

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Schütz Máté

**Webes alkalmazás készítése**

Konzulens

Albert István

BUDAPEST, 2021

Tartalomjegyzék

[1. Bevezetés 3](#_Toc90331962)

[1.1 Feladatkiírás 3](#_Toc90331963)

[1.2 A program környezete 4](#_Toc90331964)

[1.3 A program készítése során felhasznált eszközök 4](#_Toc90331965)

[2. Megvalósítás 5](#_Toc90331966)

[1.4 Architektúra 5](#_Toc90331967)

[1.5 Rétegek 6](#_Toc90331968)

[1.5.1 Domén entitások rétege 6](#_Toc90331969)

[1.5.2 Adatelérési Réteg (Data Access Layer) 8](#_Toc90331970)

[1.5.3 Szolgáltatások rétege 8](#_Toc90331971)

[1.5.4 Web réteg 9](#_Toc90331972)

[3. Adatbázis felépítése 11](#_Toc90331973)

[4. Általános megoldások, megfontolások 12](#_Toc90331974)

[5. Felhasználói felület 15](#_Toc90331975)

[6. Továbbfejlesztési lehetőségek 17](#_Toc90331976)

[7. Irodalomjegyzék 18](#_Toc90331977)

# Bevezetés

Az Evosoft Hungary Kft.-nél 2020 nyarán kezdtem el dolgozni diákként szoftverfejlesztő pozícióban. Egy kisebb, ötszemélyes fejlesztő csapatba kerültem, akikhez az agilis szoftverfejlesztés módszertanának megfelelően egy Scrum Master és egy Product Owner is tartozott. A csapat egy TargetStudio nevű tooling szoftver fejlesztését végzi .NET keretrendszer környezetben. Az alkalmazás többek között különböző kiterjesztésű fájlok betöltését, fa struktúrában való megjelenítését és ezen fa struktúrák összehasonlítását végzi.

A munkám első három hónapja főként betanulással, illetve a fejlesztendő szoftverrel való ismerkedéssel telt. A betanulási szakasz végére már mélyebb ismeretekre tettem szert az alkalmazással kapcsolatban, és vállalni tudtam kisebb fejlesztői feladatok önálló elvégzését is. Ennek köszönhetően lehetőségem volt arra, hogy szakdolgozatomat a cégnél végzett fejlesztői munkám keretei között készítsem el.

## Feladatkiírás

A projekt során a célom egy-egy webes backend illetve frontend technológia megismerése, és ezek segítségével egy webes alkalmazás elkészítése volt. A kiválasztott technológiák az ASP.NET és az Angular lettek. Egy létező kisvállalkozás számára készítettem el a webshopot, a vállalkozás különböző hajgöndörítők, hajgumik és hasonló termékek értékesítésével foglalkozik. Eddig csupán közösségi médián, és emailes rendeléseken keresztül értékesítették a termékeiket, és igény volt egy saját weboldalra. A webalkalmazásom igazából egy REST API-ból és egy webes frontendből áll. A backendet ASP.NET-tel, a frontendet Angularral készítettem el. Az adatbáziskezelésre az EntityFramework Core-t a felhasználókezelésre az Identity plaftormot használtam.

A webshop funkciói:

* a vevők a különböző hajgöndörítő és egyéb termékeket megtekinthetik, és a kosárhoz adhatják, majd a kosár tartalmát később feladhatják rendelésre
* az adminisztrátor megtekintheti a leadott megrendeléseket

## A program környezete

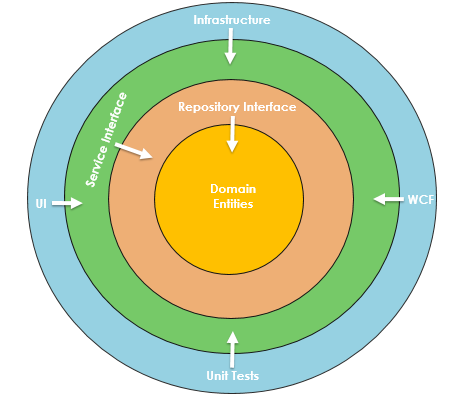
A webalkalmazás kliens oldala egy Angular, a szerver oldala ASP.NET Core alapú, monolitikus, hagyma architektúrát követő alkalmazás lesz. A .NET Core multiplatformitásának köszönhetően könnyen konténerizálható, és könnyen telepíthető különböző környezetekbe (például felhőbe). Az alkalmazásunk relációs, MS SQL adatbázissal fog rendelkezni. A program a felhasználó által történő használatához csak egy böngészőt igényel, mert a szerver alkalmazás nem a felhasználó gépén fut.

## A program készítése során felhasznált eszközök

* Visual Studio [1] és Visual Studio Code [2]: fejlesztőkörnyezet
* MSSQL Server Express [3]: adatbázis-kezelő rendszer
* Entity Framework Core [4]: az adatok objektum-relációs leképezésére
* Angular Material [5]: grafikus felhasználói felület elemei
* Microsoft Word: a dokumentáció elkészítése
* Microsoft SQL Server Management Studio [6]: adatbázis struktúra diagram előállítása
* Git [7], GitHub [8]: verziókezelésre

# Megvalósítás

## Architektúra



1. ábra: A hagyma architekúra rétegei

A fejlesztés során az ASP.NET Core projektek esetében gyakran alkalmazott hagyma architektúrára építettem, amellyel lazán csatolt megoldások készíthetők.

Szoros csatolásnak azt nevezzük, amikor egy osztály egy másik osztály konkrét implementációjától függ. Ezzel ellentétben a laza csatolás fennállása esetén egy osztály anélkül használhat egy másik osztályt, hogy függene tőle. Ennek elérésével csökkenthető a komponensek közötti függőség és ezzel együtt az átterjedő módosítások kockázata.

A hagyma architektúra előnye emellett a jobb karbantarthatóság, mivel az összes függőség csak egyetlen irányba, kívülről-befelé mutathat. Ezért adott réteg módosítása esetén biztosak lehetünk benne, hogy a tőle belsőbb rétegek nem szorulnak korrigálásra. Ha valahol máshol is változást vált ki a módosításunk, az csak külsőbb rétegekben lehet.

Az architektúra nagy mértékben épít a függőségek megfordításának alapelvére, a rétegek közötti kommunikáció interfészekkel történik. Ez magával hordozza azt is, hogy a konkrét implementációk futásidőben vannak biztosítva az alkalmazás számára. Ennek lehetővé tételét az Autofac kontrol megfordítási keretrendszer segítségével oldottam meg. Az architektúra az ábrán látható módon négy rétegből áll, amiket a saját implementációnkba is átvezettünk:

1. Domén entitások rétege: az alkalmazás központi része, a domén entitásokat tartalmazza.
2. Adatelérési réteg: absztrakciós réteg, amely egységes adatelérést biztosít az üzleti logikai réteg számára. Ez a réteg foglalkozik az adatforrás lekérdezésével, a kapott adatok objektumokká történő leképezésével, illetve a módosítások adatforráson történő érvényesítésével.
3. Szolgáltatások rétege: üzleti logikai rétegnek is hívható, az alkalmazás funkcióinak implementációját tartalmazza. A web réteg által jelzett műveleteket végzi el, eközben, ha szükség van nem elérhető adatokra akkor az adatelérési rétegtől kérdezi le azokat. Az adatok módosítása után az adatelérési réteget bízza meg a változtatások tárolásával is.
4. Web réteg: az alkalmazást teszi elérhetővé, hogy kliensek segítségével a felhasználók élni tudjanak azokkal a funkciókkal, szolgáltatásokkal, amiket a szoftver nyújt. A felhasználók által kezdeményezett műveleteket továbbítja az üzleti logikai réteg megfelelő részére.

## Rétegek

### Domén entitások rétege

Ebben az alfejezetben az adatmodell és azon entitásai kerülnek ismertetésre, ami alapján az adatbázis struktúrája is kialakult. Nem soroljuk fel az összes tulajdonságot, amikkel ezek az entitások rendelkeznek, csak azokat, amelyek az alkalmazás szempontjából fontosabb szerepet játszanak.

**Users**:

A Users tábla az alkalmazásbeli entitások közül az ApplicationUser osztály leképezésének felel meg. Az említett osztályt a felhasználók modellezésére hoztuk létre és az Identity által definiált IdentityUser osztályból származik le. Ez az ős egyrészt tartalmazza azokat a tulajdonságokat, amelyekre az authentikáció során szükség lehet, szóval azok megtervezésének terhét levette a vállunkról. Másrészt az Identity API használatához erre az osztályra vagy leszármazottjára van szükség és mivel igénybe vettük az API segítségét, ezért emiatt is adott volt az ős használata.

**AspNetRoles**:

Az AspNetRoles tábla az alkalmazásbeli entitások közül az ApplicationRole osztály leképezésének felel meg. Az említett osztályt a felhasználói szerepek modellezésére hoztuk létre és az Identity által definiált IdentityRole osztályból származik le. Ez az ős egyrészt tartalmazza azokat a tulajdonságokat, amelyekre az authorizáció során szükség lehet, másrészt az Identity API használatához erre az osztályra vagy leszármazottjára van szükség és mivel igénybe vettük az API segítségét, ezért emiatt is adott volt az ős használata. Az örökölt tulajdonságok mellett nem definiáltunk újakat. Az alkalmazásban csupán egyféle felhasználói szerep van definiálva: “Admin”, azaz az adminisztrátor, aki megtekintheti a leadott rendeléseket.

**AspNetUserRoles**:

Kapcsoló tábla a felhasználók és a felhasznlói szerepek között.

**Products:**

A Product entitásnak felel meg, egy terméket reprezentál.

**ProductCategories:**

A ProductCategory entitásnak felel meg, egy termék kategóriát reprezentál.

**Orders:**

Az Order entitásnak felel meg, egy rendelést reprezentál.

**OrderItems:**

Az OrderItem entitásnak felel meg. Egy kapcsolatot reprezentál egy termék és egy rendelés között. Segítségével modellezhetők a különleges árkedvezmények egy adott termékre, illetve a termék mennyiségének megadása a rendelésnél.

### Adatelérési Réteg (Data Access Layer)

Az adatelérési réteg feladata az adathozzáférés biztosítása a felsőbb rétegek számára. Ez a réteg tartalmazza az ApplicationDbContext osztályt, amely az IdentityDbContext osztályból származik. Ez reprezentálja az adatbáziskapcsolatot, és arra használjuk, hogy lekérdezzünk, illetve mentsünk adatokat az adatbázisba. A DbContext osztály a “Unit of work” és a “Repository” minták kombinációja. Itt definiáltam az adatbázisomban lévő táblákat, és az entitások konfigurációit.

Az adatelérési réteg tartalmazza továbbá az alkalmazás Repository osztályait. Ezek az osztályok az ApplicationDbContext osztály példányának segítségével kommunikálnak az adatbázissal.

Az adatelérési rétegben végeztük el az adatbázis seed-elését, azaz alapadatokkal való feltöltését is. Ez a fejlesztési és tesztelési folyamatokat könnyítette meg. Ezt a ModelBuilderExtensions osztályban valósítottam meg.

### Szolgáltatások rétege

Az üzleti logika megvalósítását tartalmazza, itt találhatók az alkalmazás lényegi funkcionalitását nyújtó osztályok. Ebben a rétegben vannak definiálva a DTO-k, az ezekhez tartozó validátorok és a ViewModel-lek is.

A DTO-k, ViewModel-lek és a domain entitások között szükség van valamilyen leképezésre. Amikor például egy lekérdező kérést szolgálunk ki, akkor a megfelelő domain entitásokból kell ViewModel-leket készíteni és ezeket visszaküldeni a válaszban. Amikor pedig egy domain entitást módosító kérést kell kiszolgálunk, akkor a DTO-ból kell a módosításnak megfelelő domain entitást előállítani. Ezeket az átalakításokat időpazarlás minden alkalommal manuálisan elvégezni, hiszen nagyon repetitív feladatról van szó. Ezen feladat elvégzésére az AutoMapper könyvtár szolgáltatását vettem igénybe, mely előre definiált mappelési szabályok alapján elvégzi helyettem az átalakítást.

Az üzleti logikát megvalósító kódot Service osztályokba szerveztem, a webes Controllerek a megfelelő Service interfészen keresztül történő meghívásával tudják igénybe venni az általa nyújtott szolgáltatásokat. A szolgáltatások működésük során lekérdeznek, létrehoznak, módosítanak, illetve törölnek domain entitásokat az üzleti logikai szabályoknak megfelelően. Ezen műveletek elvégzéséhez az adatelérési réteg Repository osztályait használják, amelyek az adatbáziselérést absztraktálják.

### Web réteg

A web rétegben található Controller osztályok felelősek a klienstől érkező HTTP kérések fogadásáért, feldolgozásáért, majd a megfelelő válasz visszaküldéséért.

[HttpGet("{productId}")]

public Task<ProductViewModel> GetProductById([FromRoute] int productId)

{

return \_productService.FindProductByIdAsync(productId);

}

A fenti példakód a ProductsController osztály egyik függvénye. A függvény az alkalmazott annotációk miatt akkor hívódik meg, amikor egy HTTP GET kérés érkezik az api/products/{productId} végpontra. A web réteg csak a kommunikáció lebonyolításáért felel, így a mintán látható módon üzleti logikai elemeket nem tartalmaz. Feladata egyszerűen az, hogy a felhasználói kéréseket lefordítsa az üzleti logikai réteg megfelelő függvényének meghívására.

Az alkalmazásban található többi Controller függvényei az itt bemutatott példával teljesen analóg módon működnek, ezért azok redundáns ismertetése helyett csak egy felsorolást teszek róluk:

**AuthController**: az api/auth útvonalon hallgatózik.

**ProductsController:** az api/products útvonalon hallgatózik.

**ProductCategoriesController:** az api/productscategories útvonalon hallgatózik.

**OrdersController:** az api/orders útvonalon hallgatózik.

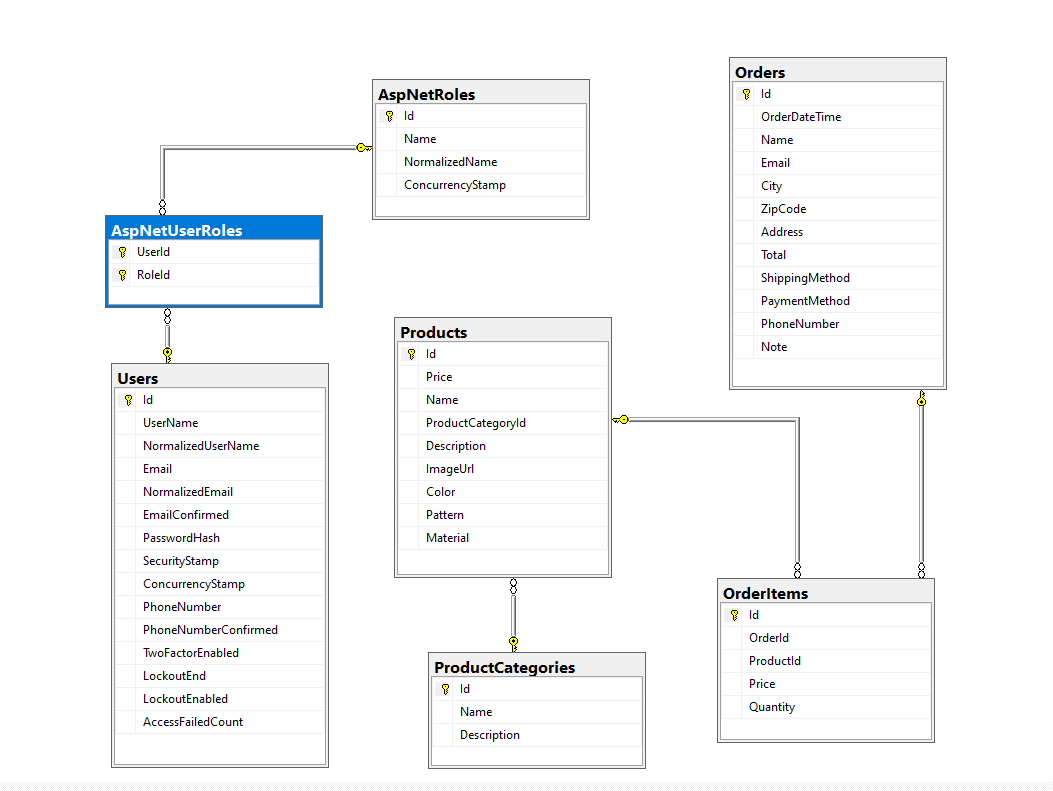
**OrderItemsController:** az api/orderitems útvonalon hallgatózik.



2. ábra: API leíró Swagger-rel

A Controller osztályokból automatikusan OpenAPI leírót is genreáltam, amely amellett, hogy dokumentálja az API-t arra is használható, hogy egyszerűen kipróbálhassam az implementált végpontokat.

# Adatbázis felépítése



3. ábra: Adatbázis struktúrája

Az adatok tárolására MSSQL adatbázist használtam, melyben a fenti ábrán látható struktúra alakult ki. A Code-First megközelítést alkalmazva a domén entitások és a kontextus osztály implementálása, konfigurálása után a séma elkészítését az Entity Framework technológiára bíztuk.

# Általános megoldások, megfontolások

Ebben az alfejezetben azokat a megoldásokat, megfontolásokat foglalom össze, amik az alkalmazás túlnyomó részében, szinte minden funkcionalitással kapcsolatban megjelennek. Ezért a funkciókat részletező fejezetekben az itt bemutatott dolgokra már csak legfeljebb említés szintjén fogok kitérni. A jobb érthetőség érdekében kívülről befelé haladva, egy elképzelt felhasználói kérést követve.

Az alkalmazási logika API-n keresztül elérhető, aminek egyes végpontjaiért kontrollerek felelnek. A kliens és API között adatátviteli objektumok segítségével zajlik a kommunikáció. Elnevezés tekintetében megkülönböztettük a klienstől érkező, illetve a kliensnek küldött objektumokat, az előbbieket DTOknak, az utóbbiakat pedig ViewModel-eknek nevezve.

Az alkalmazásban elérhető néhány művelet authentikációhoz van kötve. A felhasználók azonosítását JSON web tokenekkel (JWT) végeztük el. Érvényes felhasználónév és jelszó páros megadásával a felhasználó hozzáférési tokent kérhet az alkalmazástól, amit a további kérések Authorization fejlécében kell feltüntetni. A fejlécben található érték automatikusan ellenőrzésre kerül, sikeres authentikáció után a kérés tovább haladhat a feldolgozási láncon.

Mielőtt a kontroller megkezdhetné a beérkezett kérés feldolgozását, ellenőrzésre kerülnek a klienstől kapott adatok egy felkonfigurált validációs osztály alapján. Hibát, hiányosságot találva a kontroller érintése nélkül validációs hibák leírását tartalmazó választ küld vissza a rendszer.

A validációs osztályokat az alkalmazás Startup osztályában a kontrollerek és a számukra szükséges szolgáltatások IoC konténerbe történő regisztrálásánál adtuk meg. A későbbi bővítés rendkívül egyszerűen megtehető, ugyanis a validátorokat nem egyesével, hanem az összeset szerelvényből kiolvasva állítottam be. Így újabb validátor hozzáadásakor csak arra kell figyelni, hogy a megfelelő projektben (BLL) hozzuk azt létre.

Az adatok sikeres validálása után megkezdődik a kérés tényleges feldolgozása. Az egész alkalmazásban igyekeztem a kontrollereket „vékonyan” tartani, ezért a kontrollerek függvényei szinte üresek, mindössze a felhasználói kéréseket és a kérésekre adott válaszokat továbbítják a felhasználó és az üzleti logikai réteg között. Ezzel a kód újrafelhasználhatóságát állt szándékomban javítani.

Az architektúra leírásánál már említettem, hogy a rétegek interfészeken keresztül kommunikálnak egymással a laza csatolás elérése érdekében. A kontrollerek tehát a BLL projektben található interfészekre támaszkodnak a szolgáltatások elérésénél. Konkrét implementációkat futásidőben, függőség befecskendezésen keresztül kapnak. Az implementációs folyamat leegyszerűsítése és a lefelejtett regisztrációkból következő hibák elhárítása miatt nem az ASP.NET Core által biztosított IoC konténert használtuk.

Helyette ezzel a feladattal az Autofac konténerét bíztuk meg. Az Autofac segítségével modulokat lehet létrehozni, amelyekben a típusok regisztrálása szerelvények alapján is történhet. A végeredmény jól érthető és tisztán tartja a Startup osztályt. A szolgáltatások elvégzik a végpontra érkezett kérésnek megfelelő műveletet. Ennek során adatokkal dolgoznak, amelyeket az adatelérési rétegen keresztül érnek el és módosítanak. Egy-egy kérés feldolgozásánál fontos észben tartani, hogy a szolgáltatás több lépésben végzi el az adatmódosító műveleteket, melyek közben hibába ütközhet. Ilyen helyzetben szigorúan kerülendő az, hogy az adatbázis ennek következtében inkonzisztens állapotba kerülhessen. A szolgáltatások által végzett műveleteknek ezért tranzakciós határok között kell helyet foglalniuk.

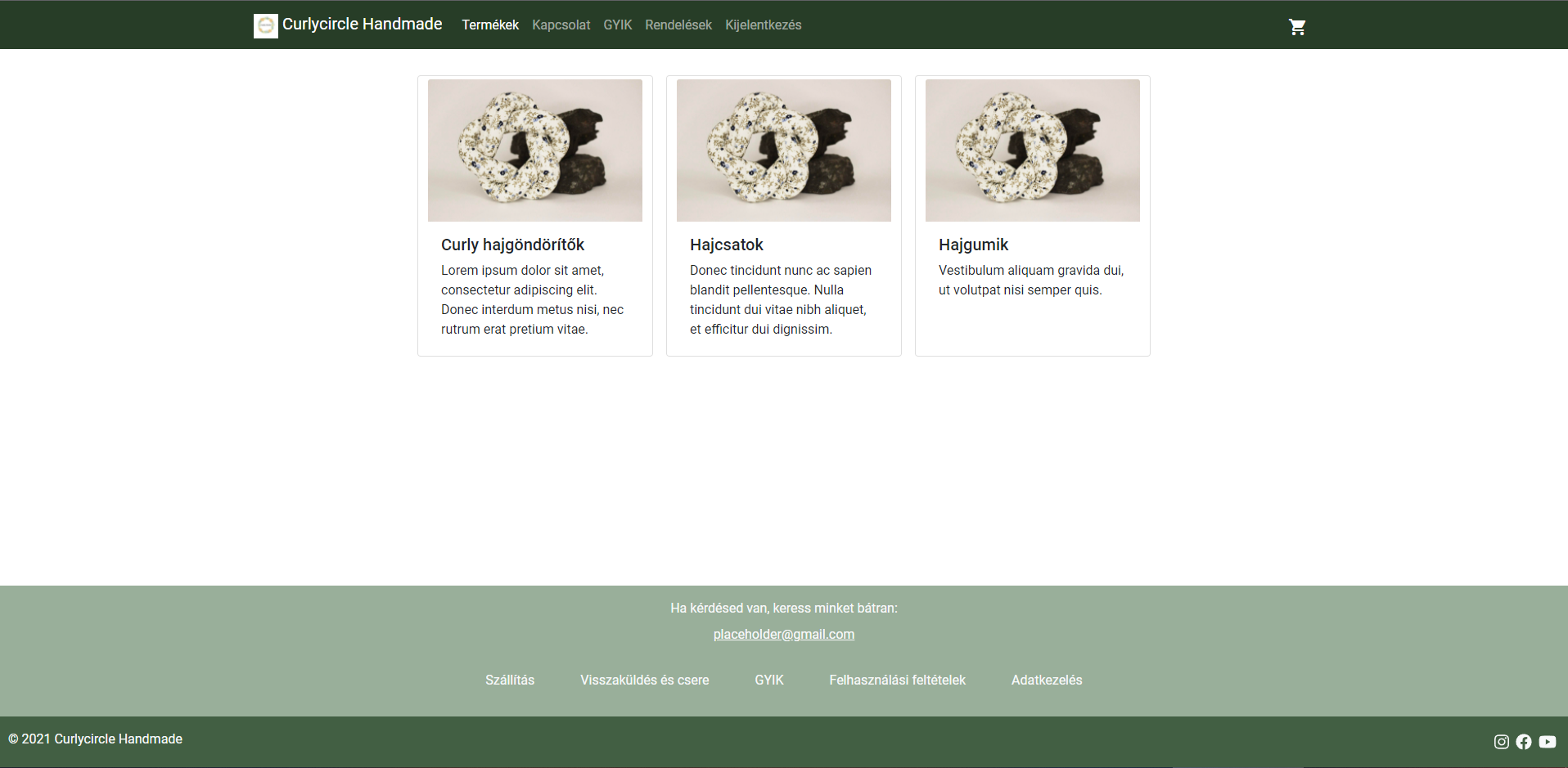
Az Entity Framework Core nagy segítséget nyújtott a tranzakciós határok kialakításában. Az adatbáziskontextus az elvégzett módosításokat nyomon követi, de az adatbázisba csak a SaveChanges() függvény explicit meghívásával vezeti át őket. Ennek köszönhetően a szolgáltatások törzsében a repository-k megfelelő függvényeinek használatával végrehajthattuk a módosításokat, amiket a sikeres lefutás végén már csak egy SaveChanges() hívással kell lezárni és véglegesíteni. Az alkalmazási és adatelérési rétegek felelősségeinek elkülönítése viszont azzal jár, hogy az alkalmazási réteg nem rendelkezhet rálátással az adatbáziskontextusra. Neki az éppen használt adattárolási technológiától függetlennek kell lennie és az EF Core a benne található adatbáziskontextussal együtt csak relációs adatbázisok esetén használható.

Ennek orvoslására az adatelérési rétegnek elegendő az IUnitOfWork interfészt megvalósítania, amelyet az alkalmazási réteg is gond nélkül használhat, hiszen így már a konkrét implementáció az adatbáziskontextussal együtt el van fedve előle. Az adatelérési réteg maradékának megvalósítását az IUnitOfWork interfészhez hasonlóan könnyű volt elkészíteni az EF Core-ban felkínált lehetőségekkel.

Az alkalmazás működése során folyamatos naplózást végzek egy konzolos felületre, amin a meghívott végpontok, függvények, ezek futásideje stb. mind meg van jelenítve. Egy beállítás hozzáadásával tudnánk emellett fájlba és/vagy adatbázisba is naplózni, de nekünk elegendő segítséget nyújtott a konzolos felület. Eddig nem ejtettem szót a hibakezelésről és a hibáknak megfelelő üzenetek küldéséről, amik a felhasználó tájékoztatását szolgálják. A kéréseket feldolgozó csővezetékbe regisztráltam az ErrorHandlerMiddleware osztályunkat globális hibakezelőként, aminek az Invoke(…) függvényében végezzük el a szükséges intézkedéseket. Ez egyszerűen egy try-catch blokkal körülvett áthívás a csővezeték következő elemébe, így, ha a csővezeték bármely ezt követő részén kivétel keletkezik, az legkésőbb ebben a függvényben el lesz kapva. Több kivételt definiáltam, amelyekkel az implementáció során dolgoztam: EntityNotFoundException, DomainException, ValidationAppException stb. Ezekre építettem a hibakezelést és a válaszhoz tartozó státuszkód megfelelő beállítását. Fejlesztői környezetben munkám könnyebb ellenőrzése végett a válaszba a kivételhez tartozó stacktrace-t is beleraktam.

# Felhasználói felület

A grafikus felületet az Angular keretrendszert felhasználva készíttem el. Emellett használtam a Bootstrap és az Angular Material keretrendszereket is.



4. ábra: Termékkategóriák képernyő

A képen szöveg látható

Automatikusan generált leírás

5. ábra: Rendelés képernyő

A képen szöveg látható

Automatikusan generált leírás

6. ábra: Rendelések képernyő

# Továbbfejlesztési lehetőségek

A webalkalmazásban rengeteg továbbfejlesztési lehetőség rejlik. Látható volt a képernyőképek alapján, hogy a felhasználói felület még elég kezdetleges és egyszerű, ahhoz, hogy a felhasználói élmény jobb legyen, még rengeteg részén lehetne fejleszteni, de ezek főleg designer szempontok, így eddig nem élveztek prioritást a fejlesztés során. Emellett a felhasználó kezelést is lehetne tovább fejleszteni. Lehetővé kéne tenni a vevők számára a regisztrációt és bejelentkezést, amely megkönnyítené a rendelés menetét. Emellett az admin számára ki lehetne alakítani külön felületet, ahol hozzáadhat új termékeket, illetve szerkesztheti a meglévőket.

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | „Microsoft Visual Studio,” [Online]. Available: https://visualstudio.microsoft.com/. [Hozzáférés dátuma: 11 12 2021]. |
| [2] | „Microsoft Visual Studio Code,” [Online]. Available: https://code.visualstudio.com/. [Hozzáférés dátuma: 11 12 2021]. |
| [3] | „MSSQL Server Express,” [Online]. Available: https://www.microsoft.com/en-us/sql-server/sql-server-downloads. [Hozzáférés dátuma: 12 12 2021]. |
| [4] | „Entity Framework Core,” [Online]. Available: https://docs.microsoft.com/en-us/ef/core/. [Hozzáférés dátuma: 11 12 2021]. |
| [5] | „Angular Material,” [Online]. Available: https://material.angular.io/. [Hozzáférés dátuma: 11 12 2021]. |
| [6] | „Download SQL Server Management Studio,” [Online]. Available: https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15. [Hozzáférés dátuma: 11 12 2021]. |
| [7] | „Git,” [Online]. Available: https://git-scm.com/. [Hozzáférés dátuma: 11 12 2021]. |
| [8] | „GitHub,” [Online]. Available: https://github.com/. [Hozzáférés dátuma: 11 12 2021]. |