

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Schütz Máté

**Webshop fejlesztése webes és mobil kliensre**

Konzulens

Albert István

BUDAPEST, 2022

Tartalomjegyzék

[1 Bevezetés 7](#_Toc121906034)

[1.1 Témaválasztás 7](#_Toc121906035)

[1.2 A dolgozat szerkezete 8](#_Toc121906036)

[2 Hasonló megoldások, technológiák bemutatása 9](#_Toc121906037)

[2.1 Hasonló megoldások 9](#_Toc121906038)

[2.2 Felhasznált technológiák 9](#_Toc121906039)

[2.2.1 Verziókezelő eszközök 9](#_Toc121906040)

[2.2.2 Backend alkalmazás technológiák 10](#_Toc121906041)

[2.2.3 Webes alkalmazás technológiák 11](#_Toc121906042)

[2.2.4 Mobil alkalmazás technológiák 12](#_Toc121906043)

[3 Tervezés 14](#_Toc121906044)

[3.1 Követelmények, feladatleírás pontosítása 14](#_Toc121906045)

[3.2 A rendszer architektúrája 15](#_Toc121906046)

[4 Backend alkalmazás 16](#_Toc121906047)

[4.1 Az alkalmazás architektúrája 16](#_Toc121906048)

[4.2 Adatbázis felépítése 19](#_Toc121906049)

[4.3 Domain réteg 20](#_Toc121906050)

[4.3.1 Entitás osztályok 20](#_Toc121906051)

[4.3.2 Repository interfészek 22](#_Toc121906052)

[4.3.3 Kivételek 23](#_Toc121906053)

[4.3.4 Kisegítő osztályok 23](#_Toc121906054)

[4.4 Adathozzáférési réteg 25](#_Toc121906055)

[4.4.1 Adatbázis modell, konfiguráció 25](#_Toc121906056)

[4.4.2 Repository osztályok 27](#_Toc121906057)

[4.4.3 Adatbázis feltöltése alapadatokkal 28](#_Toc121906058)

[4.5 Szolgáltatások rétege 29](#_Toc121906059)

[4.5.1 DTO és ViewModel osztályok 29](#_Toc121906060)

[4.5.2 Kivételek 31](#_Toc121906061)

[4.5.3 Service osztályok 31](#_Toc121906062)

[4.6 Web réteg (API) 36](#_Toc121906063)

[4.6.1 Controller osztályok 36](#_Toc121906064)

[4.6.2 Globális hibakezelő middleware 37](#_Toc121906065)

[4.6.3 A szerver alkalmazás konfigurálása 37](#_Toc121906066)

[4.7 Tesztelés 37](#_Toc121906067)

[4.8 Telepítés 39](#_Toc121906068)

[5 Webes kliens alkalmazás 40](#_Toc121906069)

[5.1 Az alkalmazás architektúrája 40](#_Toc121906070)

[5.2 Központi részek 40](#_Toc121906071)

[5.2.1 HTTP kliens 40](#_Toc121906072)

[5.2.2 Token interceptor 41](#_Toc121906073)

[5.2.3 Guard osztályok 42](#_Toc121906074)

[5.2.4 Modell osztályok, interfészek 42](#_Toc121906075)

[5.2.5 Segéd osztályok, függvények 42](#_Toc121906076)

[5.2.6 Navigációs sáv 43](#_Toc121906077)

[5.3 Termékkategóriák, termékek 44](#_Toc121906078)

[5.4 Felhasználókezelés 46](#_Toc121906079)

[5.5 Kosár, rendelés 48](#_Toc121906080)

[5.6 Adminisztrátori felület 50](#_Toc121906081)

[5.7 Telepítés 52](#_Toc121906082)

[6 Mobil kliens alkalmazás 53](#_Toc121906083)

[6.1 Az alkalmazás architektúrája 53](#_Toc121906084)

[6.2 Központi elemek 57](#_Toc121906085)

[6.2.1 Függőségek injektálása 57](#_Toc121906086)

[6.2.2 Navigáció 60](#_Toc121906087)

[6.2.3 Hálózati kommunikáció, autentikáció 62](#_Toc121906088)

[6.2.4 Adatbázis 65](#_Toc121906089)

[6.2.5 UI komponensek, kisegítő osztályok 66](#_Toc121906090)

[6.3 Termékek, termékkategóriák 67](#_Toc121906091)

[6.4 Profil 71](#_Toc121906092)

[6.5 Kosár, rendelés 73](#_Toc121906093)

[7 Összefoglaló 76](#_Toc121906094)

[8 Irodalomjegyzék 77](#_Toc121906095)

[Függelék 80](#_Toc121906096)

Hallgatói nyilatkozat

Alulírott **Schütz Máté**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 12. 06.

...…………………………………………….

Schütz Máté

Összefoglaló

Manapság a kisvállalkozások számára elengedhetetlen, hogy valamilyen formában jelen legyenek az online térben, hiszen a fogyasztók jelentős része már internetes rendeléseken keresztül intézi el a különböző vásárlásait. Diplomamunkámnak egy webshop rendszer fejlesztését választottam, az egyetemen szerzett fejlesztői tudásom kamatoztatása és további tapasztalatszerzés céljából.

Ebben a dokumentumban bemutatom a diplomamunkám keretein belül lefejlesztett webshop rendszert, amely egy szerver oldali és két kliens oldali alkalmazásból áll. Ismertetem a fontosabb fejlesztői döntéseket, az érdekesebb megoldásokat, illetve az alkalmazások architektúráját és az implementáció részleteit. A diplomamunka keretein belül sikerült megterveznem és lefejlesztenem előbb említett alkalmazásokat, illetve a szerver oldali és a webes kliens alkalmazást telepítettem egy webes tárhely szolgáltató igénybevételével, így igazoltam, hogy a megoldás nem csak fejlesztői környezetben működik, hanem valós használat közben is.

A kitűzött feladat megoldása során rengeteg szakmai tapasztalatot szereztem, úgy érzem, hogy szoftverfejlesztőként hatalmasat fejlődtem. Biztos vagyok benne, hogy az itt szerzett tudást és tapasztalatokat később karrierem során kamatoztatni tudom.

Abstract

Nowadays, it is essential for small businesses to be present in the online space in some form, since a significant number of consumers already make their various purchases through internet orders. I chose the development of a webshop system as my diploma thesis, in order to make use of my development knowledge acquired at the university and to gain further experience.

In this document, I present the webshop system developed within the framework of my thesis, which consists of a server-side and two client-side applications. I will describe the most important developer decisions, the more interesting solutions, as well as the application architecture and implementation details. Within the framework of the thesis, I managed to design and develop the aforementioned applications, and installed the server-side and web client applications using a web hosting provider, thus proving that the solution works not only in a development environment, but also during real use.

I gained a lot of professional experience while solving the assigned task, and I feel that I have improved enormously as a software developer. I am sure, that I will be able to use the knowledge and experience gained here later in my career.

# Bevezetés

## Témaválasztás

A szoftverfejlesztés területén a webfejlesztés egyre nagyobb teret nyert az elmúlt évtizedek során. Ma a fejlesztők többségének a programozás egyet jelent a webfejlesztéssel [1]. Éppen ezért én is fontosnak tartottam, hogy jelentősebb tapasztalatot szerezzek a webfejlesztés terén, hogy később ezt karrierem során kamatoztathassam. Egyetemi tanulmányaim során néhány tárgy keretén belül már megismerkedtem a webfejlesztés különböző aspektusaival, azonban még jobban el akartam mélyíteni a tudásomat ezen a területen. Ebből az okból kifolyólag mindenképpen egy webfejlesztéshez kapcsolódó témát akartam választani a diplomamunkámnak.

Egy ismerősöm az elmúlt években elindított egy kisvállalkozást, amely kézzel készített hajgöndörítő, illetve hajápoló termékek eladásával foglalkozik. Eddig a vevőket különböző szociális média platformokon keresztül érte el – Instagram és Facebook. A vevők itt tudták leadni a rendeléseiket személyes üzeneteken keresztül. A vállalkozás sikerességéből adódóan felmerült az igény egy saját webshop készítésére, amely megkönnyítené a vevők számára a rendelések leadását, javítaná a vásárlási élményt, illetve levenné a vállalkozó válláról a rendelések manuális kezelésének terhét. Ezt a webshop alkalmazás meg is valósítottam kifejezetten webshopok készítésére alkalmas szoftverek segítségével. Ezek előnye, hogy amelyek segítségével bármilyen fejlesztői munka nélkül létre tudtam hozni az alkalmazást. Ezután támadt az az ötletem, hogy le szeretnék fejleszteni egy ehhez hasonló webshopot, mivel egyrészt így szakmai tapasztalatot tudok szerezni a webfejlesztés területén és el tudom mélyíteni ezt a tudást, másrészt megtapasztalhatom az előbb említett szoftverek használatának előnyeit, illetve hátrányait a „kézi” webfejlesztéssel szemben. Egy további előnye annak, hogy egy létező webshophoz hasonló alkalmazást fejlesztek le az, hogy valós igényeknek megfelelően kell létrehoznom az alkalmazásomat, így egy olyan megoldást implementáltam, amely közel áll a valós, használatban lévő webshopokhoz.

Egy webshop alapvetően két alkalmazásból áll. Az egyik a szerver oldali alkalmazás, amely a webshop üzleti logikáját valósítja meg, és tartalmazza a szükséges adatokat. A másik maga a kliensoldali webalkalmazás, amely a felhasználó böngészőjében fut, és egy megfelelő felületet biztosít a vevő számára. Emellett én úgy döntöttem, hogy megvalósítok egy második kliens oldali alkalmazást is, amely egy Android platformra készített mobil alkalmazás. Ennek oka, hogy már évek óta érdekel a mobil szoftverfejlesztés és a vállalkozás számára ez is egy hasznos bővülési lehetőség, amellyel még több vevőt tud elérni.

## A dolgozat szerkezete

Ebben az alfejezetben ismertetem a dolgozatom további fejezeteit, szerkezetét. A dolgozatom hét további fejezetből áll.

A második fejezetben írok a hasonló, már létező megoldásokról, illetve arról, hogy ezekből miket merítettem, ültettem át a saját megoldásomba és miket csináltam másképpen. Emellett itt bemutatom a fejlesztés során használt technológiákat is.

A harmadik fejezetben ismertetem a követelményeket a rendszerrel szemben, pontosítom a feladatleírást, illetve bemutatom a három alkalmazásból álló rendszer architektúráját.

A negyedik, ötödik és hatodik fejezet a szerver oldali alkalmazásról, a webes kliens alkalmazásról és a mobil kliens alkalmazásról szól, ebben a sorrendben. Ezekben a fejezetekben egyesével bemutatom a három különböző alkalmazás részleteit és kitérek a fontosabb fejlesztői és tervezői döntésekre, amelyeket a fejlesztés során hoztam meg.

A hetedik fejezetben röviden összefoglalom a tapasztalataimat és élményeimet, amelyeket a diplomamunka készítése közben szereztem, emellett tömören kitérek a továbbfejlesztési lehetőségekre is.

A nyolcadik fejezet az irodalomjegyzéket tartalmazza, amelyekben a dolgozat elkészítéséhez források találhatóak.

# Hasonló megoldások, technológiák bemutatása

Ebben a fejezetben bemutatok különböző létező megoldásokat, amelyekből ihletet merítettem, illetve amelyeket átültettem a saját alkalmazásomba. Emellett ismertetem a fejlesztés során felhasznált technológiákat, amelyek az alkalmazásom alapját képezik.

## Hasonló megoldások

Az alkalmazásom megtervezésekor egyrészt a korábban létrehozott webshopra, másrészt a saját felhasználói tapasztalataimra támaszkodtam, emellett rengeteg különböző webshopot áttekintettem. Ezekből merítettem ihletet, illetve megtapasztaltam, hogy mik azok a tipikus hibák, amelyek a felhasználói élményt csökkentik.

Az egyik ilyen fő hiba az, hogy sok webshop különböző megfontolásokból és okokból adódóan nem enged regisztráció, illetve bejelentkezés nélkül rendelést leadni. Ez sokszor el tudja tántorítani a potenciális vásárlót, hiszen általában egy fiók létrehozása teljesen feleslegesnek tűnik a felhasználó számára, főleg, ha csak egy egyszeri vásárlásról van szó. Sokan azzal érvelnek, hogy azzal, hogy regisztrációra kényszerítjük a vevőt, nagyobb eséllyel konvertálhatjuk visszatérő vásárlóvá. Véleményem szerint azonban egy modern e-commerce oldal esetében sokkal fontosabb a felhasználói élmény emelése. Ha a felhasználó úgy érzi, hogy feleslegesen kell időt töltenie a regisztrációval, sokkal nagyobb az esélye, hogy egy másik webshophoz fog fordulni, vagy egyszerűen elmegy a kedve a vásárlástól.

Egy másik jellegzetes hiba, ami általában a kisebb, kevesebb erőforrással rendelkező webshopokra jellemző, hogy miután a felhasználó hozzáadta a megrendelni kívánt termékeket a kosarához, majd ezután regisztrált, illetve bejelentkezett, elveszíti a kosarának a tartalmát. Ez rendszerint nagy mértékben csökkenti a felhasználói élményt, és frusztrációhoz vezethet.

## Felhasznált technológiák

### Verziókezelő eszközök

Verziókezelésre és forráskód kezelésre Gitet használtam, amely manapság a legelterjedtebb eszköz erre a feladatra. [2] A Git egy ingyenes és nyílt forráskódú elosztott verzió kezelő rendszer, amely egyaránt használható effektíven kis és nagy méretű projekteknél. A forráskódom biztonságos tárolására a GitHub [3] honlapot használtam, amely a legelterjedtebb Git alapú forráskód tároló és kezelő rendszer.

### Backend alkalmazás technológiák

A backend alkalmazásomat a Visual Studio [4] fejlesztői eszköz segítségével készítettem el C# nyelven. A backend alkalmazást ASP.NET Core platformon fejlesztettem. [5] Az ASP.NET Core a Microsoft nyílt forráskódú keretrendszere, amely segítségével modern, internetkapcsolattal rendelkező alkalmazásokat lehet készíteni.

A backend alkalmazásom rendelkezik egy relációs adatbázissal, amely a webshop adatait tárolja. Az adatbázis létrehozásához, az adathozzáféréshez és az objektum-relációs letérképezéshez az Entity Framework Core [6] keretrendszert használtam. Ez a keretrendszer nagyban megkönnyíti az adatbázis hozzáférést .NET Core platformon, hiszen a segítségével .NET objektumokon keresztül kezelhetjük az adatbázist és nincs szükség szinte semmilyen adathozzáférési kód megírására. Az adatbázis megtekintésére és az adatbázis struktúra diagram előállításához a Microsoft SQL Server Management Studio [7] szoftvert használtam.

Az API leírásához a Swashbuckle.AspNetCore [8] NuGet csomagot használtam, amely egy Swagger leíró implementációt tartalmaz. Ez automatikusan generál egy OpenAPI specifikációnak [9] megfelelő API dokumentációt az ASP.NET Core projektből.

A backend alkalmazás Data Transfer Object (DTO) osztályainak validálásához a FluentValidation [10] NuGet csomagot használtam. Ez egy .NET könyvtár, amely segítségével a segítségével attribútumok megadása helyett ember által könnyen olvasható módon, külön validációs osztályokban írtam meg validációs kódot a DTO osztályok számára. Így sokkal tisztább maradt ezen osztályok kódja, nem szennyeztem be a sok validációs attribútummal.

Az ViewModel, DTO és domain entitások osztályainak leképezéséhez az AutoMapper [11] könyvtárat használtam. Ennek a segítségével elég volt egyszer megírnom a leképezési szabályokat, és ezután a könyvtár függvényeivel egyszerűen leképezhettem ezeket az objektumokat.

Az alkalmazásban a termékek és termékkategóriák képeinek feldolgozására, formázására az ImageSharp [12] könyvtárat használtam. Ez az egyik legnépszerűbb képfeldolgozó könyvtár a .NET platformon.

A webes fizetés megvalósításához a Barion [13] szolgáltatót választottam, mivel náluk ingyenesen kipróbálható a szolgáltatásuk teszt környezete, így egy egyszerű regisztráció után már kommunikálhatott is a backend alkalmazásom a Barion rendszerével. A teszt környezetükben előre megadott teszt kártyákat tudtam használni a fizetések lebonyolításához. Emellett egy másik előnyük, hogy nagyon átlátható és összeszedett az API fejlesztői dokumentációja. Az egyetlen másik opció az OTP SimplePay szolgáltatása lett volna, azonban náluk a teszt rendszer használatához egy regisztrációs díjat kellett volna fizetnem.

A Barion API meghívásához a RestSharp [14] könyvtárat használtam, mivel ez a legnépszerűbb REST API kliens könyvtár a .NET platformon és kiváló dokumentációval rendelkezik.

Az egységtesztek elkészítéséhez a Moq [15] és az XUnit [16] könyvtárakat használtam, mivel .NET platformon ezek a legnépszerűbbek, illetve ezeket ajánlja a Microsoft is.

Az alkalmazás telepítéséhez a Microsoft Azure [17] platformját választottam, mivel kiválóan integrálva van a Visual Studio szoftverrel, illetve nagyon népszerű, így rengeteg útmutatót és forrást találtam hozzá. Az Azure portálon regisztráció után egy ingyenes webszervert béreltem, amely természetesen alacsony teljesítményű, de az telepítési és konfigurálási folyamat kipróbálásához teljesen megfelelt.

### Webes alkalmazás technológiák

A webes kliens alkalmazás elkészítését a Visual Studio Code [18] fejlesztői eszköz segítségével végeztem el TypeScript [19] nyelven.

A webalkalmazásomat egy modern JavaScript keretrendszer segítségével szerettem volna elkészíteni, és végül az Angular [20] mellett döntöttem. Az Angular egy webes keretrendszer, fejlesztői platform, amelyet a Google fejleszt, és single-page alkalmazások (SPA) készíthetőek vele.

A felhasználói felület elkészítéséhez az Angular Material [21] könyvtárat használtam. Ez egy komponens könyvtár, amelyet a hivatalos Angular csapat fejleszt és megfelel a Material Design [22] specifikációnak. Segítségével könnyen tudtam integrálni modern és szép UI komponenseket az Angular alkalmazásomba.

A PrimeNG [23] könyvtárból további komponenseket használtam a felhasználói felület létrehozásához. Ez egy népszerű, ingyenesen használható nyílt forráskódú UI komponens könyvtár az Angular keretrendszerhez.

A backend alkalmazáshoz hasonlóan a webes kliens telepítéséhez szintén az Azure platformot választottam. Ezt az alkalmazást egy külön webszerverre telepítettem, amelyet megfelelően konfiguráltam, hogy ki tudja szolgálni az Angular alkalmazásomat a felhasználók számára.

### Mobil alkalmazás technológiák

Az Android alkalmazás fejlesztésénél különösen figyeltem arra, hogy a legmodernebb technológiákat használjam, és a Google által hivatalosan ajánlott megközelítéseket kövessem. Így egy átfogó képet kaptam az Android fejlesztés aktuális állapotáról, és beleláttam, hogy milyen irányba viszi tovább a Google a platformot.

Az alkalmazás fejlesztéséhez az Android Studio szoftvert használtam. [24] Ez a Google által is hivatalosan támogatott integrált fejlesztői környezet (IDE) Android alkalmazások fejlesztéséhez. Az alkalmazást Kotlin programozási nyelvben készítettem el, mivel ma már ez a Google által is előnyben részesített nyelv Android alkalmazások fejlesztéséhez.

Az alkalmazás szinte minden funkcióját az Android Jetpack könyvtárrendszer segítségével készítettem el. [25] A Jetpack olyan könyvtárak halmaza, amelyeket a Google fejleszt, és amelyek segítenek a fejlesztőknek követni a platform best-practice módszereit, csökkenteni a boilerplate kód mennyiségét, és olyan kód írását, amely konzisztensen működik többféle Android verzión és eszközön is. A következőkben ezeket a könyvtárakat fogom bemutatni.

A Google iránymutatását követve, az alkalmazás architektúrájának kialakításához, és a függőség injektáláshoz a Hilt könyvtárat használtam. Ez a könyvtár a Dagger függőség injektáló keretrendszerre épül, és egy szabványos módszert ad arra, hogy hogyan építsük be a Dagger függőség kezelést az Android alkalmazásunkba. Emellett leegyszerűsíti és megkönnyíti a Dagger használatát, mert elfedi annak a komplexitását. [26]

Az adatok perzisztálását a Room könyvtárral végeztem, amely egy SQLite alapú keretrendszer, [27] gyakorlatilag egy absztrakciós réteget képez az SQLite felett, leegyszerűsíti és megkönnyíti annak használatát, így biztosabb adatbázis kezelést érhetünk el kevesebb kód írásával.

A felhasználói felület elkészítéséhez az új Jetpack Compose könyvtárat használtam, amely lecseréli a régi View megközelítésű implementációt, ahol XML nyelven készült a felület. A Compose a Google által fejlesztett keretrendszer, amellyel natív Android felhasználói felületeket hozhatunk létre. [28] Előnye a View alapú megközelítéssel szemben, hogy sokkal kevesebb kódot kell írnunk hozzá, Kotlin nyelven írjuk le a felhasználói felületet és használata intuitívabb. A Compose első verziója 2021 júliusában lett kiadva, tehát valóban új keretrendszernek számít, azonban a Google az új alkalmazások felhasználói felületének implementálásához ezt ajánlja. A Compose implementálja a Material Design komponenseket, így az összes beépített elemnek modern, letisztult kinézete van. Emellett az alkalmazás témáját is nagyon könnyen testre lehet szabni vele, és támogatja a sötét és világos témák közötti váltást.

A backend alkalmazással való kommunikáció megvalósításához az OkHttp [29] és az erre épülő Retrofit [30] könyvtárat használtam. A hálózati kommunikáció megvalósítására ezek a legnépszerűbb könyvtárak Androidon, és már korábban is használtam őket, így nem volt számomra kérdés a választás. A Retrofit leegyszerűsíti az OkHttp használatát, úgy, hogy elég egy Kotlin interfészt definiálnom, ami tartalmazza a hálózati kommunikációhoz szükséges HTTP API-nak megfelelő függvényeket. A Retrofit ezután legenerálja ehhez az interfészhez a megfelelő implementációt és létrehozza a megfelelő OkHttp klienst. Nekem azonban több konfigurálási lehetőségre volt szükségem az autentikáció miatt, így én külön létrehoztam és bekonfiguráltam az OkHttp klienst, majd ezt megadtam a Retrofit osztályának.

# Tervezés

## Követelmények, feladatleírás pontosítása

A diplomamunka keretében megtervezendő és elkészítendő egy webshop rendszer egy Angular alapú webes és egy Android alapú mobil klienssel, amelyek egy REST API-n keresztül kommunikálnak a szerver oldali ASP.NET Core alkalmazással.

A webshoppal szemben támasztott általános követelmény, hogy a felhasználó képes legyen a tipikus online vásárlási műveleteket elvégezni, azaz megtekintheti a termékeket, válogathat közülük, hozzáadhatja a termékeket a kosarához, majd a megfelelő adatok megadásával leadhatja a rendelését. A webshopban különböző termékkategóriák, és ezeken belül termékek lehetnek, és egy termék mindig pontosan egy termékkategóriához kell, hogy tartozzon. Biztosítani kell, hogy a felhasználó akár bejelentkezés nélkül is használhassa a webshopot, ilyenkor a rendelésnél minden alkalommal kézzel kell megadnia a rendelési adatait. A felhasználó a megfelelő adatok megadásával regisztrálhat a webshopba, majd a fiókjába be is jelentkezhet. Bejelentkezésnél biztosítani kell, hogy ha voltak a felhasználó kosarában korábban termékek, ezek nem vesznek el, átkerülnek a felhasználó fiókjához tartozó kosárba. Rendeléskor a bejelentkezett felhasználó adatai automatikusan ki kell, hogy töltődjenek a korábban megadott regisztrációs adatok alapján. Továbbá a bejelentkezett felhasználó meg kell, hogy tudja tekinteni a korábban leadott rendeléseit a webshopban. További elvárás a webshoppal szemben, hogy akár be van jelentkezve a felhasználó, akár nem, amennyiben bezárja az alkalmazást és később újra megnyitja, akkor a kosarának tartalma nem szabad, hogy elvesszen.

A szerver oldali alkalmazással szemben támasztott követelmény, hogy biztosítsa az adatbázist és a megfelelő szolgáltatásokat a kliens oldali alkalmazások számára, és emellett az üzleti logikát implementáló részekhez készüljenek egységtesztek.

A webes kliensben a korábban említett általános webshop és felhasználókezelési funkciók mellett további elvárás, hogy a felhasználó képes legyen a termékeket szűrni az áruk, illetve különböző tulajdonságaik alapján, ezzel megkönnyítve a termékek közötti válogatást. Továbbá a webes kliensen a felhasználó számára biztosítani kell webes fizetési lehetőséget is, amelyet valamelyik online fizetési szolgáltató használatával kell implementálni. Emellett a webes kliensnek rendelkeznie kell egy adminisztrátori felülettel, amelyet csak az adminisztrátor jogú felhasználó érhet el, miután bejelentkezett a fiókjába. Ezen a felületen az adminisztrátor hozzáadhat, eltávolíthat és módosíthat termékkategóriákat és termékeket, illetve azok tulajdonságait (színük, anyaguk, mintájuk), emellett megtekintheti a webshopban leadott rendelések részleteit, így ki tudja szolgálni azokat.

A szerver oldali alkalmazást és a webes klienst telepíteni kell valamilyen webes tárhely szolgáltató igénybevételével, ezzel bizonyítva azt, hogy a rendszer nem csak fejlesztői környezetben működik. Természetesen ez magába foglalja az alkalmazások konfigurálását annak érdekében, hogy a telepítési és kiadási folyamatok megfelelően működjenek.

Az Android alapú mobil kliens alkalmazásnak meg kell valósítania a fejezet elején említett általános webshop és felhasználókezelési funkciókat, azonban itt nem követelmény az extra funkciók, adminisztrátori felület, webes fizetés, telepítés megvalósítása. Az Android alkalmazás fejlesztésénél a fő elvárás a technológiák megismerése és a megfelelő alkalmazás architektúra létrehozása.

## A rendszer architektúrája

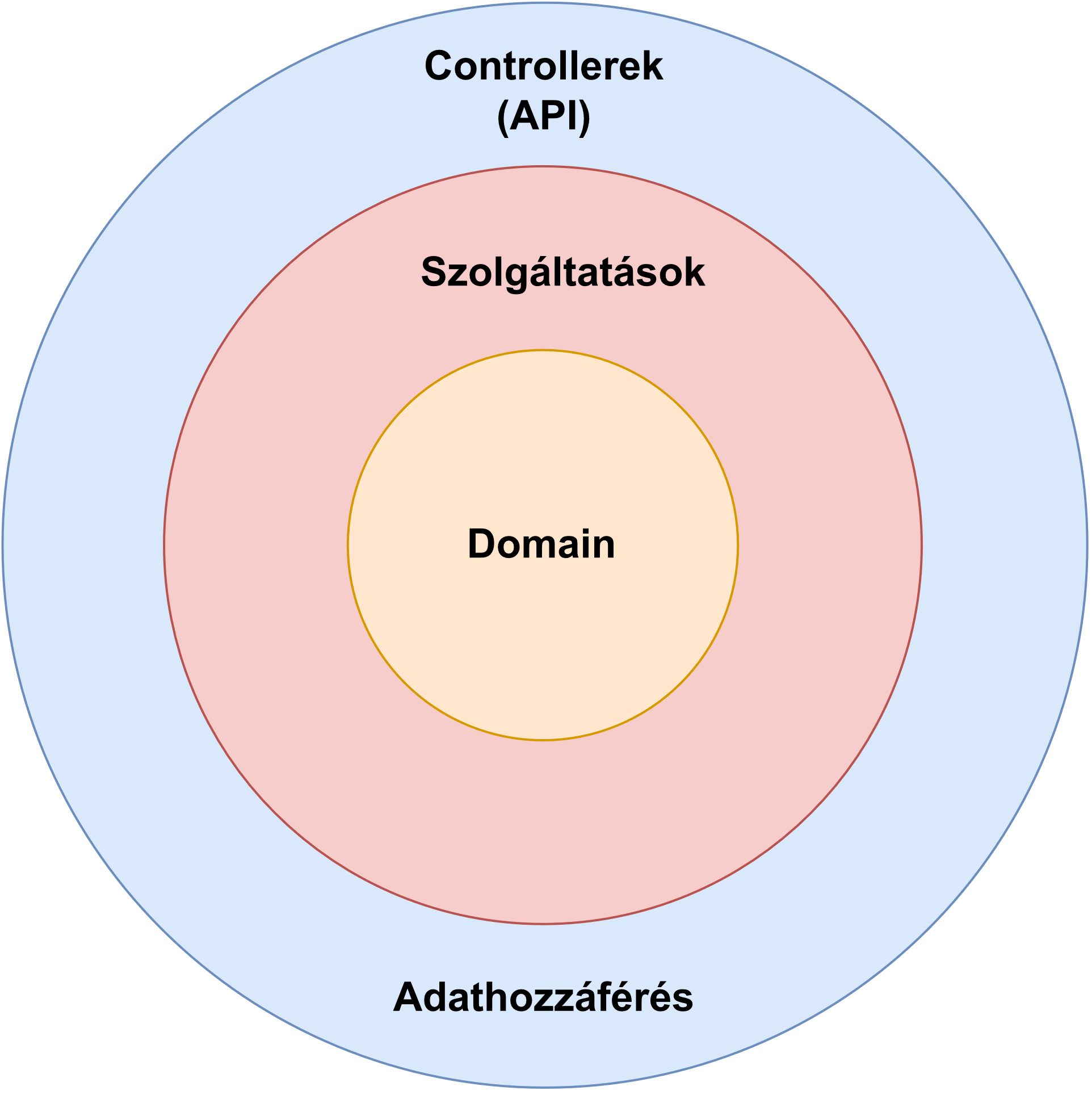
A webshop rendszer összesen három alkalmazásból áll: a szerver oldali alkalmazásból és a webes és mobilos kliensekből. A szerver oldali ASP.NET Core alkalmazás egy tipikus monolitikus, rétegezett alkalmazás, amely egy REST API-t publikál a külvilág felé. A kliensek ezen keresztül tudják meghívni az alkalmazás szolgáltatásait, ezen keresztül kommunikálhatnak vele. A szerver oldali alkalmazás rendelkezik egy relációs alapú adatbázissal, amelyben a webshop működéséhez szükséges adatokat tárolja. A kliens alkalmazások biztosítják a megfelelő felhasználói felületet a webshop vevői számára, és HTTP kéréseken keresztül hívják meg a szerver oldali alkalmazás szolgáltatásait.

# Backend alkalmazás

Ebben a fejezetben a backend alkalmazást fogom ismertetni. Kitérek architektúrájára, az adatbázis felépítésére, majd egyesével bemutatom az alkalmazás rétegjeit, és végül írok az alkalmazás teszteléséről és telepítéséről is.

## Az alkalmazás architektúrája

A backend alkalmazás fejlesztése közben igyekeztem minél jobban a Microsoft ajánlásait [31] és a legjobban bevált módszereket követni az ASP.NET Core Web API projektekkel kapcsolatban. Ennek részéként a Clean Architecture-t, vagy másik ismert nevén a hagyma architektúrát választottam. A hagyma architektúra egy rétegelt architektúra, ahol a rétegeket koncentrikus körökként vizualizálhatjuk. Ezt a vizualizációt el is készítettem, az 1. ábra tartalmazza.



1. ábra - A hagyma architektúra

A hagyma architektúra lényege, hogy a rétegek az alattuk lévő rétegekkel szigorúan csak interfészeken keresztül kommunikálnak. A függőségek iránya befele, a kör közepe felé mutat. Mivel a függőségek iránya megfordul, és a rétegek absztrakcióktól és nem implementációktól függenek, így magukat az implementációkat futás időben transzparensen kicserélhetjük. A rétegek fordítás időben interfészektől függenek, és magukat az implementációkat futásidőben kapják meg. Ennek a megközelítésnek az egyik legfontosabb előnye a jobb tesztelhetőség, hiszen az absztrakcióknak köszönhetően a függőségeket nagyon könnyen helyettesíthetjük teszt objektumokkal (*kimockolhatjuk* őket). Erről a 4.7-es fejezetben részletesebben is írok még.

A hagyma architektúra mögött a fő ötlet a függőségek befele mutató iránya. Minél mélyebben van egy réteg, annál kevesebb függősége van, így például a Domain rétegnek semmilyen függősége nincs más rétegektől, el van szigetelve a külvilágtól. Ez azért fontos, mert a függőségek iránya meghatározza, hogy egy adott réteg mit csinálhat a hagyma architektúrában. Mivel csak az alatta lévő rétegektől függ, így csak olyan metódusokat hívhat, amelyeket lejjebbi rétegek publikálnak.

Az belsőbb rétegeket használtam arra, hogy interfészeket, azaz szerződéseket definiáljak. A kintebb lévő rétegek implementálják ezeket az interfészeket. Így be tudtam csomagolni az egész üzleti logikát a Domain és a Szolgáltatások rétegébe anélkül, hogy bármilyen implementációs részletét ismernem kéne az infrastruktúrának. Azaz ez azt jelenti, hogy a Szolgáltatások rétege csak a Domain réteg által definiált interfészektől függ. Ennek köszönhetően az üzleti logika teljesen független az adathozzáférés módjától, illetve az adatbázis részleteitől, azaz egyszerűen ki lehetne cserélni akár az adattárolás, vagy az adathozzáférés módját anélkül, hogy az üzleti logikában bármit is módosítanom kéne.

Tehát az architektúra négy rétegből áll, ez az én implementációmban a következőképp néz ki:

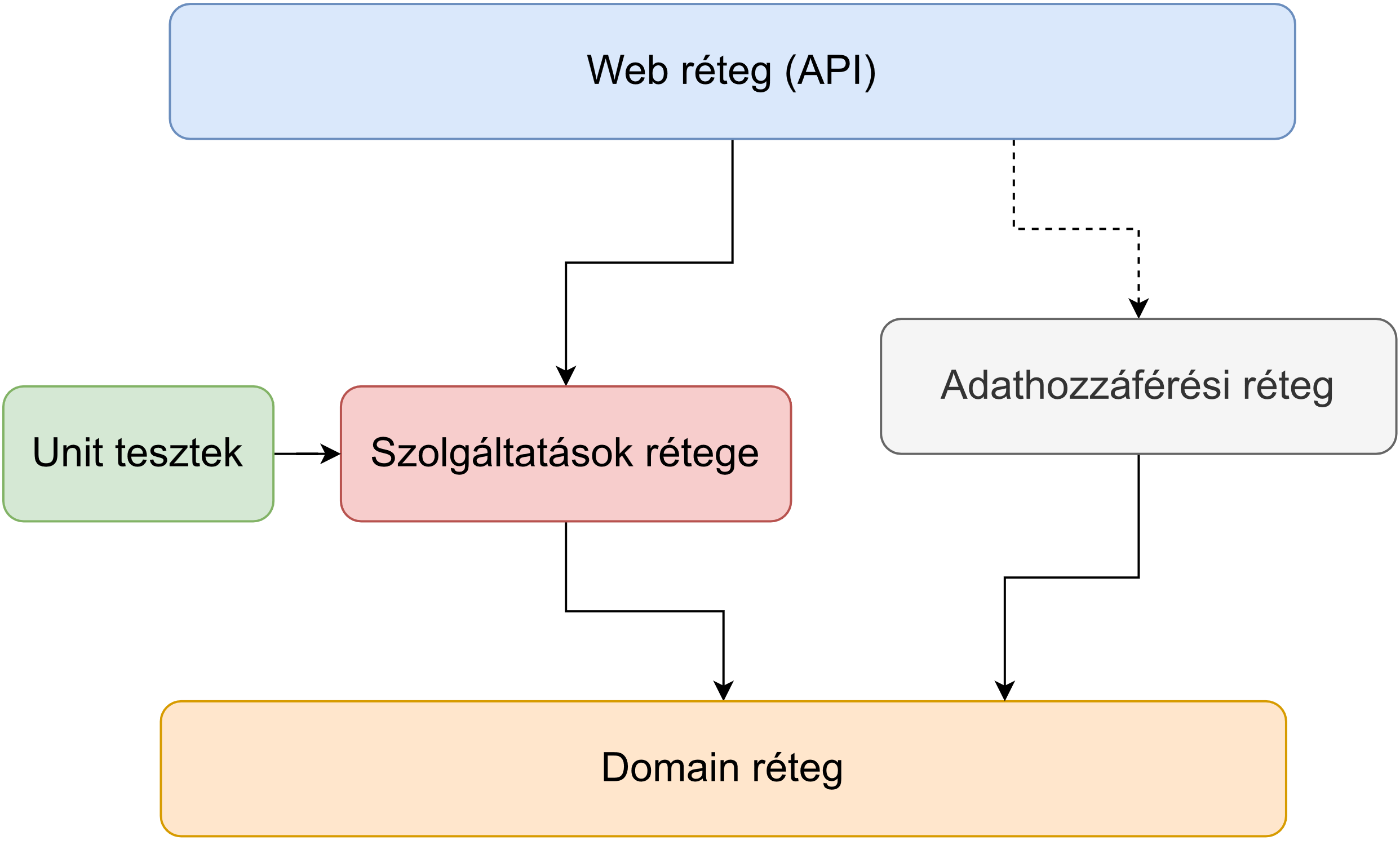
1. Domain réteg: az alkalmazás központi része, az alkalmazás entitás osztályait, a repository interfészeket, illetve a kivételeket tartalmazza.

2. Szolgáltatások rétege: az alkalmazás üzleti logikáját tartalmazza, az entitás osztályokkal dolgozik, így függ a Domain rétegtől.

3. Adathozzáférési réteg: absztrakciós réteg, amely egységes adatelérést biztosít az üzleti logikai réteg számára. Implementálja a Domain réteg által definiált repository interfészeket, így ő is függ a Domain rétegtől. Ez a réteg foglalkozik az adatforrás lekérdezésével, a kapott adatok objektumokká történő leképezésével, illetve a módosítások adatforráson keresztül történő érvényesítésével.

4. Web réteg: az alkalmazás funkcióit teszi elérhetővé a kliens alkalmazások számára egy RESTful API-n keresztül, Controller osztályok segítségével. Ez a réteg teszi lehetővé, hogy a felhasználók élni tudjanak azokkal a funkciókkal, szolgáltatásokkal, amiket a szoftver nyújt. A felhasználók által kezdeményezett műveleteket továbbítja az üzleti logikai réteg felé.

A rétegek közötti függőségek vizualizálásához létrehoztam egy diagramot, ezt a 2. ábra tartalmazza. A szaggatott nyíl a Web réteg és az Adathozzáférési réteg között azt reprezentálja, hogy itt csak azért van fordítás idejű függés a két réteg között, mivel a Web réteg tartalmazza a Startup osztályt, amiben a dependency injection konfigurálása történik és itt be kell regisztrálnunk az adathozzáférési réteg osztályait is. Tényleges logikai függés azonban nincs a két projekt között.



2. ábra - A rétegek és függőségeik

A Visual Studio solutionöm 5 különböző projektből áll:

* Domain, ez tartalmazza a Domain réteget.
* BLL (business logic), ez tartalmazza a Szolgáltatások rétegét.
* DAL (data access layer), ez tartalmazza az adathozzáférési réteget.
* CurlycircleWebApi, ez tartalmazza a Web réteget.
* UnitTests: ez tartalmazza a Szolgáltatások rétegéhez készült unit teszteket.

A projektek közötti referenciákat a rétegek közötti függőségeknek megfelelően állítottam be.

## Adatbázis felépítése

Az adatok tárolására relációs SQL alapú adatbázist használtam. A fejlesztési környezetben egy MSSQL lokális adatbázist használtam a számítógépemen, amelyet a Microsoft SQL Server Management Studio szoftver segítségével ellenőriztem. A Code-First megközelítést alkalmazva az entitás osztályok és a kontextus osztály implementálása, konfigurálása után a séma elkészítését az Entity Framework Core technológiára bíztam.

Az adatbázis felépítésének vizualizációjához létrehoztam egy diagramot, ezt a 3. ábra tartalmazza. Az adatbázis diagramját a Microsoft SQL Server Management Studio szoftver segítségével generáltam a tényleges adatbázis alapján. A diagramról szándékosan lehagytam néhány kevésbé lényeges táblát az áttekinthetőség érdekében. Ezek mind az ASP.NET Identity keretrendszer által létrehozott táblák, és a felhasználók különböző adatait tárolják, például a bejelentkezéseiket, a hozzáférési tokenjeiket, illetve a szerepeiket.

A webshopban regisztrált felhasználókat a *Users* táblában tárolom el. Egy felhasználóhoz pontosan egy kosár tartozik, azonban egy kosár létezhet felhasználó nélkül is, hiszen a nem bejelentkezett felhasználók is használhatják a webshopot. A kosarakat a *Carts* táblában tárolom el. Egy kosárhoz több *CartItem* is tartozhat, ez reprezentálja a kosárba tett termékeket, illetve azok mennyiségét. A *CartItem* tábla így egy kapcsolótábla a termékek és kosarak között. A leadott rendeléseket az *Orders* táblában tárolom. Egy rendelés tartozhat egy felhasználóhoz, amennyiben bejelentkezve lett leadva. Egy rendeléshez tartozhat egy *WebPayment*, amennyiben a webes kliensen fizetési módként webes fizetés lett kiválasztva, így annak részleteit a *WebPayments* táblában tárolom el. Egy rendeléshez tartozhat több *OrderItem*, ez egy kapcsolótábla a rendelések és a termékek között. A termékkategóriákat a *ProductCategories* táblában tárolom. Egy termékkategóriához több termék is tartozhat, azonban egy termék mindig pontosan egy termékkategóriához tartozik. A termékeket a *Products* tábla tartalmazza. Egy termékhez tartozhat egy minta, egy anyag, illetve több szín is. Ezeket az entitásokat a *Patterns*, *Materials*, illetve *Colors* táblákban tárolom. A *ColorProduct* tábla egy kapcsolótábla a termékek és színek között, amely a many-to-many relációt reprezentálja.



3. ábra - Az adatbázis felépítése

## Domain réteg

### Entitás osztályok

Ahogy korábban említettem, a Domain réteg tartalmazza az alkalmazás entitás osztályait, a repository interfészeket – ezeket használja a Szolgáltatások rétege – és a kivételeket. Emellett ez a réteg tartalmaz még néhány modell osztályt, amelyek nem kerülnek be az adatbázisba, tehát nem entitás osztályok, viszont az alkalmazás központi részéhez tartoznak. Az alkalmazás entitásai azok az osztályok, amelyek az üzleti logika alapját képezik, és amelyeket az adatbázisban is eltárolok. A korábban bemutatott adatbázis diagramon ezeknek az osztályoknak megfelelő táblák találhatók.

Minden entitásnak szüksége van egy azonosítóra ahhoz, hogy dolgozni tudjak velük és ahhoz, hogy az adatbázisban el lehessen tárolni őket. Ezért létrehoztam egy *EntityBase* absztrakt osztályt, amely az összes entitás osztály őse és egy integer típusú *Id* nevű tulajdonságot tartalmaz. Ez lesz az entitások azonosítója.

Az entitások mellett egy rendszerben szerepelnek olyan objektumok és adatok, amelyekhez nincs értelme azonosítót rendelni, ilyenek például a value objectek. A webshopban például egy rendeléshez eltárolt cím egy value object, hiszen nincs értelme azonosítót hozzárendelni, önmagában sosem létezhet, mindig egy entitáshoz tartozik. Gyakorlatilag a cím egy komplex érték, ami egy városból, irányítószámból és hasonló mezőkből tevődik össze. A value objectek implementáláshoz létrehoztam a *ValueObject* absztrakt ősosztályt a Microsoft útmutatója alapján. [32] Ez az ősosztály alapvető kisegítő metódusokat tartalmaz, mint például két value object összehasonlítása a mezőik értékeinek alapján.

A webshopban alapvetően termékek vannak, amelyeket a vevők meg tudnak vásárolni. Egy terméket a *Product* entitás reprezentál, ennek főbb tulajdonságai a neve, ára, leírása, illetve a hozzá tartozó képek URL azonosítói. Emellett a termék elérhetőségét egy bool típusú tulajdonságban tárolom. Egy termék pontosan egy termékkategóriához tartozik, ehhez a *ProductCategory* entitást hoztam létre. Ennek is van neve, leírása, képe, illetve elérhetőséget jelző tulajdonsága. Egy terméknek lehet anyaga, mintája, illetve több színe is, azonban ezek mind opcionális relációk. Ezeket a *Material*, *Pattern* és *Color* entitásokkal reprezentálom, melyeknek mind egyetlen fő tulajdonságuk van, a nevük. Ezek alapján tudnak rászűrni a termékekre a felhasználók a webes kliensen.

A webshop leadott rendeléseket is eltárolja, hogy az adminisztrátor láthassa ezeket, illetve, hogy a bejelentkezett felhasználók megtekinthessék a korábbi rendeléseiket. A rendeléseket az *Order* entitással reprezentálom. Ez eltárolja a rendeléshez szükséges összes adatot, például a rendelő elérhetőségeit, nevét, címét, a rendelő által leadott megjegyzést, a kiválasztott szállítási és fizetési módot, a rendelés összegét és idejét. Emellett tartozik a rendeléshez több *OrderItem*, amely a termékek és a rendelések közötti many-to-many kapcsolatot reprezentáló entitás. A rendeléshez opcionálisan tartozhat egy felhasználó is, amennyiben bejelentkezve lett leadva a rendelés és egy webes fizetést reprezentáló entitás is, amennyiben a fizetési módnál a webes fizetés lett kiválasztva. A rendeléshez megadott címet egy *OrderAddress* típusú value object segítségével implementáltam, az adatbázisban ez azonban egyszerű oszlopokra van bontva a tulajdonságai alapján. A lehetséges fizetési és szállítási módokat a *PaymentMethod*, illetve a *ShippingMethod* enumeráció típusokkal reprezentálom.

A webshopban a regisztrált felhasználók adatait az ApplicationUser entitás osztállyal reprezentálom, amely az *IdentityUser* osztályból származik le. Az *IdentityUser* az ASP.NET Core Identity keretrendszer osztálya, alapvető tulajdonságokat definiál a felhasználók kezelésre számára. Ilyen tulajdonságok például a felhasználó e-mail címe, jelszójának hashelt értéke, azonosítója, telefonszáma és a többi. Emellett az *ApplicationUser* osztályban további tulajdonságokat definiáltam: a felhasználók rendelkeznek vezetéknévvel, keresztnévvel, refresh tokennel, illetve egy címmel. Ezt a címet a *UserAddress* osztály segítségével reprezentálom. Emellett egy felhasználóhoz tartozhat több rendelés, illetve egy kosár entitás is.

A felhasználói szerepeket az *ApplicationRole* entitás osztállyal reprezentálom, ez az Identity keretrendszer *IdentityRole* osztályából származik le. Az alkalmazásomban csupán két szerepet definiáltam, a *User* és az *Admin* szerepeket. Egy felhasználóhoz pontosan egy szerep tartozik, sima felhasználó vagy adminisztrátor. Ezt reprezentálja a *Role* enumeráció típus.

A *Cart* entitás osztállyal reprezentálom a webshop felhasználóinak kosarát, amelybe termékeket tehetnek. Minden regisztrált felhasználóhoz tartozik egy kosár, azonban a kosarak önmagukban is létezhetnek, mivel az anonim, azaz a nem bejelentkezett felhasználók is használhatják a webshopot. Ezért a *Cart* és az *ApplicationUser* között a reláció opcionális. Egy kosár tartalmazhat többféle terméket is, illetve egy adott termékből több darabot is. Ezt a kapcsolatot reprezentálom a *CartItem* entitás osztály segítségével, amely eltárolja az adott termék mennyiségét és árát is. Első ránézésre redundánsnak tűnhet a termék árát itt is eltárolni, azonban később, ha például adott megrendelt mennyiség utáni kedvezményeket szeretnénk bevezetni, akkor ahhoz ez már szükséges lenne. Így az üzleti logika jobban fel van készítve később felmerülő igényekre.

### Repository interfészek

A Domain rétegben definiáltam a repository interfészeket. Ezeket az interfészeket implementáltam az Adathozzáférési rétegben, és ezeket az interfészeket használja a Szolgáltatások rétege. Az összes főbb entitáshoz definiáltam repository interfészeket, ezek az interfészek definiálják a megfelelő írási és olvasási műveleteket, amelyek a Szolgáltatások rétegéből lesznek felhívva. Emellett itt definiáltam az *IUnitOfWork* interfészt is, amelyet a Szolgáltatások rétegében adatok adatbázisba való mentéséhez használtam. Ezt is az Adathozzáférési rétegben implementáltam. Nem sorolom fel az összes definiált interfészt és azok függvényeit, csupán egyet bemutatok, hiszen az összes interfész lényegében hasonló műveleteket tartalmaz.

Az *IProductRepository* a termékekkel kapcsolatos műveletek függvényeit definiálja.

public interface IProductRepository

{

int AddProduct(Product product);

Task<PagedList<Product>> GetAllAsync(ProductQueryParameters   
 productQueryParameters);

Task<Product> GetProductByIdAsync(int productId);

void UpdateProduct(Product product);

Task DeleteProductAsync(int productId);

}

A kódrészletből jól látszik a függvények nevei alapján, hogy milyen műveletekért felelősek. Például az *AddProduct()* egy termék hozzáadását végzi, míg a *GetProductByIdAsync()* egy terméket kérdez le az adatbázisból az azonosítója alapján. A hosszabb ideig tartó műveleteket aszinkron függvényekkel valósítom meg, ezek visszatérési értéke egy *Task* típusba van csomagolva, illetve neveik mind az „Async” szóval végződnek.

### Kivételek

A Domain rétegben előfordulhatnak kivételek, például, ha egy entitást az azonosítója alapján akarunk lekérdezni, és ez nem található az adatbázisban, akkor ezt egy kivétel dobásával jelzem. Különböző osztályokat hoztam létre a különböző típusú kivételek reprezentálására. Ezek az osztályok mind a .NET keretrendszer *Exception* osztályából származnak le. A különböző kivétel osztályok értelme az, hogy később a típusok alapján a globális hibakezelő middleware-ben a különböző típusú kivételekre különbözőképp tudok reagálni. Például pontosabb hibaüzenetben tudom tájékoztatni a kliens alkalmazásokat. A nem található entitás miatt dobott kivételeknél az *EntityNotFoundException* osztályt, a rosszul megadott paramétereknél a *BadParameterException* osztályt, míg az egyéb kivételeknél az általánosabb *DomainException* osztályt használom.

### Kisegítő osztályok

A Domain rétegben létrehoztam különböző kisegítő osztályokat, melyek nem entitások, azonban az alkalmazás központi részéhez tartoznak, így ebben a rétegben van a helyük.

Létrehoztam egy *QueryParameters* absztrakt osztályt, amelyből az *OrderQueryParameters* és a *ProductQueryParameters* osztályokkal származtam le. Ezekre az osztályokra a szerver oldali lapozás (pagination) miatt van szükség. A lapozás lényege, hogy ha egy kérésre nagyon sok entitást kéne lekérdeznünk az adatbázisból, akkor az eredménynek csak egy adott részhalmazát adjuk vissza, és a kliens alkalmazásokban engedjük lapozni a felhasználót. Ezzel hálózati adatforgalmat, és időt spórolunk meg. Az előbb említett osztályok hordozzák a rendelések és a termékek lapozásához szükséges adatokat. A *QueryParameters* osztály tartalmazza a lekérdezett lap indexét és a lap méretét. A leszármazott osztályok emellett a rendelések és termékek szűréséhez szükséges adatokat is hordozzák.

public class ProductQueryParameters : QueryParameters

{

public int? ProductCategoryId { get; set; }

public IEnumerable<int> ColorIds { get; set; } = new List<int>();

public IEnumerable<int> PatternIds { get; set; } = new List<int>();

public IEnumerable<int> MaterialIds { get; set; } = new List<int>();

public int MinPrice { get; set; } = 0;

public int MaxPrice { get; set; } = int.MaxValue;

public bool ValidPriceRange => MaxPrice >= MinPrice;

}

A kódrészleten láthatóak a termékek szűréséhez szükséges adatok. A kliens alkalmazások ennek segítségével rá tudnak szűrni a termékek színeire, mintájára, anyagára, megadhatnak egy ár intervallumot, illetve megadhatják, hogy melyik termékkategórián belüli termékeket szeretnék lekérdezni.

A lapozáshoz továbbá létrehoztam egy *PagedList* nevű osztályt, amely a .NET keretrendszer *List* osztályából származik le. A lekérdezett rendelés és termék entitásokat ennek az osztálynak a segítségével adom vissza. Ez tartalmazza az adatokat a kliens alkalmazás számára, amellyel megjelenítheti a megfelelő lapozási információkat a felületen, például az oldal indexét, az összes oldal számát és az oldalak méretét. Emellett az osztályban létrehoztam egy *CreateAsync()* függvényt, amely egy *IQueryable* típusból hoz létre egy *PagedList* típust a lapozás paraméterei alapján, ezt használom fel később a megfelelő repository osztályokban.

var count = await source.CountAsync();

var items = await source.Skip((pageIndex) \*   
 pageSize).Take(pageSize).ToListAsync();

return new PagedList<T>(items, count, pageIndex, pageSize);

## Adathozzáférési réteg

Az adathozzáférési réteg tartalmazza az adatbázis elérésének implementációs részleteit, ezt eltakarja a többi réteg elől. Fordítás időben a többi réteg a Domain rétegben definiált repository interfészeken keresztül éri el az adatokat. Futásidőben természetesen az interfészek mögé bekerülnek az Adathozzáférési rétegben létrehozott repository osztályok.

### Adatbázis modell, konfiguráció

Az adatbázissal való kommunikációt és az adatok leképezését objektumokká az Entity Framework Core (EF Core) keretrendszer segítségével végeztem. Így .NET objektumokon keresztül tudtam az adatbázissal dolgozni és semmilyen SQL kódot nem kellett írnom. Az EF Core keretrendszerben az adathozzáférés egy úgynevezett modellen keresztül történik. A modell entitás osztályokból és egy kontextus osztályból áll, amely az adatbázissal való kapcsolatot reprezentálja. Ebből a modellből az EF migrációk segítségével generáltam az adatbázist.

Az alkalmazásomban az *ApplicationDbContext* az adatbázis kontextus osztály. Ezt az Identity keretrendszer *IdentityDbContext* osztályából származtattam le, mivel ezt az ősosztályt használva az alkalmazás adatbázisa automatikusan fel lesz készítve a felhasználókezelésre, létrejönnek az ahhoz szükséges táblák. Az *IdentityDbContext* osztálynak megadtam az *ApplicationUser* és az *ApplicationRole* osztályokat, így tudja, hogy ezeket az osztályokat kell használnia a felhasználók és a szerepek létrehozására. Ezen az osztályon belül definiálom az adatbázis tábláit *DbSet* típusú publikus tulajdonságok létrehozásával.

Az EF Core az entitás osztályok alapján létrehoz egy modellt, azonban ezt a modellt további konfigurációkkal kellett ellátnom, hogy megfelelően nézzen ki az adatbázis sémája. A konfigurációt a *ModelBuilder* API segítségével készítettem el. Ennek a segítségével az entitás osztályok módosítása nélkül, fluent módon, azaz ember által könnyen olvasható módon tudtam elvégezni a modell konfigurálását. Ezt úgy végeztem el, hogy az osztályban implementáltam az ősosztály által definiált *OnModelCreating()* függvényt és ezen belül a *ModelBuilder* objektum segítségével konfiguráltam be az adatbázis modelljét. Itt végeztem el például a *PaymentMethod* és *ShippingMethod* enumeráció típusok konfigurálását, így az adatbázisba *string* típusként lesznek elmentve, majd onnan kiolvasva át lesznek váltva a megfelelő *enum* értékre.

Annak érdekében, hogy az *OnModelCreating()* metódus méretét csökkentsem, és átláthatóbbá tegyem, az entitások konfigurálását kiszerveztem külön *EntityTypeConfiguration* osztályokba. Ezekben az osztályokban implementáltam az *IEntityTypeConfiguration* interfészt, így a *Configure()* függvényben megadtam az adott entitás konfigurációját.

builder.ToTable("Products");

builder.HasKey(c => c.Id);

builder.HasOne(p => p.ProductCategory)

.WithMany(nameof(ProductCategory.Products))

.HasForeignKey(nameof(Product.ProductCategoryId));

A kódrészleten a *Product* entitás konfigurálása látható. A *ToTable()* függvénnyel megadom, hogy mi legyen a termékeket tartalmazó tábla neve és a *HasKey()* metódussal beállítom, hogy melyik tulajdonságot használja azonosítóként. Ezután több metódus egymás utáni láncolásával definiálom a termékkategóriák és termékek között lévő one-to-many kapcsolatot, valamint megadom, hogy melyik tulajdonságot használja idegen kulcsként.

A korábban említett value objecteket, tehát a rendeléshez és felhasználóhoz tartozó címeket owned entity-ként tárolom el az adatbázisban. Az EF Core owned entity funkciója segítségével hozzá tudunk rendelni olyan típusokat az entitásainkhoz, amelyeknek nincs saját azonosítójuk a domain modellben. Így az adatbázisban az owned entity tulajdonságai le lesznek képezve a hozzá tartozó entitás oszlopaira.

orderConfiguration.OwnsOne(o => o.Address, a =>

{

a.WithOwner(a => a.Order).HasForeignKey(a => a.OrderId);

a.Property(a => a.ZipCode).HasColumnName("ZipCode");

a.Property(a => a.City).HasColumnName("City");

a.Property(a => a.Line1).HasColumnName("Line1");

a.Property(a => a.Line2).HasColumnName("Line2");

});

A kódrészletből látható, hogy a rendelés entitás *OrderAddress* típusú *Address* tulajdonságát beállítottam owned entity-ként az *OwnsOne()* függvény segítségével, és ugyanitt megadom az oszlopok neveit, amelyekre az *OrderAddress* típusú objektum tulajdonságai le lesznek képezve az adatbázisban.

A Domain rétegben definiált *IUnitOfWork* osztályt is az Adathozzáférési rétegben implementálom. Az EF Core *DbContext* osztálya úgy van kitalálva, hogy egy példányát egyetlen számítási egységhez (unit-of-work) használjuk, ezután ezt a példányt eldobjuk. Ez azt jelenti, hogy a *DbContext* példány életideje általában nagyon rövid. Egy unit-of-work általában a következőképp néz ki: létrehozunk egy *DbContext* példányt, majd a kontextus osztály elkezd entitásokat trackelni, miután le lettek kérdezve az adatbázisból. Ezután változtatásokat viszünk véghez a trackelt entitásokon, majd felhívjuk a kontextus osztály *SaveChangesAsync()* függvényét, ami beírja a változtatásokat az adatbázisba. Végül a *DbContext* példányt eldobjuk. Mivel a szolgáltatások rétegében nincs referenciám a *DbContext* objektumra, ezért a *UnitOfWork* osztályon keresztül elérhetővé tettem a kontextus osztály *SaveChangesAsync()* függvényét. Így a Szolgáltatások rétegében, miután egy függvény véghez vitte a megfelelő módosításokat a trackelt entitásokon, fel tudja hívni ezt a függvényt, és az összes változtatás egy tranzakcióként lesz beírva az adatbázisba.

### Repository osztályok

Az Adathozzáférési rétegben implementálom a Domain réteg által definiált repository interfészeket. A repository osztályok nem tartalmaznak üzleti logikát, így a legtöbb függvényük elég rövid, azonban olyan helyeken van kivétel, ahol az adathozzáférési logika is bonyolultabb. Ilyen eset például a termékek lekérdezése, ahol a korábban említett szerver oldali pagination logikát implementálom. A repository osztályok mindegyike megkapja konstruktor injektálással az *ApplicationDbContext* objektumot, így ezen keresztül érik el az adatbázist. A best-practice-nek megfelelően a repository osztályok legtöbb függvénye aszinkron, így nem blokkolják az őket felhívó szálakat. Ezek alól csupán azok a függvények a kivételek, ahol nagyon egyszerű műveleteket végzünk el, például egyetlen entitást adunk hozzá az adatbázishoz.

Példaként bemutatom a *ProductRepository* osztály termékeket lekérdező repository függvényét.

public async Task<PagedList<Product>> GetAllAsync(ProductQueryParameters productQueryParameters)

{

if (!productQueryParameters.ValidPriceRange)

{

throw new BadParameterException(

"Max price of product cannot be less than min price of product."

);

}

Látható, hogy a függvény elején ellenőrzőm, hogy a kliens alkalmazástól kapott paraméterek érvényesek-e, amennyiben nem, akkor a megfelelő kivételt dobom egy üzenettel.

var products = \_dbContext.Products

.Where(p => p.Price >= productQueryParameters.MinPrice &&

p.Price <= productQueryParameters.MaxPrice && p.IsAvailable);

Ezt követően lekérdezem a kontextus objektumon keresztül az ár intervallumnak megfelelő termékeket.

var colors = await \_colorRepository.GetColorsByIdsAsync(productQueryParameters.ColorIds);

var materials = await \_materialRepository.GetMaterialsByIdAsync(  
productQueryParameters.MaterialIds);

var patterns = await \_patternRepository.GetPatternsByIdAsync(productQueryParameters.PatternIds);

Ezután lekérdezem a paraméterek alapján a megfelelő szín, anyag és minta entitásokat a többi repository osztály segítségével. Ezt követően segéd függvényeken keresztül leszűröm a termékeket a paraméterek alapján. Ezek a függvények összehasonlítják a termékek tulajdonságait a paraméterben megadott szűrési feltételekkel. Például a *SearchByPattern()* függvény azokat a termékeket adja vissza, ahol a paraméterben megadott minták között szerepel a termék mintája:

products = products.Where(p => p.Pattern != null && patterns.Contains(p.Pattern));

Végül a *PagedList* osztály *CreateAsync()* függvényének segítségével az *IQueryable* típusú termékkollekcióból, és a paraméterekben megadott oldal index és oldal méret alapján létrehozom a megfelelő lapozott listát, és ezt visszaadom a hívónak:

return await PagedList<Product>.CreateAsync(products,   
 productQueryParameters.PageIndex, productQueryParameters.PageSize);

}

Ezen a példán keresztül lehetett látni, hogy a Domain rétegben definiált kivételeket hogyan használom, illetve a repository függvények hogyan érik el az adatbázist a kontextus objektumon keresztül.

### Adatbázis feltöltése alapadatokkal

Az adathozzáférési rétegben végeztem el az adatbázis alapadatokkal való feltöltését is. Ez a fejlesztési és tesztelési folyamatokat könnyítette meg. Az EF Core lehetővé teszi, hogy már a modellt úgy konfiguráljam be, hogy megadok alap adatokat, amelyekkel feltölti az adatbázist a létrehozása közben. Létrehoztam egy *Seed()* nevű extension metódust a *ModelBuilder* típushoz, és ezt felhívtam a korábban említett *OnModelCreating()* függvény végén. Ebben a *Seed()* függvényben végeztem el az alap adatok megadását. Például hozzáadtam az adminisztrátori és felhasználói szerepeket, illetve egy-egy felhasználót, akik ezekkel a szerepekkel rendelkeznek. Emellett hozzáadtam termékeket, termékkategóriákat, színeket, mintákat és anyagokat.

## Szolgáltatások rétege

A Szolgáltatások rétege szolgáltató (Service) osztályokon keresztül implementálja az alkalmazás üzleti logikáját. A szolgáltató osztályok mindegyikéhez tartozik egy interfész, így a Web réteg ezektől az interfészektől függ, a tényleges implementációktól nem. A réteg emellett tartalmaz ViewModel és Data Transfer Object (DTO) osztályokat is, amelyek a hálózati kommunikációhoz szükségesek, illetve kisegítő osztályokat. Itt találhatóak olyan kivétel osztályok is, amelyek üzleti logikából adódnak és itt alakulhatnak ki.

### DTO és ViewModel osztályok

A kliens alkalmazásokkal való kommunikáció során nem előnyös a Domain rétegben definiált entitás osztályokat használni, hiszen adott esetekben sok felesleges adatot is küldenénk így, illetve olyan adatot is küldenénk, amelyet a kliensnek nem kéne látnia. Ennek megoldására létrehoztam DTO és ViewModel osztályokat a klienssel való hálózati kommunikációhoz. A DTO és ViewModel osztályoknak megfelelő osztályokat definiáltam a kliens alkalmazásokban is, így teljesen mértékben olyan objektumokkal dolgoznak, amelyeket a szerver alkalmazással való kommunikáció során is használniuk kell.

A DTO osztályok azokat az adatokat reprezentálják, amelyeket a kliens alkalmazások küldhetnek számunkra a kérések testében. Minden különböző kéréshez, ahol adatot kell küldeni definiáltam egy megfelelő DTO osztályt, így a kliens alkalmazásoknak mindig csak a ténylegesen szükséges mennyiségű adatot kell küldeniük. A DTO osztályok neveit igyekeztem beszédesre írni, így például a regisztrációhoz szükséges adatokat a *RegisterDto* osztály, míg egy termék hozzáadásához, vagy módosításához szükséges adatokat a *ProductUpsertDto* osztály tartalmazza.

A DTO osztályokat a kliens oldalon is validálom, azonban természetesen szükség van szerver oldali validációra is. Ezt a FluentValidation könyvtár segítségével valósítottam meg. Ennek lényege, hogy attribútumok helyett ember által jól olvasható módon, áttekinthetően írhatjuk meg a validációs logikát. A validációt úgy végeztem el, hogy minden DTO osztályhoz létrehoztam egy neki megfelelő validációs osztályt, amelyet a FluentValidation könyvtár *AbstractValidator* ősosztályból származtattam le. Így a validátor osztály konstruktorában a *RuleFor()* függvény segítségével validációs szabályokat hoztam létre.

public ChangePasswordDtoValidator()

{

RuleFor(ent => ent.Email)

.NotEmpty().WithMessage("Email is required.");

//...

}

A kódrészleten a jelszó változtatáshoz kapcsolódó DTO osztály validátorának konstruktora látható. Látszik, hogy az e-mail cím kötelező tulajdonság, nem lehet üres. Ha egy bejövő kérés DTO osztálya nem felel meg a validátor osztály szabályainak, akkor a kérés automatikusan el lesz utasítva a megfelelő hibaüzenettel.

A ViewModel osztályok azokat az adatokat reprezentálják, amelyeket a szerver alkalmazásról küldök el a kliens alkalmazások számára. Minden szcenárióhoz, ahol különböző adatokra van szükség definiáltam egy külön ViewModel osztályt, így a kliensnek mindig csak a ténylegesen szükséges adatokat küldöm el, ezzel hálózati forgalmat spórolva. Az olyan kéréseknél, ahol egy új entitást hoztam létre, ott az *EntityCreatedViewModel* típust használom. Ez csak egy tulajdonságot tartalmaz, az újonnan létrehozott entitás azonosítóját.

A DTO és ViewModel objektumokat természetesen a Szolgáltatások rétegében le kell képeznem entitás objektumokká, hogy azokkal dolgozni tudjak. Erre az AutoMapper könyvtárat használtam, melynek lényege, hogy megspórolhatom vele a leképezések kézzel írását. A könyvtár segítségével létrehoztam leképezéseket a megfelelő DTO-entitás és ViewModel-entitás osztály párosok között. Ezeket a leképezéseket az *AutoMapperProfile* osztályban tárolom, amely a könyvtár *Profile* osztályából származik le. Ennek a konstruktorában a *CreateMap()* függvény segítségével hoztam létre a leképezéseket.

CreateMap<Product, ProductViewModel>()

.ForMember(pvm => pvm.ImageUrls, options => options.MapFrom(

p => p.ImageUrls.Split(';', StringSplitOptions.RemoveEmptyEntries)

.ToList()

)

);

A kódrészleten látszik a *Product* és *ProductViewModel* típusok közötti leképezés, itt a legtöbb tulajdonságot a neveik alapján az AutoMapper automatikusan leképezi egymásra, azonban az *ImageUrls* tulajdonságnál speciális leképezési logikára van szükség. Az entitás osztályban az *ImageUrls* tulajdonság string típusú, az összes termékhez tartozó kép URL azonosítóját tartalmazza, azokat pontos vesszővel elválasztva. A ViewModel osztály azonban egy string listaként tartalmazza az URL azonosítókat, mivel így a kliens alkalmazásokban sokkal könnyeben lehet ezeket használni. Ezért itt létrehoztam ezt a speciális leképezési logikát, ahol a beépített *Split()* függvény segítségével felbontom az eredeti stringet a pontos vesszők mentén egy string listába.

### Kivételek

A Szolgáltatások rétegében is definiáltam kivétel osztályokat. Itt olyan kivételek szerepelnek, amelyek nem a Domain entitásokhoz kapcsolódnak, hanem az üzleti logika alapján keletkeznek a Service osztályokban. Ilyen például a *NoAccessException*, amit akkor dobok, ha az adott felhasználónak nincs jogosultsága egy művelethez, illetve a *ValidationAppException*, amit validációs hiba esetén dobok. Ilyen lehet például egy helytelen jelszó megadása.

### Service osztályok

A Service osztályok tartalmazzák az alkalmazás üzleti logikáját. Minden főbb funkcióhoz tartozik egy Service osztály. Például a felhasználókezelésért felelős osztály az *AuthService*, ennek a függvényeiben a szükséges műveleteket. Az összes Service osztályra jellemző, hogy a Web rétegből hívom fel a függvényeiket, és a Web rétegben egy beérkezett kéréshez mindig pontosan egy Service osztály függvényét hívom fel. Tehát egy Service osztálybeli függvény egy adott típusú kérés kiszolgálásához szükséges logikát tartalmaz. Ezek a függvények lekérdeznek, létrehoznak, módosítanak, illetve törölnek domain entitásokat az üzleti logikai szabályoknak megfelelően. Ezen műveletek elvégzéséhez az Domain rétegben definiált repository interfészeket használják.

#### AuthService osztály

Ahogy korábban említettem, az *AuthService* osztály a felhasználókezeléshez szükséges logikát tartalmazza. Ő egyrészt használja az ASP.NET Identity keretrendszerének *UserManager* osztályát, amely gyakorlatilag egy beépített repository osztály a felhasználókezeléshez, másrészt használja az *ICartRepository* interfészt, mivel a felhasználókkal együtt azok kosarait is kezelnie kell. Példaként bemutatom a *LoginAsync()* függvényt, amely a felhasználó bejelentkezését kezeli. Ez egy hosszabb kódrészlet, így a függelékben található.

A kódrészletből látható, hogy a függvény egy *LoginDto* típusú DTO objektumot vár, illetve egy *UserViewModel* típust fog visszaadni. A *FindUserByEmailAsync()* privát függvénnyel kikeresek egy felhasználót az e-mail címe alapján az adatbázisból a *UserManager* osztály segítségével. Ezután ellenőrzöm a megadott jelszót, amennyiben ez nem stimmel egy validációs kivételt dobok a megfelelő üzenettel. Ezután privát segédfüggvényekkel létrehozom a felhasználó hozzáférési és frissítési tokenjeit, valamit lekérem az adatbázisból a felhasználóhoz tartozó szerepeket. Egy egyszerű logikával ezután beállítom a felhasználó szerepét a *Role* enumeráció segítségével. Ezt követően lekérem az *ICartRepository* interfészen keresztül a felhasználóhoz tartozó kosarat az adatbázisból. Itt egy egyszerű logikával hozzáadom a felhasználó eddigi, anonim kosarából a termékeket a fiókjához tartozó kosarába, amennyiben a bejelentkezés előtt is rakott termékeket a kosárba. Ezt onnan tudom, hogy megtette-e, hogy a kliens alkalmazás a *LoginDto* objektumban elküldi az anonim kosár azonosítóját. Ha nem küld azonosítót, az azt jelenti, hogy nem volt anonim kosara. Végül kitörlöm a régi kosarat, és elmentem az *IUnitOfWork* interfészen keresztül a változtatásokat az adatbázisba. Ezután elvégzem az entitás leképezését a megfelelő ViewModel típusra, és beállítom azokat a tulajdonságokat is rajta, amelyek más forrásból származnak – szerep, tokenek. Végül visszaadom a ViewModel objektumot a hívónak.

Az *AuthService* osztály a bejelentkezés mellett kezeli a regisztrációt, a felhasználó tokenek frissítését, a felhasználói adatok lekérését, módosítását és a jelszó megváltoztatását .

#### ProductCategoryService és ProductService osztályok

Ezek az osztályok felelnek a termékkategóriákhoz és termékekhez tartozó logika implementálásáért. A *ProductCategoryService* osztályt részletesen nem mutatom be, logikailag nagyon hasonló a *ProductService* osztályhoz, függvényeivel termékkategóriákat hozhatunk létre, kérdezhetünk le, módosíthatunk, illetve törölhetünk.

A *ProductService* osztály függvényei a termék entitásokhoz kapcsolódó műveleteket végzik el. Egy termékhez tartozhat egy bélyegkép, illetve több sima kép, amelyek a webshop felhasználói felületén fognak megjelenni. Ezeket a képeket az adminisztrátor felhasználó töltheti fel az egyes termékekhez, amikor létrehozza vagy szerkeszti azokat. A képek kezelésére létrehoztam egy külön *ImageHelper* segédosztályt.

Az *ImageHelper* segédosztály két publikus függvényt tartalmaz, az egyikkel bélyegkép fájlokat hozhatunk létre, a másikkal rendes képfájlokat. Ezt az osztályt használtam a termékkategóriák és a termékek képeinek elmentésére is.

A *CreateThumbnailFile()* függvény segítségével mentem el a bélyegképnek szánt képfájlokat. Egy termékhez és egy termékkategóriához is pontosan egy bélyegkép fájl tartozhat. Először ellenőrzöm a fájl kiterjesztését és méretét, ha ezek nem megfelelőek, akkor validációs kivételeket dobok. Ezután generálok egy új, egyedi fájlnevet, majd létrehozom a megadott útvonal alapján a fájl mentési helyét. A termékkategóriákhoz és a termékekhez tartozó bélyegképet képeket külön mappákban tárolom.

await file.CopyToAsync(memoryStream);

memoryStream.Position = 0;

using Image image = Image.Load(memoryStream, out IImageFormat format);

image.Mutate(x => x.Resize(width: 800, height: 0));

using var outputStream = new FileStream(filePath, FileMode.Create);

image.Save(outputStream, format);

A bélyegképek felbontását elmentés előtt lekicsinyítem, mivel ezeket a felhasználói felületen mindig kis méretben fogom megjeleníteni, így teljesen felesleges az eredeti nagy felbontásban tárolni, és küldeni őket. Ezt az ImageSharp [12] könyvtár segítségével oldottam meg. A könyvtár tartalmaz egy *Image* típust, amelyen elvégezhetjük a kívánt módosításokat a képen. Az *Image* típusú objektumot egy *MemoryStream* objektumból létrehoztam a *Load()* függvény segítségével, majd a *Mutate()* függvény segítségével módosítottam a felbontását. Annak érdekében, hogy a képek eredeti képarányát megőrízzem, a magasságnak 0 értéket állítottam be. Végül egy *FileStream* objektummal és a *Save()* függvénnyel elmentem a képet a megadott útvonalra.

A *ProductService* osztály *CreateProductAsync()* függvényével hozom létre az új termékeket. Először a DTO objektumban megadott képfájlok alapján létrehozom a bélyegkép fájlt, illetve a képfájlokat, majd elvégzem a Product típusú entitás osztály létrehozását a DTO objektum és a visszakapott fájl útvonalak alapján.

var id = \_productRepository.AddProduct(product);

await \_unitOfWork.SaveChangesAsync();

return new EntityCreatedViewModel(id);

Végül az *IProductRepository* interfészen keresztül hozzáadom a terméket az adatbázishoz, elmenten a változtatásokat és visszaadom a megfelelő ViewModel objektumot. A *ProductService* osztály tartalmaz további függvényeket a termékek módosítására, törlésére, a lapozott és szűrt termékek lekérdezésére és egy adott termék azonosító alapján való lekérdezésére. Fontos, hogy a termékek és termékkategóriák törlésénél nem távolítom el ténylegesen az entitásokat az adatbázisból, hanem átállítom az *IsAvailable* tulajdonságukat hamis értékűre. Így amikor az összes terméket vagy termékkategóriát kérdezik le a kliensek a webshopban való megjelenítéshez, abban a listában nem fognak szerepelni, azonban amikor azonosító szerint vannak lekérdezve akkor igen. Így a felhasználó például a korábbi rendelései között láthatja azokat a termékeket is, amelyek már nem elérhetőek a webshopban.

#### MaterialService, ColorService, PatternService osztályok

Ezek az osztályok felelősek az anyag, szín és minta entitásokon elvégzett műveletekért. Ezek az osztályok logikailag teljesen hasonlóak, illetve implementációjuk nagyon egyszerű, így nem mutatom be őket részletesebben. Mindegyikük tartalmaz függvényeket a megfelelő entitások lekérdezéséhez, hozzáadásához, törléséhez.

#### CartService osztály

A *CartService* osztályban a kosarak kezelésének logikáját implementáltam. Az osztály függvényeivel létrehozhatunk, törölhetünk, kiüríthetünk, lekérdezhetünk kosarakat. Emellett hozzáadhatunk, eltávolíthatunk, módosíthatunk és lekérdezhetünk *CartItem* entitásokat az adott kosarakon belül.

var cartItem = \_mapper.Map<CartItem>(cartItemCreateDto);

await \_cartRepository.AddCartItemAsync(cartId, cartItem);

await \_unitOfWork.SaveChangesAsync();

return new EntityCreatedViewModel(cartItem.Id);

A fenti kódrészlet az *AddCartItemAsync(int cartId, CartItemUpsertDto cartItemCreateDto)* függvényből származik. A megadott kosár azonosító és DTO objektum alapján az *ICartRepository* interfészen keresztül hozzáadom az adatbázisban az adott kosárhoz az új *CartItem* objektumot, majd elmentem a változtatásokat.

#### OrderService osztály

Az *OrderService* osztályban implementáltam a rendelés entitásokhoz kapcsolódó logikát. Az osztály függvényeivel létre lehet hozni egy új rendelést, le lehet kérdezni egy felhasználó leadott rendeléseit, le lehet kérdezni egy rendelést az azonosítója alapján és le lehet kérdezni az összes rendelést lapozott formában. Emellett ez az osztály felelős a webes fizetéssel leadott rendelések kezeléséért és a webes fizetés lebonyolításáért is.

A webes fizetés lebonyolításához a Barion szolgáltatót választottam. Ehhez a Barion sandbox környezetében regisztráltam egy boltot CurlyCircle névvel, és így kapott a boltom egy POSKey azonosítót. A Barion rendszer használatával a webes fizetés folyamata a következőképp néz ki:

1. A vevő a webes kliensben a webes fizetést választja és leadja a rendelését, ezzel meghívja a szerver alkalmazásom megfelelő végpontját.
2. A szerver alkalmazásom elindítja a webes fizetést, meghívja a Barion API /Payment/Start végpontját és megadja a POSKey kulcsot, amellyel a Barion rendszer azonosítja a boltomat. Emellett megad egyéb adatokat is, mint például, a rendelés részleteit, a végpontot, amelyen értesíteni tudja a szerver alkalmazásomat, az URL-t, amelyre vissza kell irányítani a felhasználót a webes fizetés után. A Barion visszaad egy fizetési azonosítót, és ezt a szerver alkalmazásom elmenti a rendelést és a hozzá tartozó webes fizetés entitást az adatbázisba. Kiürítem a felhasználó kosarát. Visszaküldöm a választ a webes kliensnek, hogy indulhat a webes fizetés.
3. A kliens alkalmazás átirányítja a felhasználót a Barion felületére.
4. A felhasználó elvégzi a webes fizetést.
5. Ha a fizetés állapotában változás történik (sikeres vagy sikertelen fizetés esetében is), a Barion szerver értesíti a szerver alkalmazásomat, a korábban megadott végpont meghívásával. A szerver alkalmazás ezután meghívja a Barion API /Payment/GetPaymentState végpontját, amely válaszként megadja, hogy a fizetés sikeres volt-e.
6. A szerver alkalmazásom elmenti az adatbázisba a webes fizetés új állapotát.
7. A felhasználót visszairányítja a Barion a megadott URL-re a böngészőjében, így visszatér a felhasználó a webes kliensre.

Ezt a logikát az *OrderService* osztályban két függvényben implementáltam. A *CreateWebPaymentRequestAsync()* függvényben indítom el a webes fizetést, itt hívom meg a Barion API megfelelő végpontját és mentem el az adatbázisba rendelés részleteit. A *HandleWebPaymentStatusChangedAsync()* függvényben kezelem le azt, amikor változás következett be a fizetés állapotában, és a Barion rendszer értesítette a backend alkalmazásomat. A webes fizetés folyamatábráját a függelékben a 5. ábra tartalmazza.

A Barion API-val való kommunikációhoz a RestSharp könyvtárat használtam. Ez egy REST API kliens könyvtár .NET keretrendszerhez, segítségével könnyen kommunikálhatunk ASP.NET Core alkalmazásokból más REST API alkalmazásokkal. Létrehoztam egy *BarionClient* osztályt, amely implementálja az *IBarionClient* interfészt. Ez az interfész két metódust tartalmaz: *StartPayment()* és *GetPaymentState()*. Ez a két függvény hívja meg a Barion API megfelelő végpontjait a RestSharp könyvtár *RestClient* osztályának segítségével.

## Web réteg (API)

A web réteg lényege, hogy a külvilág számára egy belépési pontként szolgál az alkalmazásunk felé, így a kliensek használni tudják az alkalmazás funkcióit. Ez a réteg az én alkalmazásomban egy ASP.NET Core Web API projekt. Ebben a projektben RESTful API végpontokat publikálok a klienseknek, amelyeken keresztül kommunikálhatnak az alkalmazással: lekérdezhetnek adatokat és módosíthatják az Domain rétegben lévő entitásokat. Ezeket a végpontokat Controller osztályokban definiáltam. Emellett ez a projekt tartalmazza a globális hibakezelő middleware-t és a *Startup* osztályt is, ami az alkalmazás indulásakor szükséges konfigurációért felelős.

### Controller osztályok

A Web réteget egy nagyon vékony rétegként implementáltam, így a Controller osztályok függvényei rövidek, átlalában egy-két sorosak, és delegálnak a Szolgáltatások rétege felé. Az alkalmazás összes főbb funkciójának van egy saját Controller osztálya, így például van *AuthController*, *ProductController*, illetve *OrderController* is.

[HttpPost]

public Task<EntityCreatedViewModel> CreateOrder([FromBody] OrderUpsertDto   
 orderCreateDto)

{

return \_orderService.CreateOrderAsync(orderCreateDto);

}

A kódrészletben a rendelés leadásához tartozó függvény látható. A *[HttpPost]* attribútummal jelzem, hogy a függvénynek megfelelő végpont HTTP Post metódusú kéréseket vár, A függvény paramétere előtt a *[FromBody]* attribútum jelzi, hogy a DTO objektumot a kérés testében lévő adatokból kell előállítani. A függvényben felhívom az *IOrderService* interfész megfelelő metódusát, és átadom neki a DTO objektumot, így delegálom a kérés lekezeléséhez tartozó logikát a Szolgáltatások rétege felé.

### Globális hibakezelő middleware

Ebben a rétegben található a globális hibakezelő middleware, amit az *ErrorHandlerMiddleware* osztállyal implementáltam. Ennek az osztálynak a lényege, hogy beregisztrálom az ASP.NET Core middleware pipeline-ba, így a dobott kivételek legkésőbb itt lekezelődnek. Így a kivételek lekezelését központosítani tudtam. Mivel a különböző típusú kivételeknek különböző osztályokat definiáltam, így itt a hibakezelő osztályban egy *try-catch* segítségével a különböző típusú kivételekre megfelelő módon tudok reagálni. Így az előjött hibának megfelelő HTTP státusz kódot tudok visszaküldeni a kliensnek.

### A szerver alkalmazás konfigurálása

Az alkalmazás konfigurálása a *Startup* osztályban végeztem el. A *ConfigureServices()* függvényben az alkalmazás által használt különböző szolgáltatásokat konfiguráltam. Ilyen például a Cross-origin Resource Sharing (CORS), ahol hozzáadtam a webes kliens alkalmazást és a Barion API-t a megengedett forrásokhoz, hogy felhívhassák a szerver alkalmazásomat. Emellett az adatbázis kontextus osztályát is ebben a függvényben regisztráltam be, hogy így az ASP.NET Core keretrendszer feltudja használni a függőségek injektálásához.

A *Configure()* függvényben az alkalmazás middleware konfigurációja történik. Itt beregisztráltam a korábban említett globális hibakezelő middleware osztályomat, beállítom az alkalmazás futási környezetének megfelelően a CORS policy-t, illetve a https átirányítást is. Emellett az *Authentication* és *Authorization* middleware is itt kerül beregisztrálásra.

## Tesztelés

A szoftverek karbantarthatósága szempontjából nagyon fontos az egységtesztelés. Az egységtesztelés során mindig az alkalmazás egy kis egységét teszteljük önmagában. Az, hogy mit jelent egy egység nincs szigorúan definiálva, azonban általában egyetlen függvényt jelent. A tesztek lényege, hogy eldöntsék, hogy egy függvény pont azt csinálja-e amiért létrehoztuk.

Az alkalmazás UnitTest projektjében hoztam létre az egységteszteket az xUnit és a Moq könyvtárak segítségével. A Visual Studio 2022-be alapból be van építve egy xUnit projekt template, ennek a segítségével hoztam létre ezt a projektet. A projektben elsősorban a Szolgáltatások rétegének függvényeihez írtam teszteket, hiszen ez tartalmazza az üzleti logikát. Emellett az AutoMapper használatával létrehozott leképezések érvényességét is leteszteltem.

A teszt függvényeket a *[Fact]* attribútummal láttam el, ez jelzi az xUnit keretrendszer számára, hogy ezek teszt metódusok. A tesztek írása közben az Arrange, Act, Assert megközelítést alkalmaztam. Ennek lényege, hogy minden teszt metódus ugyanabból a három a lépésből áll: előkészítem a szükséges objektumokat, dolgokat a teszteléshez, futtatom a metódust, amit tesztelni szeretnék és végül összehasonlítom az elvárt eredményt a tényleges eredménnyel.

A Service osztályok a függőségeiket konstruktor injekcióval kapják meg, ráadásul interfészektől. Ez a két tény nagyon egyszerűvé teszi ezeknek az osztályoknak a tesztelését. A Moq könyvtár segítségével könnyen „kimockoltam”, azaz teszt objektumokkal helyettesítettem ezeket a függőségeket. A következő kódrészlet a *MaterialServiceTests* osztály egyik tesztfüggvényéből származik:

// Arrange

var materials = new List<Material>()

{

new Material() { Id = 1, Name = "Szövet" },

new Material() { Id = 2, Name = "Fém" }

};

//...

\_materialRepositoryMock.Setup(repo => repo.GetAllAsync().Result)

.Returns(materials);

A függvény első részében előkészítettem a teszt futtatásához szükséges objektumokat, a Moq könyvtár segítségével felkészítettem a mock repository osztályt, hogy a *GetAllAsync()* függvényének meghívásakor visszaadja a megfelelő objektumokat. Emellett a mock mapper osztályt is felkészítettem, hogy visszaadja a megfelelő ViewModel objektumot.

//Act

var result = await \_materialService.GetAllMaterialsAsync();

var resultList = result.Materials.ToList();

Ezután meghívtam a MaterialService típusú objektum tesztelendő metódusát, és az eredményt egy listába tettem.

//Assert

Assert.IsType<MaterialsViewModel>(result);

Assert.Equal(2, resultList.Count);

Végül összehasonlítottam a kapott eredményt az elvárt eredménnyel. A Service osztályok összes metódusához írtam unit teszteket, azokban az esetekben, ahol láttam értelmét, ott akár több tesztet is egy metódushoz.

## Telepítés

A szerver alkalmazást telepítettem egy Azure webszerverre, így nem csak fejlesztői környezetben, hanem rendes production környezetben is működik és elérhető az interneten. A telepítés menete számomra meglepően könnyű volt, nagy részben az Azure és a Visual Studio kiváló integrációjának köszönhetően. Először is az Azure felületén létrehoztam egy Web Appot és egy SQL adatbázist, ezután a Web API projektet közvetlen Visual Studio-ból telepítettem a szerverre a Publish funkció segítségével. Ezután az adatbázis sémáját frissítettem úgy, hogy bemásoltam ideiglenesen a távoli Azure-os adatbázis connection string-jét a lokális projektem *appsettings.json* fájljába, azaz lecseréltem a lokális adatbázis connection string-jét a távolira. Ezután szimplán a Package Manager Console-ban az „Update-Database” paranccsal felfrissítettem a meglévő migration fájlok alapján az adatbázist. Ez elvégezte a kezdeti adatok hozzáadását is. Ezután ezeket a migration fájlokat eltávolítottam, hogy ne kerüljenek fel a GitHub repository-ba, hiszen érzékeny adatokat is tartalmaznak (például az adminisztrátor bejelentkezési adatait) és visszaállítottam az *appsettings.json* fájlban az adatbázis connection string-jét a lokális adatbázisra. Végül a Microsoft SQL Server Management Studio szoftver segítségével ellenőriztem a távoli adatbázis sémáját és a benne lévő alap adatokat.

# Webes kliens alkalmazás

Ebben a fejezetben az Angular alapú webes kliens alkalmazást fogom ismertetni. Kitérek az architektúrára, majd a központibb, az alkalmazás több részére is kiható részeket mutatom be, ezt követően a funkciók, képernyők mentén bemutatom az implementáció részleteit, és végül írok az alkalmazás telepítéséről is.

## Az alkalmazás architektúrája

A webes kliens alkalmazásom egy Angular alapú single-page application (SPA). Az alkalmazásom egy tipikus Angular alkalmazás architektúrájával rendelkezik, ezt a függelékben található 6. ábra tartalmazza. Service osztályokban implementálom az alkalmazás üzleti logikáját és komponens, illetve template osztályokkal hoztam létre az alkalmazás nézeteit, azaz a UI réteget. Az Angular keretrendszerbe beépített függőség injektáló automatikusan szolgáltatja a Service osztályokat a komponensek számára. Az alkalmazás egy fő modullal rendelkezik, ez az *AppModule.* Ez a modul fogja összes az összes általam létrehozott komponenst. Emellett létrehoztam egy *MaterialModule* és egy *PrimengModule* osztályt, ezekbe kiszerveztem az Angular Material és a PrimeNG UI könyvtárakból behúzott komponensek importálását. Ezt a két modult beimportáltam a főmodulba, így az alkalmazásomban tudtam használni az előbb említett könyvtárak komponenseit.

A backend alkalmazás API végpontjainak megfelelően hoztam létre a Service osztályokat. Ez azt jelenti, hogy egy Service osztály mindig egy adott típusú entitáshoz kapcsolódó műveleteket lát el. Például az *OrderService* osztály a rendelésekkel kapcsolatos kérések küldéséért és fogadásáért felelős, míg az AuthService a felhasználókezelésért. Mint a legtöbb Angular alkalmazás, az enyém is rendelkezik egy *AppRoutingModule* modullal, ahol az URL-ek és az alkalmazás komponensei közötti leképezést definiálom.

## Központi részek

### HTTP kliens

Létrehoztam egy *AppHttpClient* nevű osztályt, amely az Angular beépített *HttpClient* típusát csomagolja be. Az osztály lényege, hogy leegyszerűsíti a Service osztályok számára a hálózati hívások kezdeményezését. Az osztály tartalmazza a szerver alkalmazás API-jának bázis URL-jét, és a függvények paraméterében kapott végpontot ehhez fűzik hozzá. A különböző típusú HTTP kérésekhez különböző függvényeket hoztam létre.

public get<T>(endPoint: string): Observable<T> {

return this.http.get<T>(`${this.apiUrl}/${endPoint}`);

}

A kódrészletben látható függvény a GET típusú HTTP kérések indításáért felelős, az Angular HTTP kliensének a *get()* függvényét hívja fel.

### Token interceptor

Létrehoztam egy *TokenInterceptor* nevű osztályt, amely az Angular *HttpInterceptor* interfészét implementálja. Az Interceptor osztályok lényege, hogy segítségükkel globálisan az alkalmazás szintjén az összes kimenő és bemenő HTTP kérést kezelhetjük.

const token = this.authService.getRefreshToken();

if (token)

return this.authService.refreshToken().pipe(

switchMap((token: TokenViewModel) => {

this.accessTokenSubject.next(token.accessToken);

return next.handle(this.addTokenHeader(request,   
 token.accessToken));

}),

catchError((err) => {

this.authService.logout();

this.router.navigate(['login']);

return throwError(() => new Error(err));

})

);

A *TokenInterceptor* osztályban a felhasználó hozzáférési tokenjét adom hozzá a kimenő kérések *Authorization* fejlécéhez, amennyiben a felhasználó be van jelentkezve. Ha a szerver egy 401 Unauthorized kóddal válaszol, akkor megpróbálom a refresh token segítségével frissíteni a felhasználó tokenjeit. Ha ez sikerül, akkor elmentem az új tokeneket, és újra megpróbálom elküldeni a kérést. Ha nem sikerül, akkor a felhasználót kijelentkeztetem és átnavigálom a bejelentkezési képernyőre. A *TokenInterceptor* osztályban *AuthService* osztállyal kommunikálok, tőle kérem le, mentem el, illetve frissítem a felhasználó tokenjeit.

A token frissítő logika miatt a *TokenInterceptor* osztályt nem volt kifejezetten egyszerű implementálni, ez egy rendes fejlesztői kihívásnak bizonyult számomra, sokat tanultam belőle.

### Guard osztályok

Az Angularban a guard osztályok lényege, hogy megelőzzük, hogy a felhasználó olyan részeire navigáljon az alkalmazásnak, ahova nincs jogosultsága. Az alkalmazásomban két guard osztályt hoztam létre, a *UserGuard* és az *AdminGuard* osztályokat. Az előbbivel azokat az útvonalakat védem, amelyeket egy átlag felhasználó elérhet, viszont be kell jelentkeznie hozzá, az utóbbival pedig azokat, amelyeket csak az adminisztrátor jogosultágú felhasználó érhet el. Ezek az osztályok az Angular *CanActivate* interfészét implementálják, és ennek a *canActivate()* metódusában tartalmazzák a hozzáférési logikát. Ezeket a guard osztályokat természetes az egyes útvonalakhoz beregisztráltam az *AppRoutingModule* osztályban.

const isAdmin = this.authService.isAdminValue;

if (isAdmin) {

return true;

}

this.router.navigate(['login']);

return false;

A kódrészleten az *AdminGuard* osztály *canActivate()* metódusának teste látható. Az AuthService osztály segítségével lekérem, hogy a bejelentkezett felhasználó adminisztrátor-e, ha igen, akkor *true* értéket adok vissza, ha nem akkor *false* értéket, és átnavigálom a felhasználót a bejelentkezés képernyőre.

### Modell osztályok, interfészek

A szerver alkalmazás DTO és ViewModel osztályainak megfelelő interfészeket definiáltam, így például létrehoztam a *LoginDto* és az *OrderViewModel* interfészeket. Az alkalmazásban végig ezeket az interfészeket használtam, így garantált volt, hogy ugyanazokat az adatstruktúrákat használom, amelyeket a backend alkalmazás is elfogad.

### Segéd osztályok, függvények

#### Pipe osztályok

Angularban a pipe-ok arra használhatók, hogy template kifejezésekben átalakíthatjuk velük az adatokat a megfelelő formátumra. Az Angular rendelkezik beépített pipe-okkal, azonban egyedieket is definiálhatunk a *PipeTransform* interfész implementálásával. Én három saját pipe osztályt hoztam létre. A *HufPipe* osztály a sima *number* típusú adatokat alakítja át, hogy forintértékként jelenjenek meg a felhasználói felületen:

transform(value: number): string {

let valueAsString = value.toString();

for (let i = valueAsString.length - 3; i > 0; i -= 3) {

valueAsString = valueAsString.substring(0, i) + ' ' +   
 valueAsString.substring(i);

}

return `${valueAsString} Ft`;

}

Így például a „13000” *number* típusú érték a felhasználói felületen „13 000 Ft” fog megjelenni. A *PaymentMethodPipe* és a *ShippingMethodPipe* osztályok a *PaymentMethod* és *ShippingMethod* enumerációk értékeit jelenítik meg megfelelő szöveggel a felhasználói felületen.

#### Validátor, ErrorStateMatcher

Az alkalmazásban több helyen is előfordul, hogy a felhasználónak „Jelszó”, illetve „Jelszó megerősítése” mezőket kell kitölteni. Itt a két mező értékének összehasonlításához, és a megfelelő validációs hiba kiírásához létrehoztam egy *passwordsMatchValidator* objektumot, amely az Angular *ValidatorFn* interfészét implementálja. Ennek a lényege, hogy kap egy *FormControl* objektumot és eldönti róla, hogy vissza kell-e adnia validációs hibákat.

Emellett létrehoztam egy *PasswordMatchErrorStateMatcher* osztályt, amely az Angular *ErrorStateMatcher* interfészét implementálja. Ebben az osztályban implementáltam azt a logikát, amely megadja, hogy a jelszó mező mikor mutassa a megfelelő hibaüzenetet. Úgy állítottam be, hogy a „Jelszó megerősítése” mező akkor válik érvénytelenné, amikor a „Jelszó” mezőből a felhasználó már kilépett, de a két mező értéke még nem egyezik.

### Navigációs sáv

Az alkalmazás rendelkezik a webshopokban megszokott képernyő tetején elhelyezkedő reszponzív navigációs sávval. Kisebb képernyőkön, mint például mobiltelefonokon a navigációs sáv összecsukódik, és egy hamburger menüvel lehet lenyitni azt. A navigációs sávról elérhetőek a Kezdőlap, Termékek, Kosár, illetve Profil vagy Bejelentkezés oldal, attól függően, hogy a felhasználó be van-e már jelentkezve. A navigációs sávot a *HeaderComponent* komponens osztályban valósítottam meg.

## Termékkategóriák, termékek

A webshop központi része a Bolt komponens, amelyet a *ShopComponent* osztállyal valósítottam meg. Ez tartalmazza a termékkategóriákat és termékeket megjelenítő oldalakat, a felhasználó itt tudja böngészni az elérhető termékeket, és itt tudja a kosarába tenni őket.

A Bolt komponens egyrészt tartalmaz egy úgynevezett kenyérmorzsa (breadcrumb) navigációs eszközt, amely a kiindulóponttól a felhasználó jelenlegi tartózkodási helyéig vezető utat mutatja, másrészt tartalmaz egy *<router-outlet>* kifejezést a template-ben. Ez azért, van mert a Bolt komponenst úgy alakítottam ki, hogy a Termékkategóriák, a Termékek és a Termék részletei komponenseket beágyazva tartalmazza, így egy hierarchikus navigáció alakul ki. A komponensek között a felhasználó a termékkategóriákra és termékekre kattintva, illetve a kenyérmorzsa segítségével tud navigálni.

A Termékkategóriák komponenst a *ProductCategoriesComponent* és a *ProductCategoryService* osztályok segítségével valósítottam meg. A komponens osztály inicializálásakor – az *ngOnInit()* – függvényben felhívom a Service osztály *getProductCategories()* függvényét, amely a HTTP kliens osztályomon keresztül lekéri a backendtől az elérhető termékkategóriákat és *ProductCategoriesViewModel* típusú objektumként adja vissza őket. Ezután beállítom a komponens osztály publikus *productCategories* tulajdonságának ezt a visszakapott értéket, majd a template-ben az Angular Property Binding funkciója segítségével jelenítem meg őket a felületen. Ennek a lényege, hogy a HTML kódban hivatkozni tudok a komponens osztály publikus tulajdonságaira.

<ng-container \*ngFor="let productCategory of productCategories">

<app-product-category-card [item]="productCategory"

//...

</app-product-category-card>

</ng-container>

A kódrészleten látható, hogy az *\*ngFor* strukturális direktíva segítségével végig iterálok a *productCategories* lista elemein, és az *AppProductCategoryCard* komponens *item* tulajdonságának megadom az adott termékkategória objektumot. Ez a komponens egyszerűen megjeleníti az adott termékkategóriát az Angular Material könyvtár *Card* elemének segítségével. Ez egy úgynevezett kártya elemben tartalmazza az adott termékkategória nevét, bélyegképét és leírását. A kártyára kattintva a felhasználót átnavigálom a Termékek komponensre, ahol az adott kategóriába tartozó termékeket látja. Ezt a komponens osztályba injektálható beépített *Router* objektum *navigate()* függvényével oldottam meg.

A Termékek komponenst a *ProductsComponent*, *ProductsDataSource* és a *ProductService* osztályok segítségével valósítottam meg. Ez a komponens hasonlóan az előzőhöz kártya elemekben jeleníti meg az elérhető termékeket a felhasználó számára, azonban itt szerver oldali lapozást is alkalmaztam, illetve egy termék szűrő komponenst is megjelenítek a felületen. Ennek segítségével a felhasználó leszűrheti a megjelenítendő termékeket a színük, mintájuk, anyaguk, illetve az áruk szerint. A szűrő komponenst a *ProductFiltersComponent* és a *FilterService* osztályokkal valósítottam meg.

A lapozáshoz a felhasználói felületen a PrimeNG könyvtár *Paginator* komponensét használtam. Ennek a segítségével képes a felhasználó lapozni a termék oldalak között. A *Paginator* komponens *onPageChange* eseményére rákötöttem a komponens osztályom *paginate(event: any)* függvényét, amely áthív a *loadPrdouctsPage(pageIndex: number)* függvénybe, így amikor a felhasználó lapoz egyet, ebben a függvényben betöltöm a paraméternek megfelelő oldal termékeit.

A szűrő komponens és a termékeket megjelenítő komponens közötti kommunikációra létrehoztam a *FilterService* osztályt. Ez az osztály publikus *Observable* típusú tulajdonságokat tartalmaz a felhasználó által kiválasztott szűrők megfigyeléséhez. Emellett publikus függvényeket is tartalmaz a megfigyelt értékek frissítéséhez. A *ProductFiltersComponent* komponens meghívja ezeket a függvényeket, amikor a felhasználó módosítja a felületen a szűrőket, és a *ProductsComponent* osztály értesül a megfigyelt értékek módosulásáról. Ez azt jelenti, hogy a *ProductsComponent* osztályban feliratkozok ezekre az Observable típusú objektumokra, és amikor egy új érték érkezik, akkor meghívom a *loadPrdouctsPage()* függvényt, amely a szűrők új értékeivel betölti a termékek első oldalát.

A *ProductsComponent* inicializálásakor a szűrők üresek, így egyből betöltöm a kategóriához tartozó termékek első oldalát. Az oldal méretet 6-ra állítottam, tehát egyszerre mindig maximum 6 terméket lát a felhasználó A termékek betöltésének logikája a szűrőknek és a lapozásnak köszönhetően bonyolultabb, így ezt kiszerveztem egy külön *ProductsDataSource* osztályba. Ennek az osztálynak a *loadProducts()* függvényét hívom meg a szűrőknek és a lapozásnak megfelelő paraméterekkel a *ProductsComponent* osztályból, majd innen a *ProductService* segítségével kérem le a backend alkalmazástól a termékeket. A *ProductsDataSource* a betöltött termékeket és a termékek számát *Observable* típusú objektumokként teszi megfigyelhetővé a *ProductsComponent* számára, amely ezeket a felhasználja a megfelelő termékek és oldalszámok megjelenítéséhez. Egy adott termék kártyájára kattintva átnavigálom a felhasználót a termék részleteihez.

A Termék részletei komponenst a *ProductDetailsComponent*, a *ProductService* és a *CartService* osztályokkal valósítottam meg. A komponens inicializálásakor betölti az adott terméket és megjeleníti annak nevét, leírását, árát, tulajdonságait és a képeit. A képek megjelenítéséhez a PrimeNG könyvtár *Galleria* komponensét használtam, így a felhasználó egy galériához hasonló elemen képes megtekinteni és navigálni a termék képei között. A felhasználó egy mennyiség választóval tudja megadni a mennyiségét, majd a „Kosárba” gombbal tudja hozzáadni a kosarához a terméket. A komponens osztályban ekkor felhívom a *CartService* *addItemToCart()* függvényét, amely létrehozza a megfelelő DTO objektumot és elküldi a kérést a backend felé. A felhasználót egy *SnackBar* üzenetben tájékoztatom a művelet sikerességéről.

## Felhasználókezelés

A webshop másik lényeges része a felhasználók kezelése. Amennyiben a felhasználó nincs bejelentkezve, akkor a navigációs sávon a profil ikonra kattintva a Bejelentkezés képernyőre kerül. Ezt a képernyőt a *LoginComponent* osztályban valósítottam meg, egy egyszerű bejelentkezési űrlapot tartalmaz, amelyen a felhasználó az e-mail címe és jelszója megadásával bejelentkezhet az alkalmazásba. A bejelentkezési komponensben ekkor először lekérdezem a *CartService* osztálytól, hogy jelenleg van-e már termék a felhasználó kosarában, becsomagolom a kosár azonosítóját és a megadott bejelentkezési adatokat egy DTO objektumba, majd az *AuthService* segítségével elküldöm a kérést a backend alkalmazás felé. A művelet sikerességéről egy *SnackBar* üzenetben tájékoztatom a felhasználót, majd, ha a megadott bejelentkezési adatok helyesek, akkor a felhasználót átnavigálom a kezdőoldalra.

Amennyiben a felhasználónak még nincs fiókja a webshopban, akkor a Regisztráció oldalon képes létrehozni egy újat. Ezt az oldalt a *RegistrationComponent* osztályban implementáltam, ami egy regisztrációs űrlapot tartalmaz. A komponensben az űrlaphoz többféle validációs szabályt is megadtam: jelzem a felhasználó számára, hogy melyik mezők kitöltése kötelező, hogy a „Jelszó” és a „Jelszó megerősítése” mezők értéke egyezik-e, illetve, hogy a megadott e-mail cím megfelelő formátumban van-e. A regisztrációhoz az AuthService *register()* függvényét hívom meg, majd *SnackBar* üzenetben tájékoztatom a felhasználót a művelet sikerességéről.

Ha a felhasználó be van jelentkezve, akkor a navigációs sávon a profil ikonra kattintva egy legördülő menüből válaszhat a következő opciók közül: „Profil”, „Rendeléseim”, „Kijelentkezés”. A Profil oldalt a *ProfileComponent,* a *UserService* és az *AuthService* osztályokkal implementáltam. A komponensben egy űrlapot jelenítek meg, amelyet a komponens indulásakor a *UserService* *getUserData()* függvényének segítségével kitöltök a felhasználó adataival Az e-mail cím kivételével az űrlap mezőit módosíthatja a felhasználó, majd a „Módosítások mentése” gombbal adhatja le a kívánt módosításokat. Ekkor az *AuthService* *updateUser()* függvényét hívom meg az űrlap mezőinek értékeiből létrehozott DTO objektummal. Emellett a komponens tartalmaz egy jelszó módosítási űrlapot is. Ezen a felhasználónak meg kell adnia a régi és az új jelszavát, valamint meg kell erősítenie az újat. A jelszó változtatásához az AuthService *changePassword()* függvényét hívom meg.

A Rendeléseim oldalon a felhasználó a korábban leadott rendeléseit tekintheti meg. Ezt az oldalt a *MyOrdersComponent* osztályban implementáltam. A *UserService* *getUserOrders()* függvényének segítségével lekérem a backend alkalmazástól a felhasználóhoz tartozó rendeléseket, majd ezeket kilistázva megjelenítem a felületen. A felhasználó megtekintheti a rendelés idejét, összegét, azonosítóját, valamint a termékeket, amelyeket megrendelt.

Az *AuthService* osztály nem csak a felhasználókezeléshez kapcsolódó műveletek elvégzéséért felelős, hanem rajta keresztül érheti el az alkalmazás többi része az aktuális felhasználóhoz tartozó információkat. Egy *Observable* típusban tárolom az épp aktuálisan bejelentkezett felhasználót, és erre feliratkozva tud reagálni az alkalmazás többi része az változásokra. Emellett az *AuthService* osztályban a böngésző lokális tárhelyén - a *localStorage* objektum segítségével - eltárolom a böngészőben a felhasználó adatait és hozzáférési tokenjeit, így, ha bezárja, majd később újra megnyitja az alkalmazást, nem lesz szüksége újra bejelentkezni.

private setCurrentUser(userViewModel: UserViewModel): void {

localStorage.setItem('currentUser', JSON.stringify(userViewModel));

}

A fenti kódrészleten látható, ahogy a backend alkalmazástól kapott *UserViewModel* típusú objektumot átalakítom JSON adatformátumra és elmentem a böngésző lokális tárhelyére a „currentUser” kulccsal.

public getCurrentUser(): UserViewModel | null {

const currentUser = localStorage.getItem('currentUser');

if (currentUser === null) {

return null;

}

return JSON.parse(currentUser);

}

A fenti kódrészleten látható, hogy a lokális tárhelyből lekérem a „currentUser” kulccsal a korábban elmentett JSON formátumú adatot, majd ezt a *JSON.parse()* függvény segítségével átalakítom *UserViewModel* típusúvá. A fenti függvényeket az *AuthService* osztályban használom a felhasználó bejelentkezésénél, illetve az alkalmazás indulásánál.

## Kosár, rendelés

A felhasználó a kosarában lévő termékeket a Kosár oldalon tudja megtekinteni és módosítani. Ezt a *CartComponent* osztállyal valósítottam meg, ahol az Angular Material könyvtár *Table* komponense segítségével táblázatos formában jelenítem meg a termékeket. A táblázatban megjelenítem a termékek bélyegképét, nevét, árát, mennyiségét, és a részösszeget, valamint az utolsó oszlopban lévő gombbal a felhasználó eltávolíthatja az adott terméket a kosarából. A táblázat láblécében a kosárban lévő termékek árainak összegét láthatja a felhasználó, illetve a táblázat alatt lévő gombokkal a kosarát tudja kiüríteni, illetve tovább léphet a rendeléshez. Egy képernyőképet a kosár komponensről a függelékben a 7. ábra tartalmaz.

A kosár komponensben a *CartService* osztály függvényeit hívom meg a kosárral kapcsolatos műveletek elvégzéséhez, illetve az ebben az osztályban található *Observable* típusú *currentCart$* objektumot figyelem meg, mivel ez tartalmazza a felhasználó kosarának aktuális tartalmát. A *CartService* osztályban az összes kosárral kapcsolatos művelethez írtam egy megfelelő publikus függvényt, és emellett megfigyelem az *AuthService* segítségével a felhasználó aktuális értékét. A felhasználó változását – alkalmazás indulásakor, bejelentkezéskor, kijelentkezéskor – a *handleUserChanged(user: UserViewModel | null)* privát függvényben kezelem le, amely megkapja paraméterként a felhasználó új értékét. Ennek a függvénynek a logikája bonyolultabb, implementálása komolyabb fejlesztői kihívás volt számomra, és most tömören ismertetem ezt.

Először is a *CartService* osztályban a böngésző helyi tárhelyében eltárolom az aktuális kosár azonosítóját, illetve azt, hogy a kosár egy bejelentkezett felhasználóhoz tartozik-e. A függvény elején lekérem a helyi tárhelyből ezt az objektumot, és amennyiben a felhasználó új értéke nem *null*, az azt jelenti, hogy vagy épp egy bejelentkezés történt, vagy az alkalmazás indulásánál vagyunk, és van elmentett felhasználó a lokális tárhelyben, tehát be van jelentkezve.

if (user !== null) {

return this.getCartByIdAndSetAsUserCart(user.cartId);

}

Ekkor egyszerűen lekérem a backend alkalmazástól a felhasználó kosarát, és elmentem a lokális tárhelyen. Ha *null* értéket kaptam és nincs elmentve kosár a helyi tárhelyen, az azt jelenti, hogy alkalmazás indulásnál vagyok, viszont nincs korábbi felhasználó elmentve, ilyenkor nincs teendőm. Ha azonban *null* értéket kapok és van elmentve kosár a helyi tárhelyen, akkor két további lehetőség van. Amennyiben a kosár nem tartozik felhasználóhoz, akkor ez egy nem bejelentkezett felhasználó korábbról elmentett kosara, így ezt lekérem a backend alkalmazástól a kosarat az azonosítója alapján és a megfigyelhető objektum értékét erre állítom:

if (localCart.isAnonymous) {

return this.getCartById(localCart.id).pipe(

tap(cart => {

this.currentCartSubject.next(cart);

})

);

} else {

Ha azonban a kosár felhasználóhoz tartozik, az azt jelenti, hogy egy felhasználó épp kijelentkezett, így eltávolítom az ő kosarát a helyi tárhelyből és az aktuális megfigyelhető kosár értékét *null* értékre állítom:

return new Observable(() => {

this.removeCurrentLocalCart();

this.currentCartSubject.next(null);

});

}

A felhasználó a Kosár oldalról átnavigálhat a Rendelés oldalra, ahol megadhatja a rendelés adatait és leadhatja a rendelését. Ezt az oldalt az *OrderComponent* és az *OrderService* osztályokkal valósítottam meg. Az *OrderComponent* osztályban az Angular Material könyvtár *Stepper* komponensével egy három lépésből álló rendelési folyamatot hoztam létre. Az első lépésben a személyes adatokat, a második lépésben a szállítás módját, a harmadik lépésben a fizetés módját kell megadnia a felhasználónak. Amennyiben a felhasználó be van jelentkezve, akkor a személyes adatok mezőit előre kitöltöm az *AuthService* osztály segítségével. Természetesen a rendelési űrlapon is felállítottam a megfelelő validációs szabályokat, így például a kötelező mezőket nem hagyhatja üresen a felhasználó. A felhasználó innen tovább navigálhat a Rendelés összegző oldalra, ahol a megadott rendelési adatait láthatja. Innen vissza tud navigálni a rendelési adatok megadásához, illetve le tudja adni a rendelését.

Amennyiben a felhasználó leadja a rendelését, akkor felhívom az OrderService megfelelő metódusát, attól függően, hogy webes fizetést választott-e vagy nem. Ha nem webes fizetést választott, akkor egyszerűen a *placeOrder()* függvényt hívom meg, és tájékoztatom a rendelés leadásának sikerességéről egy üzenetben, valamint sikeres rendelés esetén átnavigálom őt a Rendelés leadva oldalra. Ha webes fizetést választott, akkor a *placeWebPaymentOrder()* metódust hívom fel, amely elküldi a webes fizetési kérést a backend felé. Ezután a backend válaszából kinyert URL alapján átnavigálom a felhasználót a Barion felületére, ahol elvégezheti a webes fizetést. Ha ott sikeresen végzett a fizetéssel, akkor a Rendelés leadva oldalra lesz visszairányítva. Ezen az oldalon láthatja a leadott rendelése azonosítóját.

## Adminisztrátori felület

Amennyiben az adminisztrátor bejelentkezik az ő fiókjával a webes kliensbe, az alkalmazás további funkciókkal bővül ki, amelyek segítségével elláthatja az adminisztrátori feladatait.

Az első ilyen extra funkció a termékkategóriák és termékek hozzáadása, törlése, szerkesztése. A termékkategóriákat és termékeket megjelenítő komponenseken a kártya elemek mellett megjelenítek egy „+” gombot, amelyre kattintva egy felugró ablakban adhat hozzá egy új termékkategóriát, illetve terméket az adminisztrátor. Emellett a kártyák alatt megjelennek a „Szerkesztés” és „Törlés” gombok, előbbire kattintva az adott egy előugró ablakban szerkesztheti, utóbbira kattintva kitörölheti az adott termékkategóriát, illetve terméket az adminisztrátor. A *ProductCategoriesComponent* és *ProductsComponent* osztályokban az *AuthService* osztály *isAdmin* megfigyelhető tulajdonsága segítségével döntöm el, hogy az adott felhasználó adminisztrátor-e, és ennek megfelelően jelenítem meg a különböző felületeket.

A második extra funkció a színek, minták, anyagok hozzáadása és törlése a rendszerből. Az adminisztrátor ezt az oldalt a navigációs sávon megjelenő új „Színek, minták, anyagok” menüpontról éri el. Ezt az oldalt az *AdminColorsPatternsMaterials* komponens és a *ColorService*, *PatternService* és *MaterialService* osztályokkal valósítottam meg.

A harmadik extra funkció a leadott rendelések megtekintése. Ezt az adminisztrátor navigációs sáv „Rendelések” menüpontjáról éri el. Ezt az oldalt az *AdminOrdersComponent* osztállyal hoztam létre, itt táblázatos, lapozható formában jelenítem meg a rendeléseket az Angular Material könyvtár *Table* és *Paginator* komponenseivel. A táblázatban a rendelések dátumát, azonosítóját, a rendelő e-mail címét és a rendelés összegét jelenítem meg. Az adminisztrátor rá is tud szűrni a megjelenített rendelésekre, egyrészt rá tud keresni egyadott rendelésre annak azonosítójával, másrészt beállíthat egy időintervallumot, így csak azok a rendelések jelennek meg, amelyek leadásának időpontja beleesik az adott intervallumba. Az időintervallum megadásához az Angular Material könyvtár *Datepicker* komponensét használtam. A táblázat fejlécében a Dátum oszlopra kattintva az adminisztrátor válthat időben növekvő, illetve csökkenő rendezések között. Emellett a táblázat alatt található lapozó komponens segítségével tud léphet az oldalak között, valamint itt adhatja meg a oldalanként megjelenítendő rendelések számát egy legördülő menüvel.

A rendelések betöltésének logikáját kiszerveztem egy *OrdersDataSource* osztályba, amely az Angular Material *DataSource* absztrakt ősosztályából származik le. Ez az ősosztály a táblázatok adatforrásának implementálásához lett létrehozva. A *connect()* és *disconnect()* metódusokat tartalmazza, ezekben adtam meg, hogy hogyan csatlakozzon a táblázat az adatforrás osztályhoz. A *connect()* metódusban visszaadom a rendeléseket tartalmazó *Observable* típusú objektumot, így a táblázat ezt az objektumot figyeli meg és reagál az új értékeire, azaz megjeleníti a rendeléseket. Az osztályban emellett létrehoztam egy publikus *loadOrders()* függvényt, amely rendelkezik a rendelési azonosítónak, rendezési iránynak, oldalszámnak, oldalméretnek és időintervallumnak megfelelő paraméterekkel. Ezek alapján a függvényben összeállítom a megfelelő DTO objektumot és felhívom az *OrderService* osztály *getOrderPage()* függvényét, amely elküldi a kérést a backend alkalmazás felé és visszaadja a választ *PagedOrdersViewModel* típusú objektumban.

loadOrdersPage(): void {

this.dataSource.loadOrders(

this.searchWord,

this.sort.direction,

this.paginator.pageIndex,

this.paginator.pageSize,

this.range.value.start,

this.range.value.end

);

}

A fenti kódrészletből látható, ahogyan a komponens osztályban felhívom az adatforrás osztály *loadOrders()* függvényét, és a paramétereket beállítom a felületen lévő elemek értékei alapján.

## Telepítés

Angularban a környezetfüggő változókat az „environment.ts” és „environment.prod.ts” fájlokban lehet megadni. Fejlesztés közben az előbbi fájlban megadott változók lesznek érvényesek míg a kiadási környezetben az utóbbiban megadottak. Ezekben a fájlokban adtam meg a backend alkalmazás API URL-jét, mivel ez fejlesztési és kiadási környezetben különböző.

A webes klienst szintén egy Azure webszerverre telepítettem, csakúgy, mint a backend alkalmazást. Először létrehoztam az új webszervert az Azure portálon, amely nem Windows, hanem Linux környezetben fut - hiszen itt nincs szükség semmilyen .NET-es függőségre, csak a Node.js keretrendszerre. Ezután az Angular CLI „ng build” parancsának segítségével létrehoztam egy kiadásra kész változatot az alkalmazásból. A Visual Studio Code szoftverben telepítettem az Azure App Service kiegészítőt, amelyet összekötöttem az Azure portálos fiókommal, így ennek segítségével néhány kattintással ki tudtam adni a webszerverre az alkalmazásomat. Egy indítási parancs segítségével elértem, hogy a webszerver tudja, hogy melyik alkönyvtárból szolgálja ki az alkalmazást, valamint azt, hogy egy single-page alkalmazást kell kiszolgálnia, így átadja az útvonal leképezést az Angular keretrendszer számára. A webes kliens alkalmazás elérhető a következő útvonalon: https://curlycircle.azurewebsites.net.

# Mobil kliens alkalmazás

Ebben a fejezetben bemutatom az Android platformra készített mobil kliens alkalmazásomat. Ez az alkalmazás felhasználói szempontból nagyban hasonlít a webes kliens alkalmazásra, ugyanazok az általános webshop funkciók érhetőek el innen is az átlag felhasználó számára. Ez az alkalmazás azonban nem tartalmaz adminisztrátori felületet, mivel erre a követelmények szerint nem volt szükség.

## Az alkalmazás architektúrája

Mint ahogy azt korábban említettem, az Android alkalmazás fejlesztése közben próbáltam minél jobban követni a Google ajánlásait és iránymutatását, így ennek megfelelően alakítottam ki az alkalmazás architektúráját is. Az architektúra alapvetően két rétegből áll: az adat rétegből, és a felhasználói felület rétegéből – továbbiakban UI réteg.

A UI réteg két fő feladata, hogy az alkalmazás állapotában, azaz az adataiban bekövetkező változásokat átalakítsa egy olyan formába, amit a felhasználói felületen prezentálni tudunk, majd ezeket megjelenítse.

Ezek alapján beszélhetünk a UI állapotáról és a UI elemekről. Ezek együtt alkotják magát a UI réteget. Az Android alkalmazásokban a UI állapotának megváltozása történhet felhasználói interakció, vagy az adatok megváltozásának hatására. Ezeknek a lekezelésére és transzformációjára szükség van valamilyen közvetítő logikára. Ezt a logikát implementálhatnánk magában a UI-ban, a UI elemek mellett, azonban így a UI-nak egyre több felelőssége lenne: ő lenne az adat tulajdonosa, előállítója, átalakítója és a többi. Ennek a megoldására ajánlja a Google a Unidirectional Data Flow (UDF) architekturális mintát, amely segít kikényszeríteni, hogy itt a felelősségi körök megfelelően el legyenek választva.

Az UDF minta szerint az úgynevezett state holder, azaz állapot tároló osztályok felelőssége a UI állapotának előállítása és az előbb említett logika implementálása. Az alkalmazásomban ezek az állapot tároló osztályok a ViewModel osztályok lesznek. A ViewModel egy olyan osztály, amely eléri az adat réteget, így azzal kommunikál, és emellett automatikusan túléli a konfiguráció változásokat, így a benne tárolt adat nem vész el például a képernyő elforgatásakor. A ViewModel osztályok definiálják az események kezeléséhez szükséges logikát, és ennek eredményeképp ők hozzák létre az új UI állapotot. Fontos, hogy nem közvetlen módosítják az állapotot, hanem az események és az előző állapot alapján létrehoznak egy új állapotot.

Az UDF architekturális minta szerint a UI állapota mindig lefelé – a UI elemek felé- folyik, míg az események felfelé – a ViewModel felé – folynak, ezt szemlélteti a függelékben található 8. ábra is. Ennek a legfőbb előnye, hogy egyetlen adatforrás van a UI szempontjából, így könnyebb elkerülni a hibákat, és a UI mindig garantáltan konzisztens állapotban marad az adatokkal.

Míg a UI réteg a felhasználói felülethez kapcsolódó állapotot és a UI logikát, addig az adat réteg az alkalmazás adatait és az üzleti logikát tartalmazza. Az üzleti logika teremti az alkalmazás igazi értékét: üzleti szabályok alkotják, amelyek leírják, hogy milyen módon hozzuk létre, tároljuk és változtatjuk meg az alkalmazás adatait. A legfőbb előnye annak, hogy az üzleti logikát leválasztjuk egy külön rétegbe, az az, hogy így könnyen újra használhatjuk a réteg osztályait több képernyőn is, az alkalmazás különböző részei között egyszerűbb megosztani az információkat, illetve az üzleti logikát elő tudjuk állítani a UI rétegtől függetlenül, például teszteléshez.

Az adat réteg úgynevezett repository és adatforrás osztályokból áll. Minden különböző adat típushoz tartozik pontosan egy repository osztály. Például az én alkalmazásomban a termékekhez tartozó osztály a *ProductsRepository*, míg a kosár adatokkal foglalkozó osztály a *CartRepository*. A repository osztályok elérhetővé teszik az adatokat az alkalmazás többi részének számára, kezelik az adatokban végrehajtandó változtatásokat, az adatforrások közti felmerülő konfliktusokat feloldják és ők maguk tartalmazzák az üzleti logikát

Az adatforrást reprezentáló osztályok mindegyike csak egy-egy adott típusú adatforrással dolgozhat. Például az alkalmazásomban létezik külön egy *ProductsRemoteDataSource* osztály, amely a szerveralkalmazástól szerzi be az adatokat, és létezik egy *ProductsLocalDataSource* osztály, amely a lokális adatbázisból teszi azt. Az adatforrás osztályok gyakorlatilag egy hídként szolgálnak az alkalmazás és az adatokat szolgáltató rendszerek között. A UI réteg soha nem érheti el az adatforrásokat közvetlenül, a belépési pont az adat rétegbe mindig a *repository* osztályok kell, hogy legyenek. Ennek a lényege, hogy így az architektúra különböző rétegei egymástól függetlenül skálázhatóak.

Az adat réteg architektúrájának tervezésénél az is egy fontos szempont volt, hogy egy offline-first alkalmazást akartam készíteni, mivel a Google is ezt ajánlja Androidon. Az offline-first alkalmazások lényege, hogy internet kapcsolat nélkül is képes az összes funkcióját, vagy a funkcióinak egy bizonyos részét elvégezni. Ez azt jelenti, hogy az üzleti logikának egy része képes offline módon működni.

A mobil eszközökre különösen igaz az, hogy az internet kapcsolat nem mindig garantált, lassú, vagy bizonytalan is lehet. Ez nagy mértékben befolyásolni tudja az alkalmazás felhasználói élményét, éppen ezért tartottam fontosnak, hogy offline-first módon tervezzem meg az alkalmazásomat. Ez a tervezés az adat rétegben indul, hiszen ez a réteg látja el az alkalmazást adatokkal, és itt van implementálva az üzleti logika is.

Az alkalmazásokban az adaton végzett műveletek alapvetően két csoportra oszthatók: olvasási műveletek, ahol valamilyen adatot kell lekérnünk és általában megjelenítenünk a felhasználó számára, illetve írási műveletek, ahol a felhasználó által generált bemenetet kell kezelnünk – ilyen például, hogy berak egy terméket a kosarába.

Egy offline-first alkalmazás legalább az olvasási műveleteket el kell, hogy tudja végezni internet kapcsolat nélkül. Mint korábban említettem, a *repository* osztályok felelőssége az adatok szolgáltatása az alkalmazás számára az adatforrások kombinálásával. Egy offline-first alkalmazásban legalább egy olyan adatforrásra szükség van *repository* osztályonként, amelyik képes internet kapcsolat nélkül elvégezni a kritikus feladatait. Egy ilyen kritikus feladat lehet például az adat kiolvasása. Az én alkalmazásomban az offline adatforrás, ahonnan az adatok kiolvasásra kerülnek egy lokális Room adatbázis. Ez teszi lehetővé, hogy például internet kapcsolat nélkül is lehessen böngészni a termékeket, vagy a felhasználó kosarát.

Az írási műveleteknél több lehetőségünk is van egy offline-first alkalmazásban, több stratégia is létezik az offline-first módon való kezelésükre. A Google által ajánlott három az online-only, a queued writes, illetve a lazy writes. Az utóbbi kettő komplexebb megoldások, komolyabb implementációt igényelnek, nagyobb, vagy kritikus funkcionalitással rendelkező alkalmazásokban van értelme használni őket. Ezeknek az implementálása nem fért volna bele a diplomamunkám keretébe, így én az online-only stratégiát választottam.

Amikor egy offline-first alkalmazásban újra elérhető lesz az internet kapcsolat, szinkronizálnunk kell a lokális adatforrást a távoli (szerveralkalmazás) adatforrással. Erre három fő stratégia létezik: a pull-based és a push-based szinkronizáció, illetve ezeket kombinálva a hibrid szinkronizáció.

A pull-based szinkronizáció lényege, hogy az alkalmazás gyakorlatilag on-demand módon kéri le a friss adatokat a távoli adatforrástól. Erre egy gyakran használt megoldás a navigáció alapú adatfrissítés, amely úgy működik, hogy amikor a felhasználó egy adott képernyőre navigál, akkor az alkalmazás azokat az adatokat kéri le, amelyikeket az adott képernyőn kell megjeleníteni. Ez egy opportunista megközelítés, akkor működik legjobban, amikor rövid, vagy közepes hosszúságú internet kapcsolat nélküli periódusokra számíthatunk. Hosszabb periódusoknál egyre nagyobb az esélye annak, hogy a felhasználó egy olyan képernyőre navigál, amelyikhez tartozó adatok nem frissek, vagy nincsenek is meg a lokális adatforrásban. A pull-based szinkronizáció előnyei, hogy viszonylag könnyű implementálni, és olyan adatok, amikre nincs szükség sosem lesznek lekérve. Hátránya, hogy hajlamos magas hálózati forgalmat generálni, mivel, ha a felhasználó gyakran navigál vissza ugyanarra a képernyőre, akkor meg nem változott adatokat kell „frissítenünk” a távoli adatforrásból.

A push-based szinkronizációnál a lokális adatforrás megpróbálja a képességeinek megfelelően utánozni, követni a távoli adatforrást. Ezzel a megközelítéssel az alkalmazás sokkal kevésbé függ a távoli adatforrástól, így hosszabb internet kapcsolat nélküli periódusokban is használható marad az alkalmazás. Hátránya ennek a megközelítésnek, hogy az adatforrások közti konfliktusok feloldása nem triviális, komplexebb implementációt igényel, illetve a szerveralkalmazásnak támogatnia kell a szinkronizációhoz szükséges értesítések küldését.

A push-based szinkronizáció hátrányai azt jelentik, hogy a szerver oldali alkalmazás és a mobil kliensalkalmazás esetében is komplexebb, komolyabb implementációra lenne szükség, azonban erre nem volt a diplomamunkám időkeretében lehetőségem. Így a pull-based szinkronizációt választottam navigáció alapú megközelítéssel. Ez a megközelítés implementáció szempontjából egyszerűbb, azonban így is elég komplex ahhoz, hogy egy megfelelő mértékű fejlesztői munkát igényeljen, és jelentősen javítja a felhasználói élményt a nem offline-first alkalmazásokhoz képest.

## Központi elemek

Ebben a fejezetben az implementációnak olyan részeiről fogok írni, amelyek vagy központi szerepet játszanak, vagy az alkalmazás több rétegére is kiterjednek, vagy szimplán nem sorolhatóak be a későbbi, képernyők szerinti csoportosításba.

### Függőségek injektálása

A függőségek injektálásához Androidon a Google a Hilt könyvtárat ajánlja, ez a Dagger keretrendszerre épül, annak a használatát könnyíti meg Androidos alkalmazások készítésekor. A Hilt lényege, hogy automatizálja a függőségek injektálását, minden Android osztályhoz szolgáltat nekünk konténereket és kezeli ezeknek az életciklusát. A Hilt könyvtárat elsősorban annotációkon keresztül tudjuk használni, így tájékoztatjuk a keretrendszert arról, hogy hogyan szolgáltassa a függőségeket.

Az első lépése a Hilt használatának, hogy létrehoztam egy *Application* osztályt, amelyet elláttam a *@HiltAndroidApp* annotációval. Ez működésbe hozza a Hilt kódgenerálását, és létrehoz egy ősosztályt, amely egy alkalmazás szintű függőség tároló konténerként szolgál. Ez gyakorlatilag az alkalmazás szülő komponense, és ez azt jelenti, hogy az összes többi komponens eléri a függőségeket, amelyeket ő szolgáltat.

Ahhoz, hogy a Hilt szolgáltatni tudja az osztályainkat, típusainkat függőségként, információra van szüksége arról, hogy hogyan lehet az adott osztályt, típust példányosítani. Többféle mód is van arra, hogy ezt az információt hogyan adjuk meg a Hilt számára. Az első és legegyszerűbb mód a konstruktor injektálás. Ilyenkor egy osztály konstruktora elé az *@Inject* annotációt illesztem be, és így a Hilt tudni fogja, hogy a konstruktorban szereplő paraméterek az adott osztály függőségei. Az alkalmazásomban az összes ViewModel osztály szolgáltatásához konstruktor injekciót alkalmaztam. A ViewModel osztályokat a felhasználói felületet leíró composable függvényekben használom, itt azonban nem kézzel példányosítom őket, hiszen konfiguráció változásoknál az adatvesztés elkerülése érdekében nem szabad új példányt létrehozni belőlük, hanem a meglévőt kell tovább használni. Pont erre jó a beépített *hiltViewModel()* függvény, amely vagy létrehoz egy új példányt, vagy, ha már létezik egy korábbi, akkor visszaadja azt. Ennek a funkcionalitásnak a használatához el kellett látnom a ViewModel osztályaimat a *@HiltViewModel* annotációval.

Vannak olyan esetek, amikor egy adott típust nem lehet konstruktor injektálni. Ennek több oka is lehet, például az, hogy a típus maga egy interfész, nem pedig egy osztály, vagy akár az is, hogy a típus egy külső könyvtárból származik, így nem én írom meg a konstruktorát. Ezekben az esetekben az úgynevezett Hilt modulokat használtam, amelyek olyan osztályok, melyek a *@Module* annotációval vannak ellátva, és információt adnak a Hilt könyvtárnak arról, hogy hogyan tud példányokat szolgáltatni egy adott típusból. Emellett az *@InstallIn* annotációval mondjuk meg, hogy a modul milyen Android osztályban lesz használva. A modulon belül szolgáltatott függőségek az összes komponensben elérhetőek lesznek, amelyek az adott Android osztályhoz kapcsolódnak. Az alkalmazásomban az *@InstallIn* annotáció paramétere minden függőségnél *SingletonCompononent::class*, mivel ez a Hilt komponens tartozik az *Application* Android osztályhoz, amit már korábban is említettem. Az ebben a komponensben lévő függőségek így az egész alkalmazásban injektálhatóak, életciklusuk megegyezik az alkalmazáséval, és mindig maximum egy példány létezhet belőlük.

Egy Hilt modulon belül úgy adtam meg egy függőséget, hogy létrehoztam egy *@Provides* annotációval ellátott függvényt. Ez a függvény visszatérési értéke megadja, hogy milyen típusnak adom vissza a példányát, a függvény paraméterei megadják, hogy az adott típusnak milyen függőségekre van szüksége és függvény teste megadja, hogy hogyan kell létrehozni az adott típus egy példányát. A Hilt végre fogja hajtani a függvény testét minden alkalommal, mikor szolgáltatnia kell az adott típusból egy új példányt.

Erre egy jó szemléltető példa az alkalmazásomból a hálózati kommunikációhoz használt Retrofit API interfész létrehozása:

@Module  
@InstallIn(SingletonComponent::class)  
object NetworkModule {

@Provides  
@Singleton  
fun provideCurlyCircleApi(  
 okHttpClient: OkHttpClient  
): CurlyCircleApi {  
 return Retrofit.Builder()  
 .baseUrl(CurlyCircleApi.BASE\_URL)  
 .client(okHttpClient)

//...  
 .build()  
 .create(CurlyCircleApi::class.*java*)  
}

}

Itt a függvény visszatérési értéke megadja a Hilt számára, hogy a *CurlyCircleApi* típushoz fogjuk megadni a példányosításának módját. A függvény *OkHttpClient* típusú paramétere megmondja, hogy szüksége van a példányosításhoz egy ilyen típusú példányra, azaz függ tőle. A függvény testében pedig pontosan megadom, hogy hogyan kell létrehozni a *CurlyCircleApi* típus példányát. Látszik, hogy a függvény egy *NetworkModule* nevű Hilt modul része, és az *@InstallIn* annotáció paramétere megadja, hogy az ebben a modulból szolgáltatott függőségek Singleton objektumok legyenek, azaz az alkalmazás szintjén mindig pontosan egy darab példány létezzen.

Előfordulhat, hogy egy adott típusú függőségből az alkalmazás különböző részein különböző implementációkat kell szolgáltatni. Ezt az úgynevezett qualifierek segítségével tehetjük meg. A qualifierek olyan annotációk, amelyek megadják, hogy melyik implementációt szolgáltassuk egy adott típushoz az adott helyen. Erre az alkalmazásomban az adatforrás osztályok esetén volt szükség, mivel egy adott típusú adatforrás interfészhez tartozik egy lokális és egy távoli adatforrás implementáció is:

@Qualifier  
@Retention(AnnotationRetention.RUNTIME)  
annotation class RemoteDataSource  
  
@Qualifier  
@Retention(AnnotationRetention.RUNTIME)  
annotation class LocalDataSource

Ezután a *DataSourceModule* nevű Hilt modulban megadtam, hogy melyik annotációhoz melyik implementáció tartozik:

@Module  
@InstallIn(SingletonComponent::class)  
object DataSourceModule {

@Singleton  
@RemoteDataSource  
@Provides  
fun provideRemoteProductCategoriesDataSource(  
 api: CurlyCircleApi): ProductCategoriesDataSource {  
 return ProductCategoriesRemoteDataSource(api)  
}  
  
@Singleton  
@LocalDataSource  
@Provides  
fun provideLocalProductCategoriesDataSource(  
 database: CurlyCircleDatabase): ProductCategoriesDataSource {  
 return ProductCategoriesLocalDataSource(database.productCategoriesDao())  
}

}

A kódrészletben látható, hogy a *@RemoteDataSource* annotációval ellátott adatforrás osztály a Retrofit interfésztől, míg a @*LocalDataSource* annotációval ellátott az adatbázistól. A függvények teste is különbözik, azonban a visszatérési értékük megegyezik, hiszen mindkettő ugyanannak a típusnak egy-egy különböző példányát szolgáltatja.

Az alkalmazásomban öt különböző Hilt modult definiáltam, mindegyikük más típusú osztályok példányait szolgáltatják, azonban mindegyik a *SingletonComponent* komponenshez tartozik. A *NetworkModule* a hálózati kommunikációhoz szükséges osztályok példányait, a *DatabaseModule* az adatbázis osztály példányát, a *DataSourceModule* az adatforrás osztályok példányait, a *RepositoryModule* a repository osztályok példányait szolgáltatja, míg a *CoroutinesModule* különböző Kotlin specifikus függőségeket szolgáltat.

Természetesen az adatforrás és a repository osztályok mind rendelkeznek interfészekkel, és az őket felhasználó osztályok ezektől az interfészektől függenek, így az alkalmazás rétegei nem függenek egymás implementációitól, tehát lazán csatolt az architektúra. Ennek előnye, hogy könnyebb módosítani, lecserélni, karbantartani az egyes rétegeket.

### Navigáció

A felhasználói felület megvalósításához az új Jetpack Compose könyvtárat használtam. Ennek lényege, hogy úgynevezett composable függvények segítségével írjuk le a felhasználói felületet Kotlin nyelven. Általánosságban igaz, hogy az alkalmazás összes képernyőjéhez tartozik egy-egy composable függvény, ami az adott képernyőt leírja. Ezek a függvények kisebb, elemibb composable függvényekből épülnek fel, pont úgy, ahogy a képernyő is kisebb elemekből épül fel.

A Jetpack könyvtár része a Navigation Component és ez támogatja a Jetpack Compose alkalmazásokat is, így ennek a segítségével valósítottam meg a képernyők közötti navigációt. A Navigation Component központi része a *NavController* osztály. Ez az osztály nyilvántartja a back stacket alkotó képernyőket, és elmenti ezek állapotát. Navigálás közben az egymás után megnyitott képernyők a back stacken tárolódnak el, ennek lényege, hogy ha a felhasználó visszafele akar navigálni, például megnyomja a beépített vissza gombot, akkor a back stackről csupán le kell szedni a legfelső képernyőt – ezt hívják pop műveletnek.

Egy *NavControllerhez* pontosan egy *NavHost* példánynak kell tartoznia. A *NavHost* összekapcsolja a *NavControllert* egy navigációs gráffal, ami megadja a composable célállomásokat, azaz képernyőket, ahova navigálni tud a felhasználó. Ahogy a felhasználó navigál ezek között, a *NavHost* tartalma automatikusan újra rajzolódik a képernyőre. Mindegyik célállomáshoz a gráfban tartozik egy *String* típusú route, és ennek egyedi értéke meghatározza az útvonalat, ami a célállomáshoz vezet. A route tartalmazhat paramétereket is, amennyiben navigálás közben erre szükség van.

Az alkalmazás felhasználói felületét úgy alakítottam ki, hogy a képernyő alján egy navigáció sávon három gomb segítségével elérhető a három fő célállomás: a Bolt, a Kosár és a Profil. Egy célállomáson belül több képernyő is szerepel, például a Bolton belül van Termékkategóriák, Termékek és Termék részletek képernyő is. A célállomásokon belül beágyazott navigációt valósítottam meg a *navigation()* composable függvény segítségével. Ennek lényege, hogy a navigációs gráfon belül nem csak képernyő lehet célállomás, hanem egy másik, beágyazott navigációs gráf is. Így sikerült azt megoldani, hogy ha például a Bolt célállomáson belül a felhasználó elnavigál a Termék részletei oldalra, utána kiválasztja a Kosár célállomást, majd visszanavigál a Boltra, akkor a Bolton belül ugyanaz a Termék részletei oldal fogadja. Így a felhasználó nem veszíti el a „munkáját” amikor átnavigál egy másik célállomásra, ezzel növeltem az alkalmazás felhasználói élményét.

A navigációt kisegítő osztályok és függvények a *CurlyCircleNavigation* fájlban találhatóak. Itt definiáltam az összes képernyő route-ját, argumentumokkal együtt, itt található a *CurlyCircleNavigationActions* osztály, amely a lehetséges navigációs műveleteket tartalmazza. Egy ilyen művelet általában egy adott képernyőre navigálás. Emellett itt hoztam létre a *BottomNavScreen* osztályt, és az ebből leszármazó *Shop*, *CartAndOrder* és *Profile* objektumokat, amik reprezentálják az alsó navigációs sávból elérhető képernyőket. Mindegyik képernyőhöz tartozik egy *route*, ami az útvonalat, egy *resourceId*, ami a képernyő nevét és egy *icon*, ami a képernyő ikonját definiálja. A képernyő neve és ikonja jelenik meg az alsó navigációs sáv gombjaiban. A fent említett osztályokat és függvényeket a *NavHost* implementációmban használom fel, ahol az alkalmazás navigációs gráfját definiálom.

### Hálózati kommunikáció, autentikáció

A hálózati kommunikáció megvalósításához az OkHttp és a Retrofit könyvtárakat használtam. Létrehoztam a *CurlyCircleApi* interfészt, ebben megvalósítottam az összes függvényt, ami a hálózati kommunikációhoz szükséges. Ezek a függvények megfelelnek a szerveralkalmazás API végpontjainak. Ezeket a függvényeket elláttam a megfelelő annotációkkal, így ebből az interfészből a Retrofit képes egy HTTP kliens generálni. Például a felhasználó korábbi rendeléseinek lekéréséhez a következő függvényt definiáltam:

@GET("user/{userId}/orders")  
suspend fun getUserOrders(@Path("userId") userId: Int): OrdersViewModel

Jól látható a kódrészletből, hogy a *@GET* annotáció megadja, hogy ez egy GET típusú HTTP kérés lesz, illetve az annotáció paramétere megadja a relatív útvonalat, amit a bázis útvonal mögé fog illeszteni a Retrofit, és így az API megfelelő végpontját fogja felhívni. A függvény *@Path* annotációval ellátott paraméterének értéke be lesz illesztve a relatív útvonalban a *{userId}* helyére. A Retrofit könyvtár így teszi lehetővé az URL manipulációját.

Az alkalmazás fejlesztése során már futott az Azure webszerveren lévő változata a szerveralkalmazásnak, így fejlesztés során is azzal a szerverrel kommunikált az Android alkalmazásom, ezért definiáltam egy statikus változót az interfészen belül, ami az API bázis URL-jét tartalmazza. Ezután a korábban említett *NetworkModule* Hilt modulban definiáltam a függvényt, amely bekonfigurálja és létrehozza a megfelelő *Retrofit* osztály példányt:

@Provides  
@Singleton  
fun provideCurlyCircleApi(  
 okHttpClient: OkHttpClient  
): CurlyCircleApi {  
 return Retrofit.Builder()  
 .baseUrl(CurlyCircleApi.BASE\_URL)  
 .client(okHttpClient)  
 .addConverterFactory(  
 GsonConverterFactory  
 .create(GsonBuilder().create()))  
 .build().create(CurlyCircleApi::class.*java*)  
}

A *Retrofit* *Builder* segítségével létrehozom a *CurlyCircleApi* interfészből a *Retrofit* példányt, és beállítom neki, hogy a Gson könyvtárat használja konvertálóként. A Gson alakítja át a Java objektumokat JSON objektumokká és fordítva. A kódból is látszik, hogy a *Retrofit* példánynak szüksége van egy *OkHttpClient* típusú objektumra, amelyet beállít kliensként magának. Ez azért van, mivel az autentikáció úgynevezett *Interceptor* és *Authenticator* osztályok segítségével történik, amelyeket az *OkHttpClient* objektumnak adok meg. Ezért szükség volt egy másik függvényre, amelyik bekonfigurálja és létrehozza ezt az *OkHttpClient* példányt:

@Singleton  
@Provides  
fun provideCurlyCircleOkHttpClient(  
 tokenInterceptor: TokenInterceptor,  
 tokenAuthenticator: TokenAuthenticator  
): OkHttpClient = OkHttpClient.Builder()  
 .addInterceptor(tokenInterceptor)  
 .authenticator(tokenAuthenticator)  
 .build()

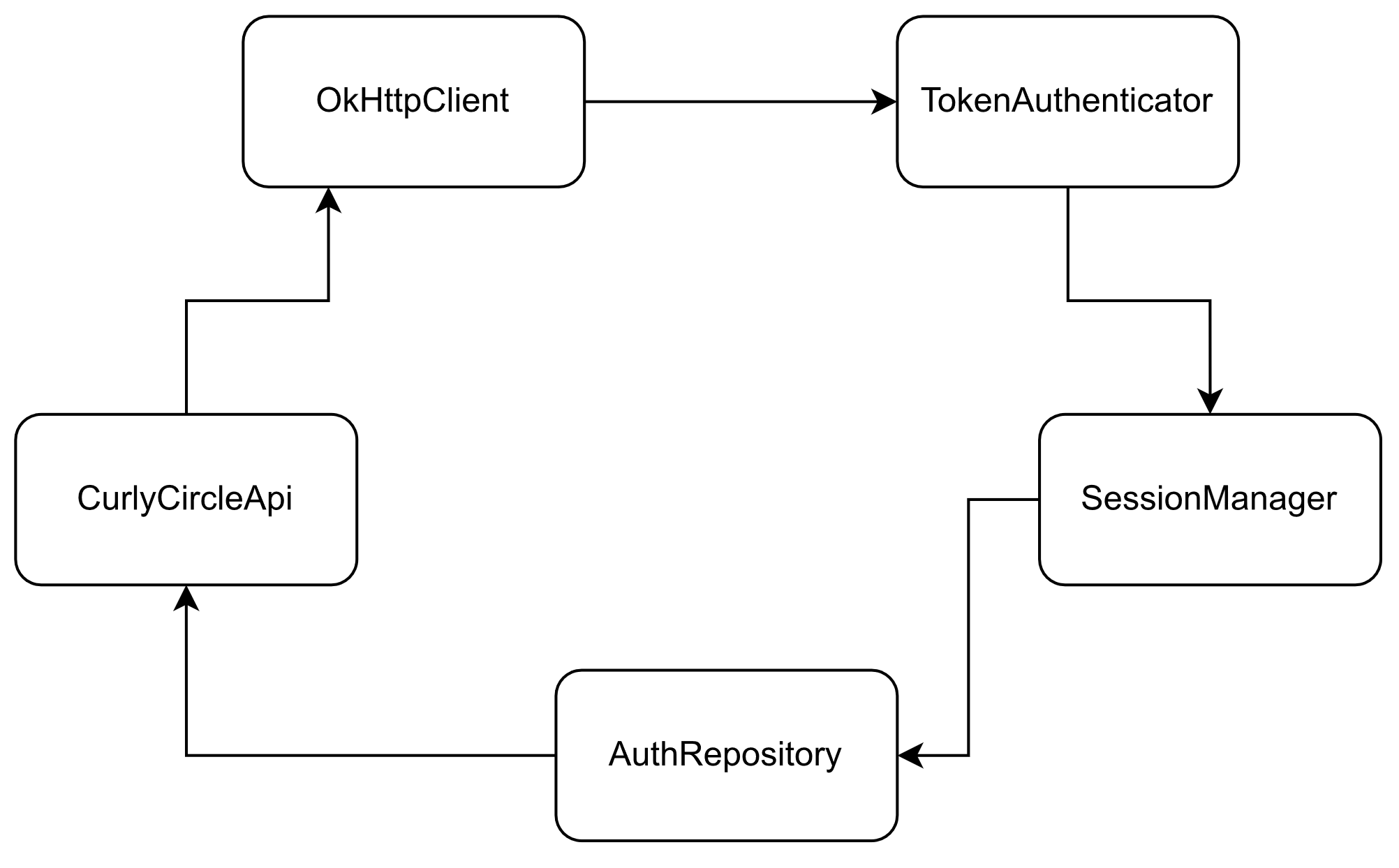
Itt az *OkHttpClient* példányt felkonfigurálom a megadott *TokenInterceptor* és *TokenAuthenticator* példányokkal. Ezekre a refresh token alapú autentikáció miatt van szükség.

A *TokenInterceptor* osztályom implementálja az OkHttp könyvtár *Interceptor* interfészét. Az *Interceptor* lényege, hogy megfigyeli a kimenő és bejövő HTTP kéréseket és válaszokat, és akár módosíthatja is azokat. Tipikusan arra használjuk őket, hogy módosítsuk a kérések fejlécét. Az *Interceptor* interfésznek csupán egy metódusa van, az *intercept()* függvény, amellyel módosíthatjuk magát a HTTP kérést, mielőtt kiküldenénk azt. Ennek a metódusnak a testében írtam meg azt logikát, ami hozzáfűzi a kérés Authorization fejlécébe a felhasználó access tokenjét.

A *TokenAuthenticator* osztályom az OkHttp könyvtár *Authenticator* interfészét valósítja meg, aminek csupán egyetlen függvénye van, az *authenticate()*. Ez a függvény akkor hívódik fel, amikor az alkalmazás egy 401 Unauthorized kóddal rendelkező HTTP választ kap a szervertől. A függvény testében a következőképp implementáltam a logikát: először megpróbálok szerezni egy frissített access tokent a szervertől, ha ez nem sikerül, mert például a refresh token is lejárt, akkor kijelentkeztetem a felhasználót, ha sikerül, akkor újra küldöm a HTTP kérést, viszont beillesztem a frissített access tokent a kérés Authorization fejlécébe.

A *TokenInterceptor* és a *TokenAuthenticator* is a *SessionManager* osztályommal kommunikál, tőlük kérik le és frissítik a felhasználó tokenjeit, illetve rajta keresztül jelentkeztetik ki a felhasználót. Ez igazából egy kisegítő osztály, ami az *AuthRepository* osztályba hív át. Az *AuthRepository* az a repository osztály, amely az autentikációhoz kapcsolódó összes műveletet tartalmazza, kommunikál a lokális adatbázissal, és a szerver alkalmazással is. A függvényei megvalósítják a bejelentkezést, a kijelentkezést, a regisztrációt, a felhasználó tokenjeinek frissítését és az eszközön lévő tokenek lekérését.

Androidon a refresh token autentikáció megvalósítása közben nagyon sok nehézségbe ütköztem, sokáig nem találtam egy egyértelmű, tiszta megoldást. A fő probléma az volt, hogy a token frissítő logika implementálása közben a Hilt keretrendszerben egy körkörös függőség alakult ki. Tömören az volt a probléma, hogy az *OkHttpClient* függött az *TokenAuthenticator* osztálytól, hiszen ezen keresztül frissítjük a tokent, amikor az access token lejár. A *TokenAuthenticator* osztály függ a *SessionManagertől*, amely függ az *AuthRepository* osztálytól, hiszen rajta keresztül tudjuk frissíteni a felhasználó tokenjét, ő viszont függ a *CurlyCircleApi* interfésztől mivel azon keresztül folyik a hálózati kommunikáció. Maga a *CurlyCircleApi* interfész, viszont függ az *OkHttpClient* típusú objektumtól, hiszen ő kell hozzá, hogy működjenek az *Interceptor* és az *Authenticator* osztályok. Így alakult ki ez a függőségi ciklus, amelyhez készítettem egy vizualizációt, ezt a 4. ábra tartalmazza.



4. ábra - A ciklikus függőség vizualizációja

Erre a problémára nem találtam egy igazán frappáns, tiszta megoldást, így egy olyan megoldást választottam, amelyet a fórumok alapján sokan mások is használtak. A megoldás lényege, hogy leválasztottam az autentikációhoz szükséges HTTP kéréseket egy külön Retrofit interfészbe – ez az *AuthApi*. A NetworkModule modulban létrehoztam egy külön függvényt, ami elkészít egy külön Retrofit példányt, amelyik ezt az interfészt használja, és nem a *CurlyCircleApi* interfészt. Így az *AuthRepository* csak ettől az interfésztől fog függeni, és így sikerült felbontani a ciklikus függőséget. Így az alkalmazásom kétféle *Retrofit* példánnyal dolgozik, ezt nem tartom a legszebb megoldásnak, de működik, és ennél jobb, tisztább megoldást nem találtam.

### Adatbázis

Mint ahogy korábban említettem, egy offline-first alkalmazás lényege, hogy rendelkezzen valamilyen lokális adatforrással is, ahonnan hálózati kapcsolat hiányában ki tudja szolgálni az adatokat. Én lokális adatforrásnak a Room adatbázist választottam, amely egy SQLite alapú relációs adatbázissal dolgozik a háttérben. A Room használatához három fő komponensre van szükségünk: az adatbázis osztályra, entitásokra és DAO osztályokra.

Az alkalmazás adatbázisát a *CurlyCircleDatabase* absztrakt osztály reprezentálja. Ez a *RoomDatabase* ősosztályból származik le, és ebből fogja a Room könyvtár a tényleges adatbázis osztályt generálni. Itt definiáltam annotációk segítségével az entitásokat reprezentáló modell osztályokat, valamint absztrakt függvények segítségével az adatbázis DAO objektumait.

A Data Access Object (DAO) lényege, hogy rajta keresztül érjük el az adatbázisban tárolt adatokat. Egy DAO igazából egy *@Dao* annotációval ellátott interfész, amely függvényeket tartalmaz az adatok elérésére, hozzáadására, módosítására, illetve törlésére. Az interfész függvényeit mi definiáljuk, így mi adhatjuk meg, hogy milyen műveleteket szeretnénk később végezni az adatokon. Például az alkalmazásomban létrehoztam a *ProductsDao* interfészt, ez a termék entitások elérésére szolgál:

@Dao  
interface ProductsDao {

@Query("SELECT \* FROM products WHERE id = :id")  
 suspend fun getProduct(id: Int): Product  
  
 @Insert(onConflict = OnConflictStrategy.*REPLACE*)  
 suspend fun insertProduct(product: Product)

//...

}

A kódrészletben látható a *getProduct(id: Int)* függvény, ez lekér egy adott azonosítóval rendelkező terméket az adatbázisból. Ezt a funkcionalitást a *@Query* annotáció paraméterében definiáltam SQL lekérdezéssel. Az *instertProduct(product: Product)* függvény hozzáad egy terméket az adatbázishoz. A Room az *@Insert* annotáció miatt fogja tudni, hogy ezt kell tennie.

Az entitásokat a *Models.kt* nevű fájlban definiáltam data class osztályokként. A data class egy Kotlin nyelvi specialitás, olyan osztályt lehet vele definiálni, amely egyetlen felelőssége, hogy adatot tárol. Például a termékkategória entitást a következőképp definiáltam:

@Entity(tableName = "productCategories")  
data class ProductCategory(  
 var name: String = "",  
 var description: String = "",  
 var thumbnailImageUrl: String = "",  
 var isAvailable: Boolean = true,  
 @PrimaryKey var id: Int  
 )

Látható, hogy az *@Entity* annotáció segítségével megadtam a Room számára, hogy ez az osztály egy entitás lesz az adatbázisban, valamint megadtam az annotáció paramétereként a tábla nevét, amelyben szerepelnie kell ezeknek az entitásoknak. A *@PrimaryKey* annotációval adtam meg, hogy az *id* mező lesz az entitás elsődleges kulcsa az adatbázisban. Az adatbázisban a következő entitások szerepelnek: termékkategóriák (*ProductCategory*), termékek (*Product*), rendelések (*Order*), kosár tételek (*CartItem*) és felhasználó (*User*). A felhasználó tábla maximum egy felhasználó adatait tárolja, az éppen bejelentkezett felhasználóét – ha van.

### UI komponensek, kisegítő osztályok

Létrehoztam néhány újra felhasználható UI komponenst, amelyek az alkalmazásban több helyen is megjelennek. Ezeket a komponenseket mind egy-egy composable függvény írja le. A Jetpack Compose keretrendszer segítségével nagyon könnyű létrehozni újra felhasználható UI elemeket, hisz egy-egy kis elemet egy-egy composable függvénnyel le tudunk írni, és ezeket bármelyik képernyőhöz tartozó composable függvényből fel tudjuk hívni.

Az első és legfontosabb a *LoadingContent* komponens. Ez a composable az alkalmazás szinte összes képernyőjén megjelenik. Lényege, hogy megadhatunk neki egy composable függvényt, ami alap esetben a képernyő tartalma lesz, illetve egy *loading* *Boolean* típusú változót, amelynek, ha igaz az értéke, akkor a tartalom helyett egy töltési indikátort fog mutatni. Ha a *loading* értéke hamis, akkor a megadott composable függvényt fogja mutatni. Emellett tartalmaz egy *empty* Boolean változót is, amely azt jelzi, hogy van-e adat, amit meg tudunk jeleníteni. Amennyiben az *empty* igaz, akkor az *emptyContent* composable függvény által megadott tartalmat fogja kirajzolni.

@Composable  
fun LoadingContent(

//...  
) {  
 if (loading) {  
 *Box*(/\*...\*/) **{** *CircularProgressIndicator*()  
 **}** }  
 else if (empty) {  
 emptyContent()  
 } else {  
 content()  
 }  
}

A kódrészleten látható, hogy a Jetpack Compose keretrendszer számára a *@Compose* annotációval tudjuk jelezni, hogy itt egy composable függvényt várunk.

Emellett létrehoztam egy *QuantityPicker* nevű composable függvényt, amellyel a Kosár és a Termék részletei képernyőkön tudja beállítani az adott termék mennyiségét. Végül létrehoztam egy *PasswordTextField* nevű composable elemet, ami a Bejelentkezés és Regisztráció képernyőkön van használva. Ez a jelszó mezőkhöz megszokott módon alapból kicsillagozza a bevitt szöveget, és tartalmaz egy gombot, amely segítségével válthatunk a jelszó elrejtése és megjelenítése között.

Létrehoztam egy *Result* osztályt, amelynek csupán két leszármazott osztálya létezik, a *Success* és az *Error*. Ezeket az osztályokat arra használtam, hogy a repository és az adatforrás osztályokban a végrehajtott műveletek értékét becsomagoltam ezekbe Success és Error objektumokba, így az az osztály, ami a műveletet felhívta, nagyon könnyen el tudja róla dönteni, hogy sikeres volt-e, vagy nem. A *Success* osztály tartalmaz egy *data* nevű mezőt, ez tartalmazza a művelet visszatérési értékét, amennyiben sikeres volt. Az *Error* osztály egy *exception* nevű mezőt tartalmaz, amely a sikertelen művelet során dobott kivételt tartalmazza, ezzel információt szolgáltatva arról, hogy milyen típusú hiba lépett fel.

sealed class Result<out R> {  
 data class Success<out T>(val data: T) : Result<T>()  
 data class Error(val exception: Exception) : Result<Nothing>()  
 //...  
}

## Termékek, termékkategóriák

Ebben az alfejezetben a Bolt célállomás képernyőit és azok mögöttes implementációját mutatom be. A célállomáson belül három képernyő szerepel: termékkategóriák, termékek és termék részletei. A UI rétegben mindegyik képernyőhöz tartozik pontosan egy composable függvény, ami leírja a képernyőn látható felhasználói felületet, illetve egy *ViewModel* osztály, ami a UI állapotát tartalmazza.

A Termékkategóriák képernyő lényege, hogy egy görgethető listában megjelennek a webshopban lévő termékkategóriák, majd egy termékkategóriára kattintva átkerül a felhasználó a Termékek oldalra, ahol a kiválasztott kategóriához tartozó termékek jelennek meg. Készítettem egy diagrammot a Termékkategóriák képernyő mögöttes osztályainak architektúrájáról, hogy könnyebb legyen bemutatni az implementáció felépítését. A függelékben található 10. ábra tartalmazza ezt a diagrammot. Egyébként az alkalmazás többi képernyőjének a mögöttes architektúrája is nagyon hasonlóan néz ki ehhez, hiszen próbáltam az egész alkalmazásban követni ezt a megközelítést.

A *CurlyCircleApi* *getProductCategories()* függvénye segítségével felhívom a szerver „productcategory” végpontját egy GET metódussal. Ez visszaadja a webshopban lévő termékkategóriákat egy listában. A *ProductCategoriesRemoteDataSource* osztály ezt a függvényt hívja fel úgy, hogy kihelyezi a végrehajtást egy külön szálkészletre (thread pool), mivel a hálózati hívás egy hosszú ideig tartó, blokkoló művelet, és nem szabad ilyen sok időre blokkolni a fő futási szálat, hiszen akkor a felhasználó annyit látna, hogy lefagy az alkalmazás amíg meg nem érkezik a szervertől a válasz. Így ez a függvény fel lesz függesztve, és visszakapja a végrehajtást az eredeti szál, amíg meg nem érkezik a szervertől a válasz. Ekkora a függvény visszatér a szervertől kapott válasszal, amennyiben az sikeres volt, és van tényleges adat, akkor becsomagolja egy *Success* objektumba, amennyiben nem, akkor a szerver által dobott *Exceptiont* becsomagolja egy *Error* objektumba.

A *ProductCategoriesLocalDataSource* osztály kezeli a termékkategóriák elmentését, törlését, kiolvasását a lokális adatbázisból. Ezt a *ProductCategoriesDao* DAO interfészen keresztül csinálja, valamint ő is becsomagolja a kapott válaszokat *Success* és *Error* objektumokba. A lokális és távoli adatforrás osztályok ugyanazt a *ProductCategoriesDataSource* interfészt implementálják, így az őket használó repository osztály számára egy egységes interfészt mutatnak, így könnyű a használatuk.

A *DefaultProductCategoriesRepository* osztály kezeli a lokális és a távoli adatforrások egyeztetését. Egy offline-first alkalmazásban előnyös, ha a repository osztályok mindig a lokális adatbázisból adják vissza az adatokat, tehát nem tudjuk lekérni közvetlenül a távoli adatbázisból azokat. Ennek előnye, hogy a lokális adatbázisban tárolt adatok konzisztensek maradnak az alkalmazás többi részével, azaz a felhasználói felületen nem jelenik meg olyan adat, ami az adatbázisban nem szerepel. Ennek megfelelően implementáltam ezt az osztályt, mindig a lokális adatbázisból adja vissza a termékkategóriákat, azonban a *getProductCategories(forceUpdate: Boolean)* függvény paraméterének segítségével megkérhetjük, hogy először kérje le a termékkategóriákat a szerver alkalmazástól, majd ezután adja vissza azokat. Az osztályban emellett létrehoztam egy *getProductCategoriesStream()* függvényt, amely egy *Flow* típusba csomagolva adja vissza a termékkategóriákat a lokális adatbázisból. A *Flow* a Kotlin nyelv egy beépített típusa, amely egy aszinkron adatfolyamot reprezentál, nagyon hasonló az RxJava *Observable*, vagy a C# *Observable* típusához. Lényege, hogy a UI rétegben könnyebben tudunk reagálni az adatok változására.

A *ProductCategoriesViewModel* osztályt abból a célból készítettem, hogy ő kezelje és hozza létre a UI állapotot. Ő kommunikál a *DefaultProductCategoriesRepository* osztállyal, onnan kapja az adatokat. Létrehoztam egy *ProductCategoriesUiState* osztályt, amely a UI állapotot reprezentálja, így a ViewModel osztálynak nem szükséges több publikus tulajdonságot elérhetővé tennie, csupán ezt az egyet.

data class ProductCategoriesUiState(  
 val productCategories: List<ProductCategory> = *emptyList*(),  
 val isLoading: Boolean = false,  
 val userMessage: Int? = null  
)

A kódból látszik, hogy ez az osztály igazából három tulajdonságot fog össze. A *productCategories* lista a termékkategóriákat tartalmazza, az *isLoading* a töltést jelzi, illetve a *userMessage* a felhasználó számára megjelenítendő üzenetet tartalmazza. A UI állapotot egy *StateFlow* típusba burkolva teszem elérhetővé a UI számára. A repository osztálytól kapott *Flow* típust a *stateIn* operátorral konvertálom *StateFlow* típussá. A *StateFlow* típus hasonlít az Androidon korábban használt *LiveData* típusra, lényege, hogy mindig pontosan egy értéke van, támogat több megfigyelőt is és az új feliratkozók mindig megkapják a legutóbbi értéket. A típus lényege, hogy biztonságosan és effektíven megfigyelhetővé tegyük a UI állapotot. Az alkalmazás összes ViewModel osztálya *StateFlow* típusba burkolva teszi elérhetővé, megfigyelhetővé a UI állapotot.

A *ProductCategoriesScreen* composable függvényben a UI állapotot a ViewModel osztálytól a beépített *collectAsStateWithLifecycle()* függvény segítségével figyelem meg. Ez a függvény a Jetpack Compose API része és lényege, hogy életciklus tudatos módon gyűjti be a *StateFlow* új értékeit. Ez azt jelenti, hogy az értékek gyűjtése csak akkor folyik, amikor az adott képernyő előtérben van, így feleslegesen nem használja a készülék erőforrásait. A termékkategóriákat a Jetpack Compose könyvtárba beépített *LazyVerticalGrid* UI elem segítségével jelenítem meg. Ennek lényege, hogy egy függőleges görgethető rácsba tehetjük bele az elemeinket, amely optimalizálva van a sok elem megjelenítésére. Ez úgy működik, hogy csak azok az elemek vannak tényleges vizuálisan létrehozva, kirajzolva, amelyek a képernyőn szerepelnek, és görgetés közben lesznek dinamikusan létrehozva és eldobva az elemek. A rácsot adaptívra állítottam, illetve beállítottam egy minimális szélességet az elemeknek, így a képernyő mérettől függ, hogy hány oszlopa van a rácsnak. Például telefonokon egy oszlop lesz, tableteken, akár több is.

Amikor a felhasználó rányom egy termékkategóriára, átkerül a Termékek oldalra, ahol az adott kategóriába tartozó termékek jelennek meg. Ennek a képernyőnek az implementációját nem részletezem, teljesen ugyanúgy építettem fel, mint a Termékkategóriák képernyőt, ugyanúgy a *LazyVerticalGrid* elemet használom a termékek megjelenítésére, illetve az osztály architektúra is teljesen ugyanúgy néz ki, csupán a ViewModel, a repository és az adatforrás osztályok nem termékkategóriákkal, hanem termékekkel dolgoznak. Az egyetlen jelentős különbség, hogy a *ProductViewModel* a *SavedStateHandle* segítségével megkapja a navigálás során átadott termékkategória azonosítót, és ez alapján kéri le a megfelelő termékeket a szerver alkalmazástól.

Miután a felhasználó kiválaszt egy terméket, a Termék részletei képernyőre kerül. Ezen a képernyőn látja a termék nevét, leírását, képét, illetve itt teheti a kosarába a terméket. Ez a képernyő a Termékek képernyőhöz hasonlóan a *DefaultProductsRepository* osztályt, illetve a mögötte lévő adatforrás osztályokat használja a termék adatainak lekéréséhez. Emellett a *DefaultCartRepository* osztályt is használja, hiszen ennek segítségével valósítottam meg a termék hozzáadását a felhasználó kosarához.

A *ProductDetailsViewModel* osztály abban különbözik a korábban bemutatott *ProductCategoriesViewModel* osztálytól, hogy itt a UI állapotot nem csak a repository osztálytól kapott adatok befolyásolják. A felhasználó a *QuantityPicker* elem segítségével kiválaszthatja, hogy hány darabot szeretne az adott termékből a kosarába tenni. Így a” source of truth” nem csupán a repository osztály, hanem maga a ViewModel osztály is. Épp ezért a UI állapotot ebben az osztályban közvetlenül is módosítanom kell. Ezt úgy oldottam meg, hogy a publikusan is elérhető *uiState* tulajdonság mögé egy privát *MutableStateFlow* típusú *\_uiState* mezőt raktam. A *MutableStateFlow* *update()* metódusa segítségével itt az osztályban közvetlenül módosítani tudom a UI állapotát.

private val \_uiState = *MutableStateFlow*(ProductDetailsUiState(isLoading = true))  
val uiState: StateFlow<ProductDetailsUiState> = \_uiState  
 .*stateIn*(/\*...\*/)

fun increaseQuantity() {

//...  
 \_uiState.*update* { **it**.copy(quantity = \_uiState.value.quantity + 1)  
 }}

A kódrészletből látszik, hogy az *increaseQuantity()* függvényben eggyel növelem a mennyiség számát, és így ezzel módosítom a UI állapotot.

## Profil

A Profil célállomáshoz három képernyő tartozik: a Felhasználói fiók, a Regisztráció és a Rendelések képernyő.

A Felhasználói fiók képernyő megjeleníti a felhasználó adatait, illetve innen navigálhat a felhasználó a Rendelések képernyőre, és itt tud kijelentkezni is. Amennyiben a felhasználó nincs bejelentkezve, akkor ezen a képernyőn egy bejelentkezési űrlapot lát. Készítettem egy diagrammot a Felhasználói fiók képernyőhöz tartozó osztályok architektúrájáról, ezt tartalmazza a függelékben található 11. ábra. Látható az ábrán, hogy az *AccountViewModel* kétféle repository osztályt is használ, az egyiket az autentikációs műveletekhez, a másikat a felhasználó adatainak lekéréséhez. Ezek a repository osztályok nem adatforrás osztályokkal dolgoznak, hanem közvetlenül a DAO és az API interfészekkel. Ennek oka, hogy ezeknél az eseteknél annyira eltérnek a lokális és a hálózati műveletek, hogy nem láttam értelmét közös interfészt implementáló adatforrás osztályok létrehozásának. Ezzel nincs probléma, a Google útmutatója szerint is elhagyhatóak az adatforrás osztályok, ennek egyetlen következménye, hogy a repository osztályoknak kicsit nagyobb felelőssége lesz.

A *DefaultUserRepository* felelőssége egyedül az, hogy visszaadja a felhasználó adatait. A *DefaultAuthRepository* osztály több műveletért is felelős: a bejelentkezésért, a kijelentkezésért, a regisztrációért, a felhasználó tokenjeinek frissítéséért, illetve a tokenek írásáért és olvasásáért. A felhasználó tokenjeit akkor kapom meg a szervertől, amikor bejelentkezik, ezt az eszközön eltárolom a felhasználó adataival együtt a lokális adatbázisban. Amikor a felhasználó hozzáférési tokenje lejár, a repository *refreshToken()* függvénye lekéri a felhasználó új tokenjeit, majd elmenti azokat az adatbázisba. Kijelentkezésnél a felhasználó adatait törlöm az adatbázisból.

Ezen a képernyőn két alapvetően különböző UI állapot lehetséges, attól függően, hogy a felhasználó be van-e jelentkezve. Ezt úgy modelleztem, hogy létrehoztam egy *AccountUiState* interfészt, amelyet két data class osztállyal implementáltam, a *HasUser* és a *NoUser* osztályokkal. Az interfész két tulajdonságot deklarál, amelyek közösek a két osztályban: az *isLoading* és a *userMessage* tulajdonságokat, az előbbi a töltést indikálja, az utóbbi a felhasználó számára megjelenítendő üzenetet tartalmazza. A *HasUser* emellett tartalmaz egy *User* típusú *user* tulajdonságot, amely a felhasználó adatait tartalmazza. A *NoUser* email és egy password tulajdonságot tartalmaz, ezek a bejelentkezési űrlap állapotát reprezentálják. A ViewModel osztály azonban egy belső reprezentációval dolgozik, amelyet az *AccountViewModelState* osztály modellez. Ez az osztály egyszerre tartalmazza a *user*, *email* és *password* tulajdonságokat is. Itt azonban a *user* tulajdonság *User?* típusú, modellezve azt, hogy a felhasználó lehet *null* értékű is. Ez az osztály rendelkezik egy *toUiState()* függvénnyel, amely a tulajdonságok alapján létrehozza a *NoUser* vagy *HasUser* objektumokat.

fun toUiState(): AccountUiState =  
 if (user == null) {  
 AccountUiState.NoUser (/\*...\*/)  
 } else {  
 AccountUiState.HasUser(/\*...\*/)  
 }

Az *AccountViewModel* itt is egy privát *MutableStateFlow* mezővel dolgozik, hiszen a bejelentkezési űrlap miatt maga a ViewModel osztály is befolyásolja a UI állapotot. Amennyiben a felhasználó helytelen bejelentkezési adatokat adott meg, a userMessage tulajdonságot frissítem, és az *AccountScreen* composable függvényben egy *Snackbar* üzenet formájában tájékoztatom a felhasználót erről. Az *AccountScreen* függvényben úgy kezelem le a különböző UI állapotokat, hogy azoknak megfelelően teljesen különböző tartalmat rajzolok ki a képernyőre.

when (uiState) {  
 is AccountUiState.HasUser -> *LoggedInContent*(  
 //...  
 )  
 is AccountUiState.NoUser -> *AnonymousContent*(  
 //...  
 )  
}

A Rendeléseim képernyőn a felhasználó a korábban leadott rendeléseit tudja megtekinteni. Itt az *OrdersScreen* composable függvényen belül a *LazyVerticalGrid* elem segítségével egyszerűen kilistázom a felhasználóhoz tartozó rendelések részleteit. Látható a rendelés dátuma, összege, és a megjegyzés, amit a felhasználó adott. Az *OrdersViewModel* osztály logikailag teljesen megegyezik a korábban említett *ProductCategoriesViewModel* osztály felépítésével, itt azonban az *DefaultOrdersRepository* osztálytól kéri le az adatokat.

A Regisztráció képernyőre a felhasználó akkor tud elnavigálni, ha nincs bejelentkezve. Itt új profilt tud regisztrálni egy űrlap kitöltésével. A *RegistrationScreen* composable tartalmazza ezt az űrlapot, a felhasználó itt megadja az adatait a szükséges és opcionális mezők kitöltésével. A *RegistrationViewModel* osztály szintén rendelkezik egy privát *MutableStateFlow* típusú mezővel, hiszen az űrlap mezőinek értékeit a ViewModel tárolja, így itt is befolyásolja a ViewModel a UI állapotot. Ez az osztály a *DefaultAuthRepository* *register()* függvényét hívja fel, így regisztrálja be a felhasználót.

## Kosár, rendelés

A Kosár és rendelés célállomáshoz két képernyő tartozik: a Kosár és a Rendelés képernyők. A felhasználó a Kosár képernyőn tekintheti meg a kosarának tartalmát, illetve itt módosíthatja még azt. Kiürítheti a teljes kosarat, eltávolíthatja az egyes termékeket, illetve változtathatja azok mennyiségét. Emellett, ha elégedett a kosarának a tartalmával itt tud átnavigálni a Rendelés képernyőre. A Kosár képernyőhöz tartozó osztályok architektúrájának diagramját a függelékben található 12. ábra tartalmazza.

A *DefaultCartRepository* osztályban implementáltam azt a bonyolultabb kosár kezelési logikát, amit korábban már a webes kliensnél is bemutattam. A *DefaultCartRepository* osztály egyrészt használja a *CurlyCircleApi* interfészt, ezen keresztül végzi el a kosárhoz tartozó írási műveleteket, illetve amikor szükséges, innen frissíti a kosár tényleges tartalmát. Másrészt használja a *CartItemsDao* interfészt, ezen keresztül menti el és olvassa ki a kosár tartalmát a lokális adatbázisból.

A legtöbb objektum-reláció leképezési könyvtár támogatja azt, hogy az entitás objektumok referálják egymást, azonban a Room kifejezetten tiltja ezt. Ez a mobil platform performancia korlátjai miatt van így. Emiatt sajnos nem tudtam ugyanúgy létrehozni a *CartItem* entitás objektumot, mint a szerver adatbázisában, hiszen annak tartalmaznia kéne egy *Product* típusú mezőt. Ezt a Room útmutatója alapján úgy oldottam meg, hogy létrehoztam egy *CartItemAndProduct* osztályt, amely tartalmazza a megadott *CartItem* objektumot, és a hozzátartozó terméket is.

data class CartItemAndProduct(  
 @Embedded val product: Product,  
 @Relation(  
 parentColumn = "id",  
 entityColumn = "productId"  
 )  
 val cartItem: CartItem  
)

A kódban látható, hogy a *@Relation* annotáció segítségével definiáltam a kapcsolatot a két entitás között.

@Transaction  
@Query("SELECT \* FROM cartItems, products WHERE cartItems.productId = products.id")  
suspend fun getCartItems(): List<CartItemAndProduct>

Ezt követően a *CartItemsDao* interfészben a *getCartItems()* és *getCartItemsStream()* függvények *@Query* annotációjával megadtam a megfelelő SQL lekérdezést. A *@Transaction* annotációra azért van szükség, mert itt igazából két lekérdezés történik, ez garantálja, hogy úgy fognak ezek lefutni, mintha egy lekérdezés történt volna. Innentől kezdve a repository és a ViewModel osztályok is ezzel az új objektummal dolgoznak, így meg tudom jeleníteni a felhasználói felületen a megfelelő termékeket a kosárban.

A *DefaultCartRepository* osztály felhasználja a *DefaultUserRepository* osztályt, mivel itt kell reagálni a felhasználó változásra, azaz a be- és kijelentkezésre. Ezt úgy implementáltam, hogy a *DefaultUserRepository* által publikált *User* adatfolyamra feliratkoztam, és minden alkalommal mikor új érték érkezik az adatfolyamon, felhívom vele a *handleUserChanged()* függvényt. Ennek a lényege, hogy amikor egy nem *null* *User* érték érkezik, akkor a felhasználó kosarának azonosítóját elmentem a *SharedPreferences* segítségével. Emellett egy *Boolean* értékkel azt is elmentem, hogy a kosár egy bejelentkezett felhasználóhoz tartozik. Ezután a szervertől lekérjük a kosár aktuális tartalmát. Ha a kapott *User* érték *null*, akkor lekérjük a *SharedPreferences* osztálytól, hogy a jelenlegi kosár bejelentkezett felhasználóhoz tartozik-e.

if (user != null) {  
 sharedPreferences.setIsCartAnonymous(false)  
 sharedPreferences.setCartId(user.cartId)  
 updateCartItemsFromRemoteDataSource(user.cartId)  
} else {   
 if (!sharedPreferences.getIsCartAnonymous()) {  
 sharedPreferences.setIsCartAnonymous(true)  
 sharedPreferences.setCartId(0)  
 dao.deleteCartItems()  
 }  
}

Ha nem, akkor nincs teendőnk, ha igen, az azt jelenti, hogy most jelentkezett ki a felhasználó, így el kell távolítanunk a kosarát. A *SharedPreferences* osztállyal kitöröljük az azonosítóját, a DAO interfész segítségével pedig kitöröljük a lokális adatbázisból a kosár tartalmát. A *SharedPreferences* osztály egy beépített kisegítő osztály Androidon, segítségével egyszerű kulcs-érték párokat tudunk elmenteni az eszközre.

A *CartViewModel* osztály meghívja a repository osztály megfelelő függvényeit a különböző írási és olvasási műveletekhez, illetve publikálja egy *CartUiState* típusú objektumban a UI állapotot. A *CartScreen* composable a UI állapota alapján vagy megjeleníti a kosár tartalmát egy dinamikus listában, vagy ha a kosár üres, akkor kiírja azt. A felhasználó csak akkor tud átnavigálni a Rendelés képernyőre, ha a kosár nem üres.

A Rendelés képernyőn a felhasználónak ki kell töltenie a rendelési űrlapot, meg kell adnia a rendeléshez szükséges adatokat. Amennyiben be volt jelentkezve a felhasználó az űrlapon a személyes adatok automatikusan kitöltődnek. Emellett a felhasználónak meg kell adnia a szállítási és fizetési módot is, illetve megadhat egy opcionális megjegyzést a rendeléshez. Az *OrderViewModel* osztályban a *DefaultUserRepository* osztályt használom ahhoz, hogy létrehozáskor automatikusan kitöltsem a rendelési űrlapot a felhasználó adataival. A *DefaultCartRepository* osztályt használom arra, hogy a rendelési adatokhoz hozzácsatoljam a kosár azonosítóját is, így a szerver alkalmazás tudni fogja, hogy melyik kosár tartalmát rendelte meg a felhasználó. A *DefaultOrdersRepository* *placeOrder()* függvényét hívom fel a rendelés leadásához, ez pedig a *CurlyCircleApi* segítségével kérést küld a szerver megfelelő végpontjára. Amennyiben a rendelés sikeres, visszanavigálom a felhasználót a Kosár képernyőre, ha azonban sikertelen, akkor egy *Snackbar* üzenetben tájékoztatom a felhasználót.

# Összefoglaló

A diplomamunka kereteiben megterveztem és megvalósítottam egy webshop rendszert, amely egy szerver oldali és két kliens oldali alkalmazásból áll. A szerver oldali és webes kliens alkalmazásokat emellett telepítettem is, így bárki számára elérhető és használható, ezzel igazoltam, hogy nem csak a fejlesztői környezetben működik. Emellett a szerver oldali alkalmazás üzleti logikát implementáló részeit egységtesztek implementálásával megfelelően leteszteltem.

A diplomamunka elkészítése során úgy érzem, hogy szakmailag, szoftverfejlesztőként rengeteget fejlődtem, hiszen az elején alig volt tapasztalatom a webes fejlesztés területén, és a végére eljutottam odáig, hogy megismertem a főbb modern webes technológiákat és átlátom a webalkalmazások működését. Az alkalmazások elkészítése közben több jelentős fejlesztői kihívásba ütköztem, amelyeket nagyobb energia- és időbefektetések árán tudtam csak leküzdeni, azonban úgy érzem, hogy ezeknek a megoldásából tanultam a legtöbbet.

A szerver oldali alkalmazás fejlesztése számomra pozitív élmény volt, mivel az ASP.NET Core és EF Core technológiák kiváló dokumentációval és útmutatókkal rendelkeznek, így a nagyobb kihívásoknál is mindig volt, amire támaszkodjak. A webes kliens alkalmazás fejlesztése közben vegyes tapasztalatokat szereztem, számomra nem volt nagy élmény HTML és CSS technológiákkal dolgozni, illetve sokszor úgy éreztem, hogy nem tudok jó döntéseket hozni, mivel a webes világban egy adott probléma megoldására mindig sokféle különböző technológia és megoldás létezik, és nem mindig volt tiszta, hogy mi a jó választás, mibe éri meg beletanulni. Az Android alkalmazás fejlesztése ezek után egy felüdülés volt számomra, a Google útmutatói nagyon tisztán leírják a legjobb módszereket az adott problémák megoldására és az alkalmazás architektúra kialakítására. Emellett az új Jetpack Compose felületleíró technológia megismerése nagyon pozitív élmény volt számomra, és érdekes volt látni, hogy milyen irányba fejlődik az Androidos fejlesztés.

A webshop rendszer rengeteg továbbfejlesztési lehetőséggel rendelkezik, ezek közül úgy gondolom, hogy a lényegesebbek a webes kliens felhasználói felületének szebbé, interaktívabbá tétele, illetve a mobil kliensben az extra funkciók – webes fizetés, termék szűrők – implementálása.

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | „StackOverflow 2021 Developer Survey,” 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies. [Hozzáférés dátuma: 22. május 2022.]. |
| [2] | „Git,” [Online]. Available: https://git-scm.com/. [Hozzáférés dátuma: 22. május 2022.]. |
| [3] | GitHub, „GitHub kezdőoldal,” [Online]. Available: https://github.com/. [Hozzáférés dátuma: 6. december 2022.]. |
| [4] | „Visual Studio,” Microsoft, [Online]. Available: https://visualstudio.microsoft.com/. [Hozzáférés dátuma: 22. május 2022.]. |
| [5] | „ASP.NET Documentation,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-6.0. [Hozzáférés dátuma: 22. május 2022.]. |
| [6] | „Entity Framework Core,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/ef/core/. [Hozzáférés dátuma: 22. május 2022.]. |
| [7] | „Download SQL Server Management Studio (SSMS),” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15. [Hozzáférés dátuma: 22. május 2022.]. |
| [8] | Swashbuckle.AspNetCore. [Online]. Available: https://github.com/domaindrivendev/Swashbuckle.AspNetCore. [Hozzáférés dátuma: 22. május 2022.]. |
| [9] | „OpenAPI Specification,” [Online]. Available: https://swagger.io/specification/. [Hozzáférés dátuma: 22. május 2022.]. |
| [10] | J. Skinner, „FluentValidation,” [Online]. Available: https://docs.fluentvalidation.net/en/latest/. [Hozzáférés dátuma: 22. május 2022.]. |
| [11] | „AutoMapper,” [Online]. Available: https://automapper.org/. [Hozzáférés dátuma: 22. május 2022.]. |
| [12] | SixLabors, „ImageSharp,” [Online]. Available: https://sixlabors.com/products/imagesharp/. [Hozzáférés dátuma: 07. december 2022.]. |
| [13] | Barion, „Barion kezdőoldal,” [Online]. Available: https://www.barion.com/hu/. [Hozzáférés dátuma: 6. december 2022.]. |
| [14] | RestSharp, „RestSharp kezdőoldal,” [Online]. Available: https://restsharp.dev/. [Hozzáférés dátuma: 6. december 2022.]. |
| [15] | „Moq repository,” [Online]. Available: https://github.com/moq/moq4. [Hozzáférés dátuma: 6. december 2022.]. |
| [16] | „xUnit.net,” [Online]. Available: https://xunit.net/. [Hozzáférés dátuma: 6. december 2022.]. |
| [17] | Microsoft, „Azure kezdőoldal,” [Online]. Available: https://azure.microsoft.com/hu-hu/. [Hozzáférés dátuma: 6. december 2022.]. |
| [18] | „Visual Studio Code,” Microsoft, [Online]. Available: https://code.visualstudio.com/. [Hozzáférés dátuma: 22. május 2022.]. |
| [19] | „TypeScript,” Microsoft, [Online]. Available: https://www.typescriptlang.org/. [Hozzáférés dátuma: 22. május 2022.]. |
| [20] | „Angular,” Google, [Online]. Available: https://angular.io/. [Hozzáférés dátuma: 22. május 2022.]. |
| [21] | „Angular Material,” Google, [Online]. Available: https://material.angular.io/. [Hozzáférés dátuma: 22. május 2022.]. |
| [22] | „Material Design,” Google, [Online]. Available: https://material.io/design. [Hozzáférés dátuma: 22. május 2022.]. |
| [23] | PrimeTek Informatics, „PrimeNG kezdőoldal,” [Online]. Available: https://www.primefaces.org/primeng/. [Hozzáférés dátuma: 6. december 2022.]. |
| [24] | „Android Studio,” Google, [Online]. Available: https://developer.android.com/studio. [Hozzáférés dátuma: 27. november 2022.]. |
| [25] | Google, [Online]. Available: https://developer.android.com/jetpack. [Hozzáférés dátuma: 27. november 2022.]. |
| [26] | Google, „Dagger - Hilt,” [Online]. Available: https://dagger.dev/hilt/. [Hozzáférés dátuma: 27. november 2022.]. |
| [27] | Google, „Room,” [Online]. Available: https://developer.android.com/jetpack/androidx/releases/room. [Hozzáférés dátuma: 27. november 2022.]. |
| [28] | Google, „Jetpack Compose,” [Online]. Available: https://developer.android.com/jetpack/compose. [Hozzáférés dátuma: 27. november 2022.]. |
| [29] | Block, Inc., „OkHttp,” [Online]. Available: https://square.github.io/okhttp/. [Hozzáférés dátuma: 28. november 2022.]. |
| [30] | I. Block, „Retrofit,” [Online]. Available: https://square.github.io/retrofit/. [Hozzáférés dátuma: 28. november 2022.]. |
| [31] | Microsoft, „Architect Modern Web Applications with ASP.NET Core and Azure,” [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/. [Hozzáférés dátuma: 5. december 2022.]. |
| [32] | Microsoft, „Implement value objects,” [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/implement-value-objects. [Hozzáférés dátuma: 6. december 2022.]. |
| [33] | Google, „Guide to app architecture,” [Online]. Available: https://developer.android.com/topic/architecture. [Hozzáférés dátuma: 28. november 2022.]. |

Függelék

Kódrészlet a 4.5.3.1 fejezetből, amely a bejelentkezési logikát tartalmazza:

public async Task<UserViewModel> LoginAsync(LoginDto loginDto)

{

var user = await FindUserByEmailAsync(loginDto.Email);

var passwordMatches = await \_userManager.CheckPasswordAsync(user,   
 loginDto.Password);

if (!passwordMatches)

{

throw new ValidationAppException("Login attempt failed.", new[]

{

"Invalid user credentials."

});

}

var accessToken = await CreateAccessTokenAsync(user);

var refreshToken = CreateRefreshToken();

user.RefreshToken = refreshToken;

var userRoles = await \_userManager.GetRolesAsync(user);

//...

user.Cart = await \_cartRepository.GetUserCartAsync(user.Id);

if (loginDto.CartId.HasValue && loginDto.CartId != user.Cart.Id)

{

var cartBeforeLogin = await   
 \_cartRepository.GetCartByIdAsync(loginDto.CartId.GetValueOrDefault());

foreach (var oldCartItem in cartBeforeLogin.CartItems)

{

await \_cartRepository.AddCartItemAsync(user.Cart.Id, oldCartItem);

}

await \_cartRepository.DeleteCartAsync(cartBeforeLogin.Id);

}

await \_unitOfWork.SaveChangesAsync();

var userViewModel = \_mapper.Map<UserViewModel>(user);

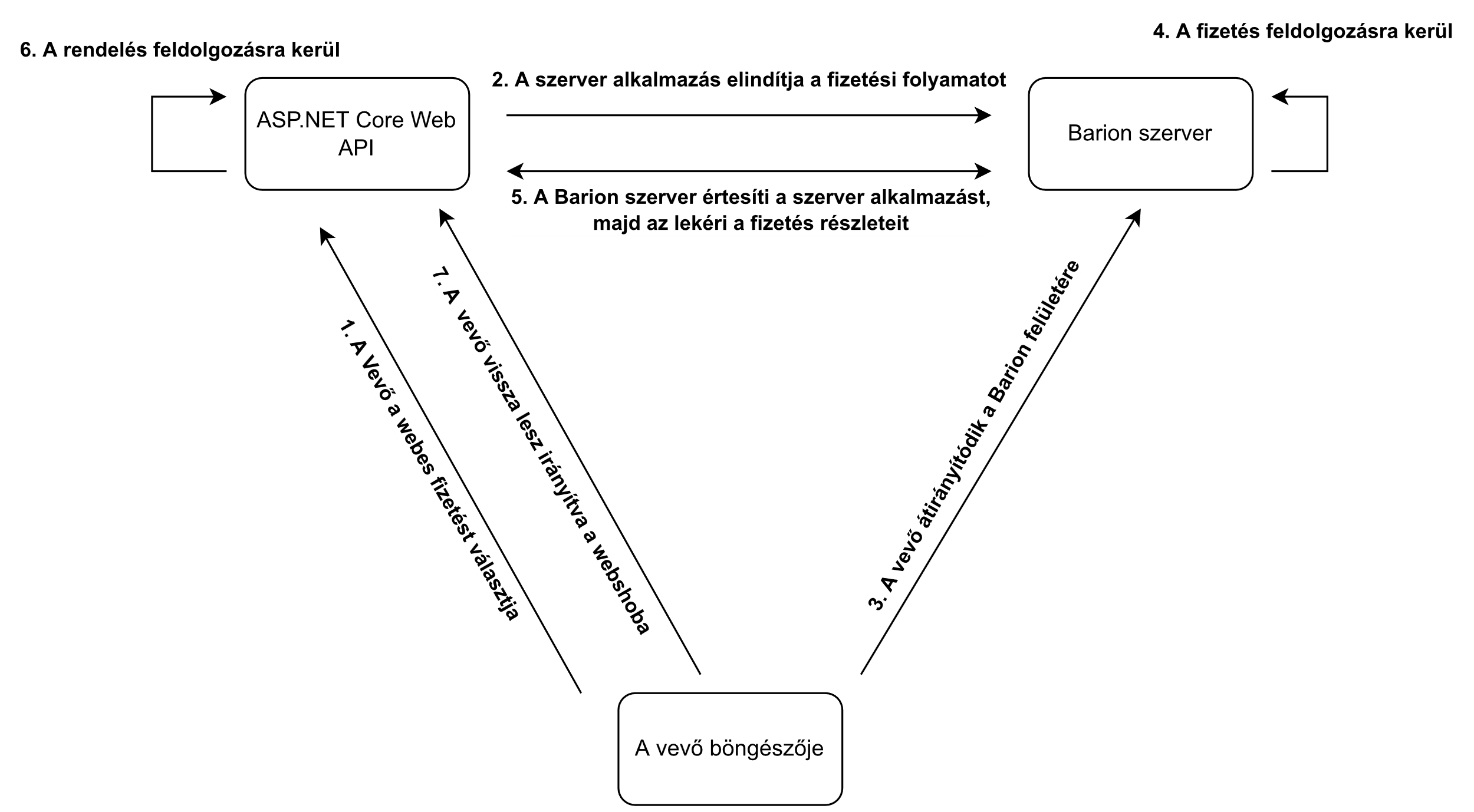
userViewModel.Role = role;

userViewModel.AccessToken = accessToken;

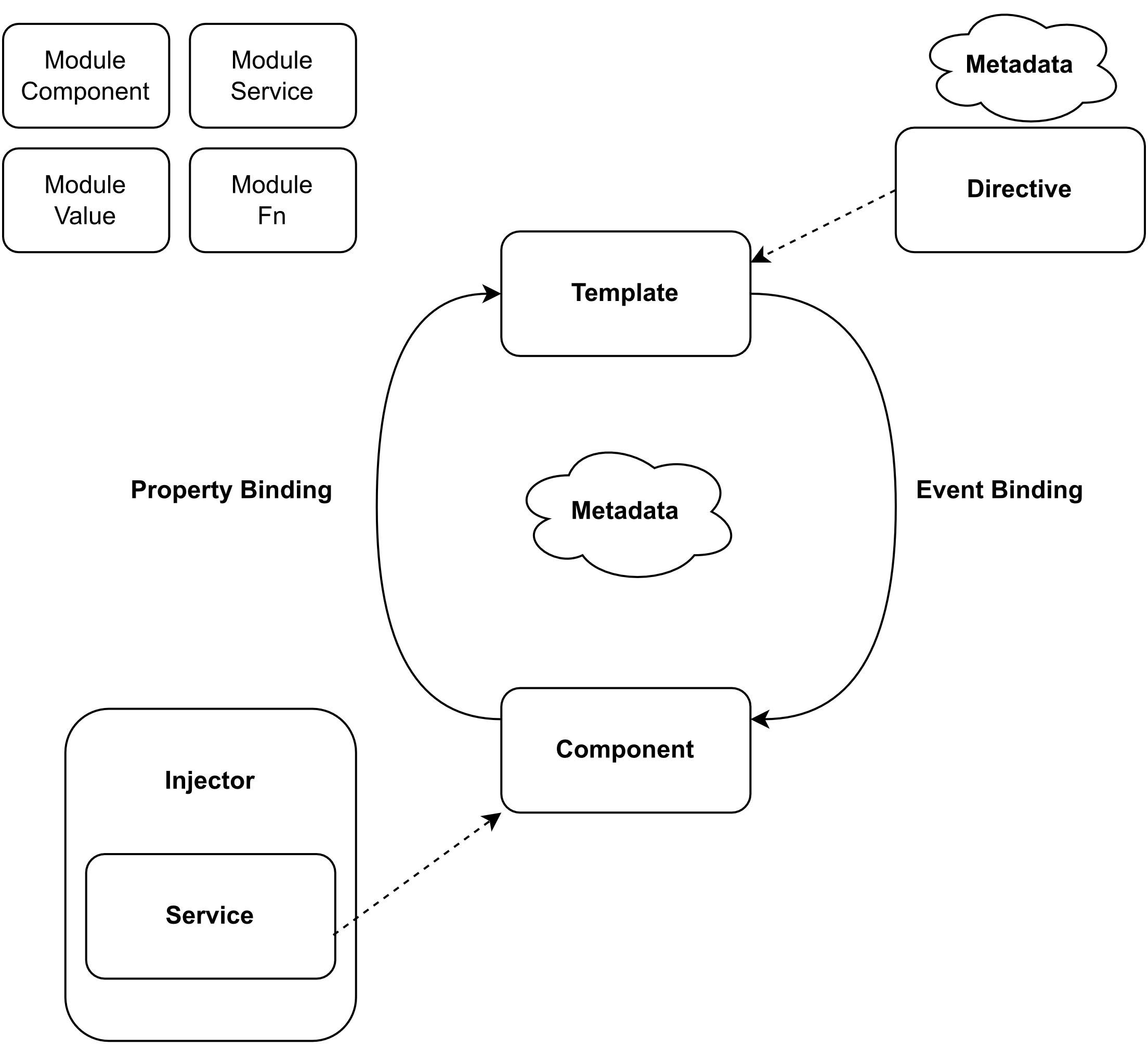
userViewModel.RefreshToken = refreshToken;

return userViewModel;

}



5. ábra - A webes fizetés folyamata

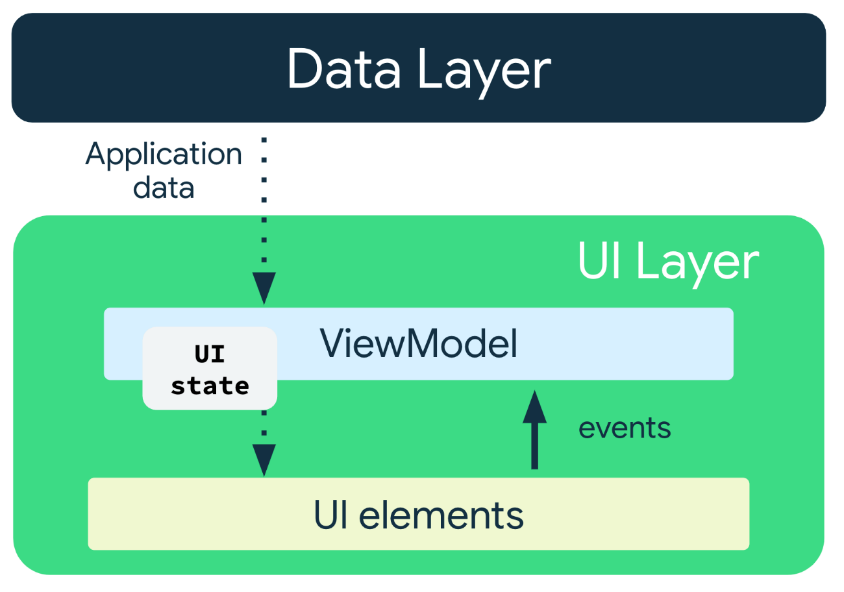


6. ábra - Az Angular alkalmazás architektúrája

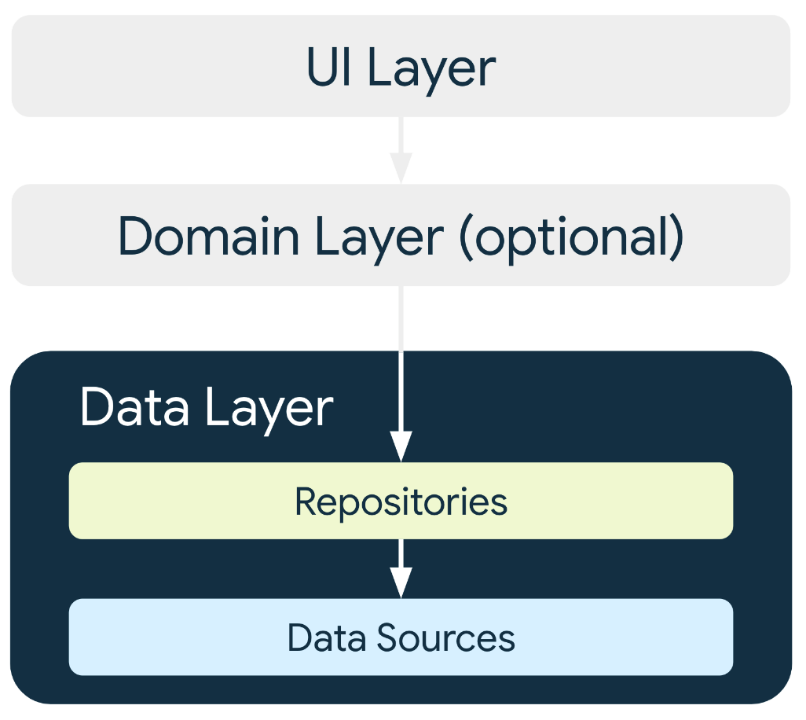
A képen szöveg látható

Automatikusan generált leírás

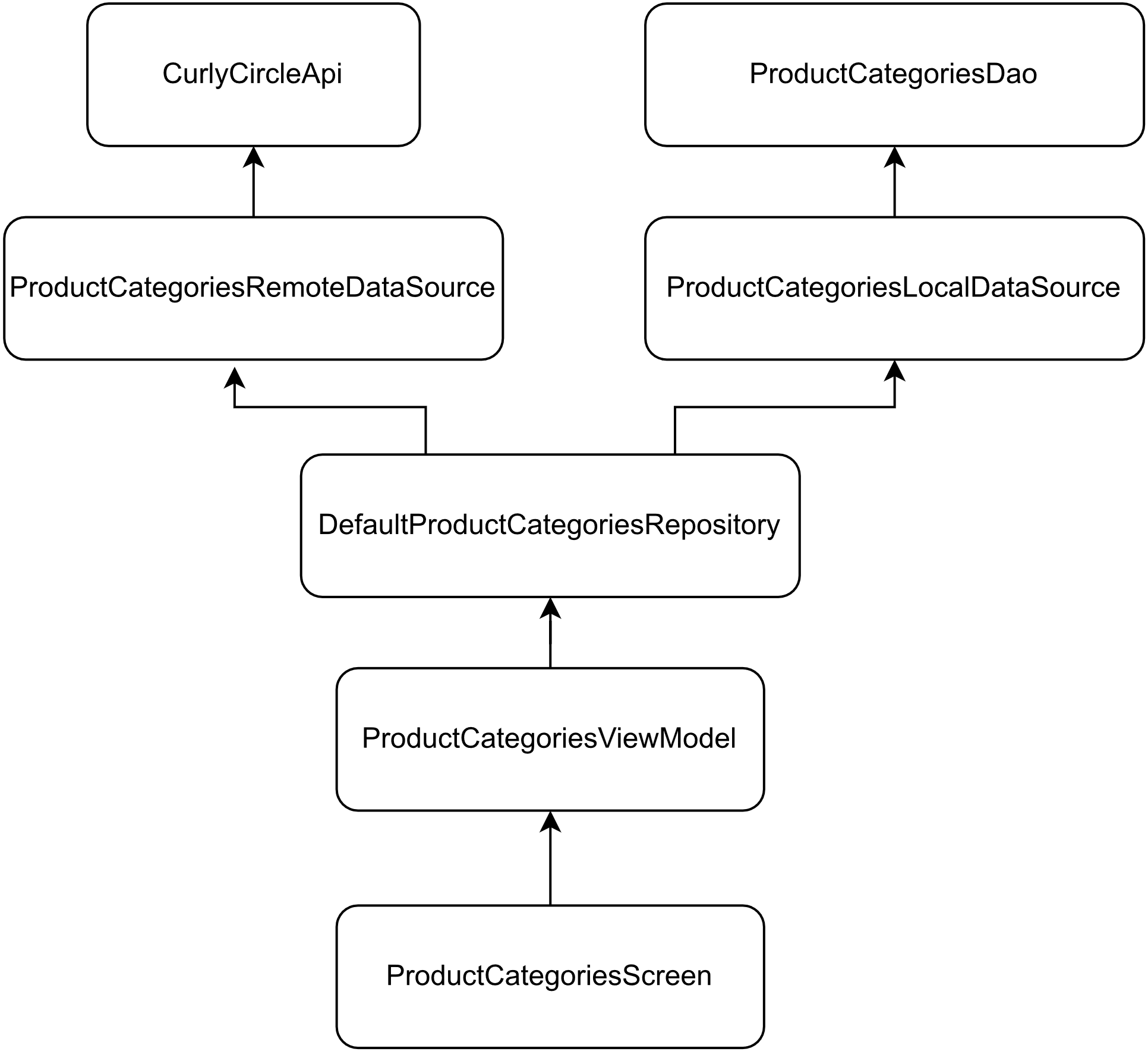
7. ábra - Képernyőkép a kosár komponensről



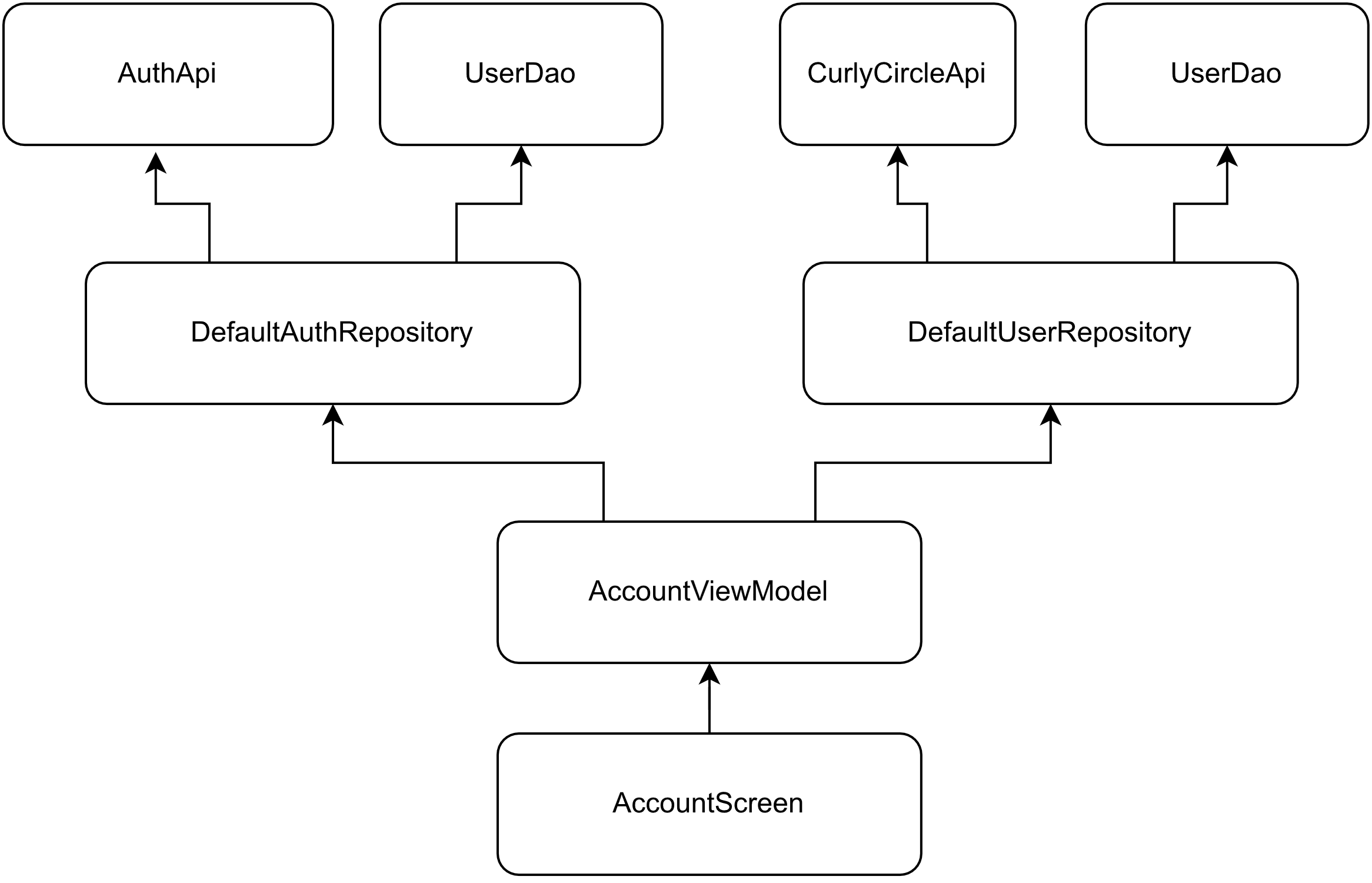
8. ábra - A Unidirectional Data Flow működése az architektúrában [33]



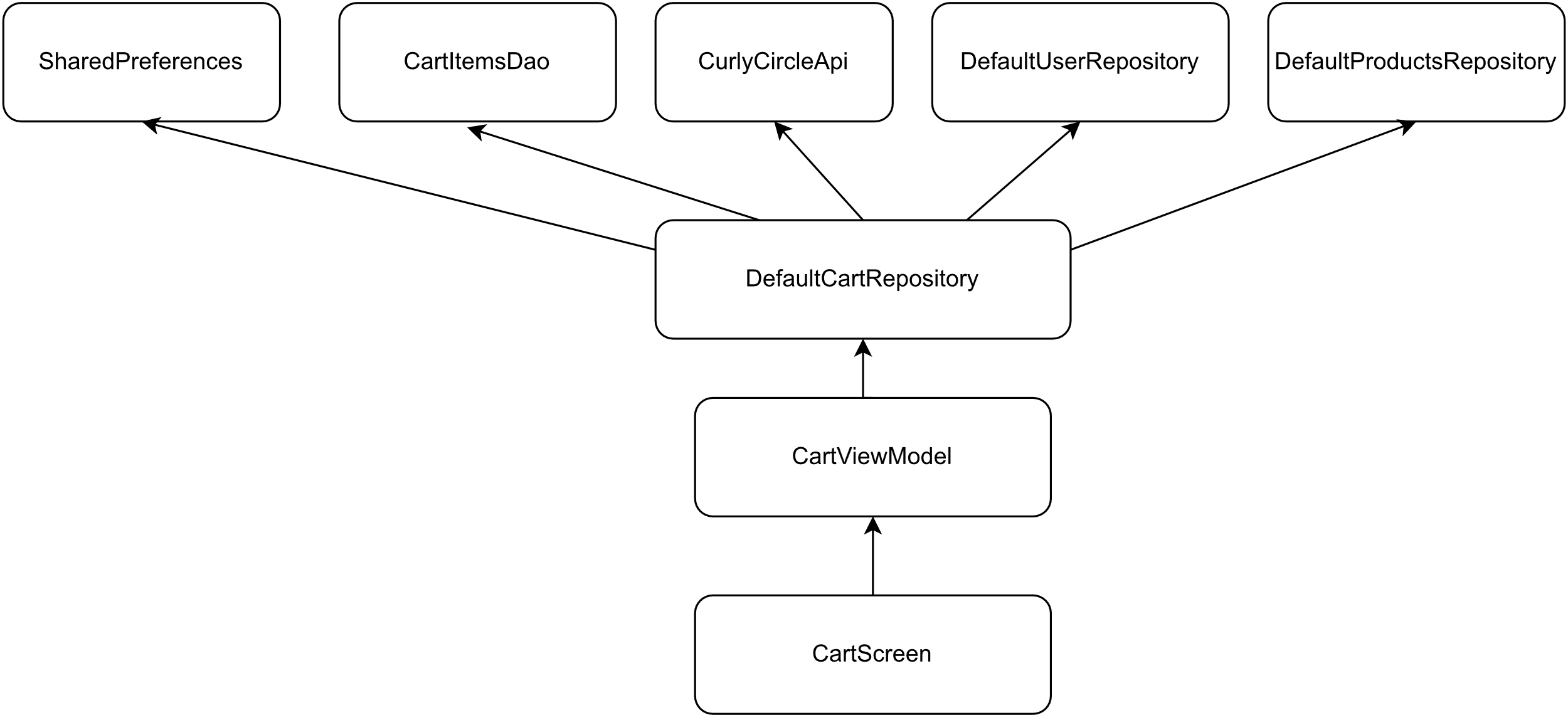
. ábra - Az adat réteg felépítése [33]



10. ábra - A termékkategóriák képernyő mögöttes architektúrája



11. ábra - Architektúra diagram a Felhasználói fiók képernyőhöz tartozó osztályokról



12. ábra - A Kosár képernyőhöz tartozó osztályok architektúrája