozzá ezekhez a konfigurációs értékekhez.

### BLL réteg

#### Mediator minta

Az alkalmazás az üzleti logikai műveletek elvégzéséhez a mediator mintát használja, a MediatR csomag használatával. Ennek lényege, hogy az API réteg nem a BLL réteg szolgáltatásaival kommunikál közvetlenül, hanem egy IMediator típusú objektummal, és ennek az objektumnak a feladata, hogy a kérést a megfelelő szolgáltatás megkapja és végrehajtsa.

Ennek megfelelően a BLL rétegban definiálunk parancsokat (command) és kéréseket (query), amiket szétválasztunk az egyes domain entitásokra tekintve (jelen esetben felhasználó és album). A két típus között az a különbség, hogy a command valamilyen adatmódosítással járó műveletet valósít meg, míg a query csak adatlekérést. Ezen az objektumok tartalmazzák mindazt az adatot, ami az adott művelet végrehajtásához szükséges. Ezeket az objektumokat kapják meg a handler osztályok, ahol a különböző típusú parancs és kérés osztályokat kezelő üzleti logikát implementáljuk.

#### Automapper

Az entitás és a kommunikációt végrehajtó osztályok közötti átalakítást nagyon sokszor elvégezzük, ezért célszerű ezt a logikát központosítani. Az egyik legnépszerűbb megoldás erre az Automapper könyvtár, ami a megfeleltetési logika széles palettáját támogatja, kezdve az egyszerű név szerinti megfeleltetési logikától a külön osztályokban definiált, tulajdonság vagy típus szerinti átalakításokig. A különböző domain entitásokhoz külön ún. profil osztályokat hozunk létre, amiknek konstruktorában megadhatjuk az adott entitás átalakítási logikáit. Amennyiben egy adott tulajdonsághoz vagy típushoz komplex műveleteket kell végezni, akkor célszerű az adott logikát kiszervezni egy külön osztályba (ahol akár elkérhetünk szolgáltatásokat a DI konténtertől), majd ezt az osztályt behivatkozni az átalakítási logikában.

#### Validálás

Az alkalmazás futása során különböző validációkra van szükségünk (pl. egy album törlését csak a létrehozó felhasználó vagy egy admin teheti meg, nem bármely bejelentkezett felhasználó). Ezeket célszerű kiszervezni más osztályokba, hogy a handler osztály a művelet végrehajtására tudjon koncentrálni. Ennek érdekében definiálunk egy IValidator interfészt, aminek egy bool visszatérésű Validate metódusa van, a különböző implementációk pedig ezt fogják megvalósítani. Annak érdekében, hogy ezek között kapcsolatokat is definiálhassunk (pl. vagy, és), létrehozunk “összekötő” osztályokat, amik kapnak két IValidator típusú objektumot, és a Validate metódusukban a megfelelő logikai művelettel kapcsolják azokat össze. Ennek segítségével egy IValidator típusú objektummal is leírhatjuk az összes, adott művelethez szükséges validációs kritériumot.

### API réteg

#### Kontrollerek

Az alkalmazás REST API-n keresztül érhető el. A beérkező kéréseket kontroller osztályok fogadják, amik egy adott entitáson végezhető műveleteket foglalják egy csoportba (pl. album létrehozása, törlése, módosítása). A kontroller metódusait ellátjuk annotációkkal, amely megfelelteti azt egy HTTP ige és útvonal kombinációjával. Ezen osztályok csak az Imediator objektummal keresztül kommunikálnak a BLL komponenseivel, ezért minden hívásnak nagyon hasonló annak kezelése: létrehozzuk a kéréshez tartozó kérés/parancs objektumot, annak átadva minden adatot amire szüksége van, majd ezen létrehozott objektumot átadjuk a mediatornak, aki a megfelelő handler metódust hívja meg ennek hatására.

Az egyes kontroller osztályokra vagy annak metódusaira elhelyezzük az Authorize vagy AllowAnonymous attribútumot attól függően, hogy az adott művelethez be kell-e jelentkezve lennie az adott felhasználónak. Az Authorize attribútumnak megadjuk azt is, hogy milyen szerepkörben kell lennie a bejelentkezett felhasználónak (sima, admin, bármilyen) a sikeres műveletvégzéshez.

#### Autentikálás

Az autentikáláshoz IdentityServert használtunk, ami JWT alapon működik. Ez a könyvtár rengeteg különféle, OAuth 2.0 kompatibilis autentikációs folyamatot tartalmaz különféle igényekhez. Mivel nekünk csak jelszó alapú bejelentkezés kell, ezért a Resource owner password folyamatot valósítjuk meg. Ennek során egy sikeres felhasználónév (vagy email) és jelszó páros megadásakor a felhasználó kap egy JWT-t, ami tartalmazza az ő adatait, többek között az id-t, a nevét és szerepkörét. A kliens a szerepkör alapján jeleníthet meg vagy rejthet el bizonyos funkciókat a felhasználói felületen, illetve a szerver ezt a tokent használhatja az azonosításhoz és validáláshoz.

Az alkalmazás elindításakor az appsettings.json fájlban található szerepkörökkel és felhasználó fiókokkal feltöltjük az adatbázist, így adhatunk hozzá új adminisztrátori fiókokat. A regisztrált felhasználók mindig normál szerepkörrel rendelkeznek.

#### Hibakezelés

A hibakezelést a Helland.Middleware.ProblemDetails könyvtár segítségével valósítottuk meg. Ennek segítségével elhelyezünk egy middlewaret a HTTP kérés pipelineban, ahol megadhatjuk az alkalmazás futása során előforduló kivétel típusokhoz tartozó válasz kódot és üzenetet.

## Frontend

**Telepítési leírás**

**A program készítése során felhasznált eszközök**

**Összefoglalás**

**Továbbfejlesztési lehetőségek**

**Hivatkozások**