ÓBUDAI EGYETEM

Keleti Károly Gazdasági Kar
Vállalkozásfejlesztés és Infokommunikációs Intézet

SZAKDOLGOZAT

ÓE-KGK
2023

Hallgató neve: **Kovács Máté**
Hallgató törzskönyvi száma: **T007145/FI12904/G**

ÓBUDAI EGYETEM

Keleti Károly Gazdasági Kar

SZAKDOLGOZAT FELADATLAP

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Óbudai Egyetem
Keleti Károly Gazdasági Kar
Vállalkozásfejlesztés és Infokommunikációs Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: Kovács Máté
Szakdolgozat száma: [redacted]
Törzskönyvi száma: T007145/FI12904/G **Neptun kódja:** [redacted]

Szak: Gazdaságinformatikus
Specializáció: Vállalatinformatikai specializáció

A dolgozat címe: Flottakezelő és foglalást lebonyolító webalkalmazás fejlesztése
A dolgozat címe angolul: Fleet management and booking web application development
A feladat részletezése:
1. Mutassa be a végrehajtandó fejlesztést!
2. Mutassa be, miért döntöttek a saját flottakezelő szoftver elkészítése mellett!
3. Mutassa be a fejlesztés lépéseit és az elkészült szoftvert!
4. Vázolja fel az elkészült szoftver továbbfejlesztési lehetőségeit!

Intézményi konzulens neve: Fehér-Polgár Pál

A kiadott téma elévülési határideje:2024. 12. 15.
Beadási határidő: 2023. 12. 15.

A szakdolgozat: Nem titkos.

Kiadva: Budapest, 2023. 11. 16.

.....
Intézetigazgató

A dolgozatot beadásra alkalmasnak találom.

Fehér – Polgár P.
.....
belső konzulens

.....
külső konzulens

ÓBUDAI EGYETEM

Keleti Károly Gazdasági Kar

HALLGATÓI NYILATKOZAT

Alulírott Kovács Máté (Neptunkód: [redacted]) hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja, a titkosításra vonatkozó esetleges megkötések mellett.

Budapest, 2023.11.30.

---

hallgató aláírása

TARTALOMJEGYZÉK

ÁBRAJEGYZÉK

1. BEVEZETÉS

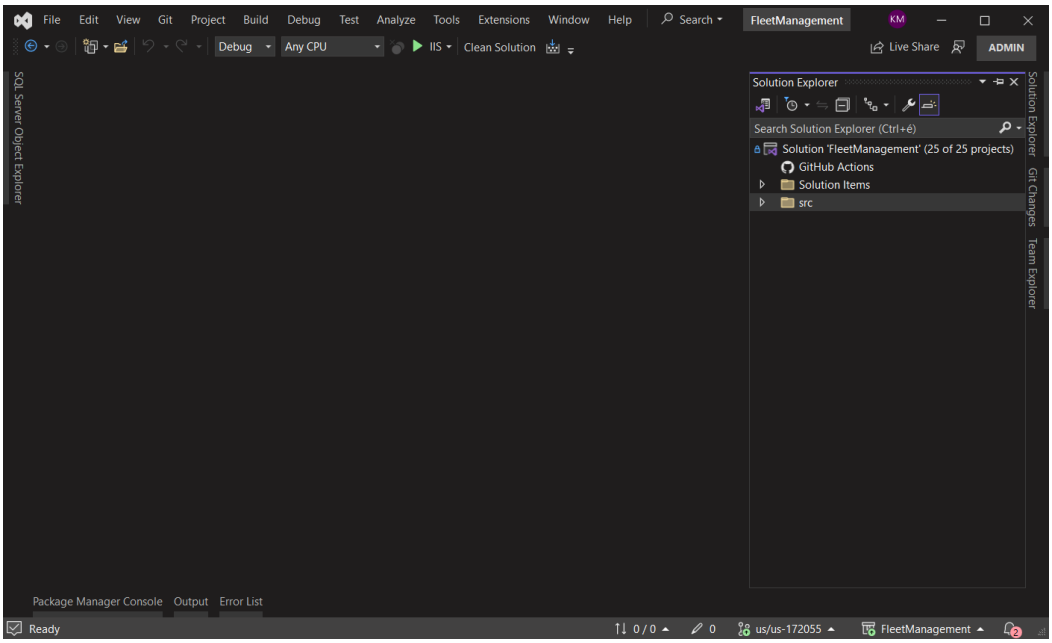
Az informatikai fejlődés folyamatosan új lehetőségeket teremt a vállalatok és szervezetek számára a hatékonyabb működés és a jobb szolgáltatásnyújtás terén. Az üzleti folyamatok digitalizációja és az online platformok elterjedése olyan kihívások elé állítja a vállalkozásokat, melyekre válaszul új és innovatív megoldásokra van szükség. Az egyik ilyen terület a flottakezelés és foglalás lebonyolítás, ahol a személygépkocsik vagy flották hatékony nyomon követése és a foglalások kezelése kiemelkedő jelentőségű. Ezen a területen egyre növekvő igény mutatkozik olyan webalkalmazások iránt, amelyek lehetővé teszik a flották egyszerű és hatékony kezelését, valamint a járművek előzetes foglalását és azok adminisztrációját. Egy ilyen alkalmazás segítségével a vállalatok könnyedén nyomon követhetik flottájuk állapotát, karbantartási igényeit, üzemanyag-fogyasztását és egyéb fontos paramétereit. Emellett az alkalmazás lehetőséget nyújt a felhasználóknak az online foglalásra és lemondásra, a járművek rendelkezésre állásának ellenőrzésére, valamint az ügyfélkapcsolatok optimalizálására. Ez igaz a KUKA Hungária Kft. nevű vállalatra is, ahol szakmai gyakorlatomat töltöttem és ahol találkoztam ezzel a témakörrel. Szerencsés módon pont egy projekt kezdésbe csöppentem bele, így átélhettem és megtudhattam milyen egy projektet az elejétől az átadásig véghez vinni. Ezt a projektet egy négy fős csapat vitte, kettő medior fejlesztő, egy junior fejlesztő és én, mint gyakornok voltunk a csapat tagjai. Mivel ez a projekt egy cégen belüli igény volt, így sokkal enyhébb követelmények és elvárások voltak felénk, mintha egy külsős megrendelés lett volna. Ez a projekt 2023 nyarán befejeződött, de még átadásra nem került. Ennek a szakdolgozatnak a célja, egy ilyen webalkalmazás tervezésének és fejlesztésének a bemutatása. Ennek a bemutatásának a keretein belül beszélni fogok a fejlesztői környezetről, a szoftvertervezési metodológiáról, amin az alkalmazás alapszik, illetve a különböző tervezési mintákról, amiket használtunk a fejlesztés során. A szakdolgozatom első részében szeretném bemutatni a fejlesztői környezetet. A második felében pedig a szoftvertervezési metodológiát és a tervezési mintákat. Ezután a fejlesztés lépéseit és majd az elkészült szoftvert fogom bemutatni. A szakdolgozatomat egy összefoglalással és egy a jövőbeli lehetséges fejlesztések ismertetésével fogom zárni.

Alapvetően a cégen belül azért döntöttek a saját szoftver elkészítésében és nem egy külsős cég szolgáltatásait vették igénybe, mivel már létezett egy régebbi verzió ehhez a felülethez. Egy régebbi fejlesztés eredménye volt ez a flottakezelő alkalmazás, az új igény viszont az volt, hogy szeretnének egy modernebb, letisztultabb verziót és felületet készíteni, ahol a foglalásokat is könnyedén le lehet bonyolítani a felhasználók szemszögéből. Illetve egy harmadik funkciót/szolgáltatást is szerettek volna beleépíteni ebbe az új projektbe, viszont ennek a része nem készült el a szakdolgozatom témájának leadási idejéig, emiatt erről nem is lesz szó a fejlesztés bemutatásának során, de megemlíteni megszeretném, mint lehetséges jövőbeli fejlesztés. A másik indok arról, hogy miért is nem külsős szolgáltatást vettek igénybe a projekt kapcsán, az az, hogy igy sokkal költséghatékonyabb megoldást tudtak találni, ami mind a cégnek, mind pedig a felhasználóknak egyaránt kedvezőbb végeredményt jelentett.

2. A FEJLESZTŐI KÖRNYEZET BEMUTATÁSA

2.1. Microsoft Visual Studio

Fejlesztőkörnyezetnek a már általam jól ismert Microsoft Visual Studio 2022-t használtam. A Microsoft Visual Studio egy integrált fejlesztői környezet másnéven Integrated Development Environment (IDE), amelyet a Microsoft tervezett és fejlesztett. Elsősorban szoftveralkalmazások létrehozására szolgál, beleértve az asztali alkalmazásokat, webalkalmazásokat, mobilalkalmazásokat stb. A Visual Studio eszközök és szolgáltatások átfogó készletét kínálja, amelyek leegyszerűsítik a szoftverfejlesztési folyamatot, megkönnyítve a fejlesztők számára a kód írását, hibakeresését és tesztelését. Az 1. ábrán láthatjuk ennek a felületét. (https://learn.microsoft.com/hu-hu/visualstudio)



1. ábra: Microsoft Visual Studio felülete
Forrás: saját kép

A Microsoft Visual Studio főbb funkciói a következők:

Kódszerkesztő: A Visual Studio robusztus kódszerkesztőt kínál olyan funkciókkal, mint a szintaktikai kiemelés, a kódkiegészítés és a kódnavigáció, hogy segítse a fejlesztőket a hatékony kódírásban.

Hibakereső eszközök: Hatékony hibakereső eszközöket biztosít, amelyek lehetővé teszik a fejlesztők számára, hogy azonosítsák és kijavítsák a kódjukban előforduló

problémákat, beállítsanak töréspontokat, megvizsgálják a változókat, és végig lépjenek a kód végrehajtásán.

Integrált verziókezelés: A Visual Studio integrálható olyan népszerű verzióvezérlő rendszerekkel, mint például a Git, és szolgáltatásokat kínál a forráskód-tárolók kezeléséhez, beleértve az elágazást, egyesítést és a kód-együttműködést.

Projektsablonok: A fejlesztők számos projektsablonból indíthatnak új projekteket, amelyek előre konfigurált struktúrát biztosítanak a különböző típusú alkalmazásokhoz, például Windows-alkalmazásokhoz, webes alkalmazásokhoz stb.

Kódelemzés és refaktorálás: A Visual Studio olyan kódelemző eszközöket tartalmaz, amelyek segítenek azonosítani a kódminőséggel kapcsolatos problémákat, és javaslatokat tesznek a kód újra faktorálására a karbantarthatóság és a teljesítmény javítása érdekében.

Bővíthetőség: A Visual Studio nagymértékben bővíthető, lehetővé téve a fejlesztők számára, hogy testreszabják és javítsák az IDE-t a Visual Studio Marketplace bővítményeivel és kiegészítőivel.

Többnyelvű támogatás: A programozási nyelvek széles skáláját támogatja, beleértve a C#, C++, F#, Visual Basic, Python, JavaScript és sok más nyelvet.

Platformok közötti fejlesztés: A Visual Studio eszközöket biztosít alkalmazások fejlesztéséhez különféle platformokra, beleértve a Windows, macOS, iOS, Android és webes alkalmazásokat.

Felhőintegráció: A fejlesztők könnyedén integrálhatják projektjeiket a Microsoft Azure-ral és más felhőszolgáltatásokkal az alkalmazások üzemeltetéséhez, telepítéséhez és méretezéséhez.

Együttműködés és csapatfejlesztés: A Visual Studio támogatja a csoportos együttműködést olyan funkciókon keresztül, mint a kódellenőrzés, a munkaelemek nyomon követése és az Azure DevOps-szal való integráció. (https://learn.microsoft.com/hu-hu/visualstudio)

Összességében a Microsoft Visual Studio egy sokoldalú és széles körben használt fejlesztői környezet, amely a szoftverfejlesztési igények széles skáláját elégíti ki különféle platformokon és programozási nyelveken.

2.2. A C# programozási nyelv

A C# (ejtsd: "C-sharp") egy modern, objektum-orientált programozási nyelv, amelyet a Microsoft fejlesztett ki a 2000-es évek elején a .NET-keretrendszer-kezdeményezés részeként. Úgy tervezték, hogy sokoldalú és hatékony nyelv legyen az alkalmazások széles skálájának elkészítéséhez, a Windows asztali alkalmazásoktól a webalkalmazásokig és mobilalkalmazásokig. A 2. ábrán láthatunk egy egyszerű kódsort C# nyelvben íródva. (Robert Sheldon 2022)

2. ábra: C# nyelvű kódsor
Forrás: learn.microsoft.com

A C# főbb jellemzői a következők:

Objektum-orientált: A C# egy objektum-orientált nyelv, ami azt jelenti, hogy az objektumok és osztályok használatát hangsúlyozza a kód rendszerezésére és strukturálására. Lehetővé teszi újra felhasználható és moduláris kódkomponensek létrehozását.

Típusbiztonság: A C# erős típusellenőrzést biztosít fordítási időben, csökkentve a futásidejű hibák esélyét és növelve a kód megbízhatóságát.

Platformfüggetlenség: Míg eredetileg Windowsra fejlesztették ki, a C# platformfüggetlenné fejlődött. Különféle környezetekben használják, beleértve a Windowst, a Linuxot és a macOS-t, így alkalmas a többplatformos fejlesztésre.

Szemétgyűjtés: A C# magában foglalja az automatikus memóriakezelést egy szemétgyűjtőn keresztül, amely segít kezelni a memóriafoglalást és -felszabadítást, csökkentve a memóriaszivárgás kockázatát, és leegyszerűsíti a memóriakezelést a fejlesztők számára.

Szabványos könyvtárak: A C# a kiterjedt .NET-keretrendszer osztálykönyvtárak előnyeit élvezi, amelyek előre elkészített funkciók és osztályok széles skáláját kínálják a gyakori programozási feladatokhoz, így hatékonyabbá téve a fejlesztést.
Modern nyelvi jellemzők: A C# minden új verzióval folyamatosan fejlődik, és olyan modern nyelvi funkciókat tartalmaz, mint az aszinkron/várakozás az aszinkron programozáshoz, a LINQ (Language Integrated Query) az adatok lekérdezéséhez és még sok más.
Együttműködés: A C# erősen támogatja a más nyelvekkel, különösen a C-vel és a C++-szal való együttműködést, lehetővé téve a fejlesztők számára, hogy kihasználják a meglévő kódot vagy más nyelveken írt könyvtárakat.
Biztonság: A C# olyan funkciókat tartalmaz, mint a kivételkezelés és a hozzáférés-vezérlés, amelyek segítenek robusztus és biztonságos alkalmazások írásában.
IDE-támogatás: A Microsoft Visual Studio a C#-fejlesztés elsődleges integrált fejlesztői környezete (IDE), amely eszközök gazdag készletét kínálja a C#-kód írásához, hibakereséséhez és teszteléséhez. (https://learn.microsoft.com/en-us/dotnet/csharp)

Összességében a C# egy sokoldalú, modern programozási nyelv, amely a termelékenységéről, az erős fejlesztőeszközökről és az alkalmazások széles skálájáról ismert.

2.3. .NET keretrendszer

A .NET-keretrendszer a Microsoft által kifejlesztett szoftverkeretrendszer Windows-alkalmazások készítésére és futtatására. Átfogó és következetes programozási modellt biztosít számos alkalmazás fejlesztéséhez, beleértve a Windows asztali alkalmazásokat, webalkalmazásokat, webszolgáltatásokat és még sok mást. A .NET-keretrendszer nagy osztálykönyvtárat és több programozási nyelv támogatását is tartalmazza, így sokoldalú platform a szoftverfejlesztéshez. A .NET Framework eszköztára a szoftverfejlesztés szinte minden aspektusát (kliens-, illetve szerveroldali megoldások, adatbázisok kezelése, játékfejlesztés stb.) lefedi. (https://learn.microsoft.com/en-us/dotnet/framework)

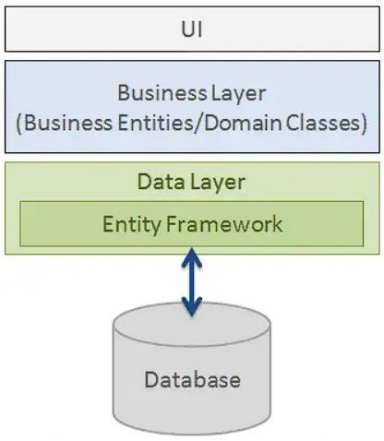
A .NET-keretrendszer két fő összetevője a Common Language Runtime és a .NET Framework osztálykönyvtár.

A Common Language Runtime (CLR) a futó alkalmazásokat kezelő végrehajtó motor. Olyan szolgáltatásokat nyújt, mint a szálkezelés, a szemétgyűjtés, a típusbiztonság, a kivételkezelés és még sok más. Az Osztálykönyvtár API-k és típusok készletét kínálja a közös funkciókhoz. Típusokat biztosít karakterláncokhoz, dátumokhoz, számokhoz stb. Az Osztálykönyvtár API-kat tartalmaz fájlok olvasásához és írásához, adatbázisokhoz való csatlakozáshoz, rajzoláshoz stb. (https://dotnet.microsoft.com/en-us/learn/dotnet)

Összefoglalva, a .NET-keretrendszer egy sokoldalú és széles körben használt fejlesztői platform Windows-alkalmazások létrehozására, míg az újabb ".NET" platform kiterjeszti hatókörét a többplatformos és a felhőalapú fejlesztésekre is.

2.4. Entity Framework

Az Entity Framework (EF) egy objektum-relációs leképező (ORM) keretrendszer, amelyet a Microsoft fejlesztett ki a .NET keretrendszer részeként. Lehetővé teszi a fejlesztők számára, hogy relációs adatbázisokkal dolgozzanak .NET-objektumok használatával, és magasabb szintű, objektum-orientált módot biztosít az adatbázisokkal való interakcióhoz, ahelyett, hogy nyers SQL-lekérdezéseket írnának. Az Entity Framework leegyszerűsíti az adatbázis-hozzáférést, és csökkenti az adatműveletekhez szükséges alapkód mennyiségét. Általában webalkalmazásokban, asztali alkalmazásokban és más olyan szoftverprojektekben használják, ahol tartós adatra van szükség. A következő ábra azt szemlélteti, hogy az Entity Framework hová illeszkedik az alkalmazásunkon belül. (https://www.entityframeworktutorial.net)



3. ábra: Entity Framework réteg
Forrás: entityframeworktutorial.net

3. AZ ARCHITEKTÚRA BEMUTATÁSA

3.1. Domain-Driven Design

A webalkalmazás tervezésének alapját a Domain-Driven Design (DDD) képezi, amely egy szoftvertervezési metodológia és gyakorlat, melyet Eric Evans vezetett be könyvében, a "Domain-Driven Design: Tackling Complexity in the Heart of Software" című művében. A DDD egy olyan megközelítés, amely a szoftvertervezést az üzleti terület központjába állítja, és azt hangsúlyozza, hogy a szoftvernek meg kell felelnie az üzleti igényeknek és a domain (terület) specifikus szabályoknak. (Eric Evans 2003, pp. 24-40.)

A Domain-Driven Design néhány kulcsfontosságú elemét érdemes megemlíteni:

Domain model: A DDD középpontjában a területi modell áll, ami egy absztrakt, strukturált reprezentációja az üzleti domainnek. Ez a modell leírja az üzleti entitásokat, értékeket, szabályokat és kapcsolatokat. A webalkalmazás tervezése során a területi modell segít az alkalmazás struktúrájának és komponenseinek meghatározásában.

Ubiquitous Language (Mindennapi nyelv): A DDD elvártja, hogy az üzleti és fejlesztői csapatok között egy közös, pontos és konzisztens nyelvezet legyen, amit mindenki ért és használ. Ez segít megelőzni a kommunikációs hibákat és biztosítja, hogy az üzleti szabályok megfelelően kerüljenek be az alkalmazásba.

Aggregate Roots (Gyökér entitások): Az Aggregate Roots olyan kulcsfontosságú entitások a területi modellben, amelyek irányítják a hozzájuk kapcsolódó entitásokat és üzleti szabályokat. Ezek az entitások felelnek az adatok konzisztenciájának és integritásának fenntartásáért.

Repository (Gyűjtemény): A Repository minta segítségével a DDD lehetővé teszi az alkalmazásnak, hogy absztrakt módon hozzáférjen az adatokhoz, anélkül, hogy közvetlenül a tárolórendszerekhez lenne szükségük. Ez javítja az alkalmazás tesztelhetőségét és karbantarthatóságát.

Bounded Context: Az üzleti domain gyakran összetett és széttagolódott lehet. A DDD azt javasolja, hogy különböző bounded contexteket határozzunk meg az alkalmazásban,

amelyek az üzleti területek részletes leírásait tartalmazzák. Ez segít elkerülni az ütközéseket és a félreértéseket a területi modellben. (Eric Evans 2003, pp. 24-301.)

A Domain-Driven Design tehát segít az alkalmazás tervezésében és fejlesztésében azzal, hogy a szoftvert az üzleti igényekhez és a Domain specifikus szabályokhoz igazítja. Ezáltal az alkalmazás jobban megfelel az üzleti elvárásoknak, könnyebben karbantartható és rugalmasabb lesz a változásokra való reagálás szempontjából. A webalkalmazás tervezésekor a Domain-Driven Design alkalmazása lehetővé teszi a hatékonyabb és jobban strukturált szoftverfejlesztést.

3.1.1. Domain model

A Domain Model egy olyan tervezési minta vagy módszer, amelyet szoftvertervezés során alkalmaznak a szoftverrendszerek tervezésére és strukturálására. A Domain Model egy olyan absztrakt modell vagy reprezentáció, amely leírja egy adott üzleti terület alapvető fogalmait, entitásait és azok közötti kapcsolatokat. A célja az, hogy a fejlesztőknek és a projekt résztvevőinek segítsen megérteni és kommunikálni az adott területre vonatkozó szabályokat és fogalmakat, valamint egy közös nyelvet kínáljon a fejlesztők és az üzleti szakemberek között. A legrosszabb esetben az üzleti logika nagyon összetett lehet. A szabályok és a logika számos különböző viselkedési esetet és ferdeséget írnak le, és az objektumokat ennek a bonyolultságnak a kezelésére tervezték. A Domain Model összekapcsolt objektumok hálóját hozza létre, ahol minden objektum valamilyen értelmes egyént képvisel. Ha egy Domain Modelt elhelyezünk egy alkalmazásban, akkor egy egész réteg objektumot kell beilleszteni, amelyek azt az üzleti területet modellezik, amelyben dolgozunk. Találhatunk olyan objektumokat, amelyek utánozzák az üzletben lévő adatokat, és olyan objektumokat, amelyek rögzítik az üzlet által használt szabályokat. A Domain Model gyakran alkalmazott tervezési minta az objektumorientált szoftvertervezésben. Az entitásokat osztályokként vagy objektumokként ábrázolja, amelyek tulajdonságokkal és viselkedésekkel rendelkeznek. Az entitások közötti kapcsolatok pedig például az öröklődés, az asszociációk vagy az aggregációk révén jelennek meg. A Domain Model különösen hasznos azokban a projektekben, amelyek összetett üzleti logikát és fogalmi hálózatot igényelnek. Az egyes projektekben a Domain

Modelt az adott terület specifikus igényekhez és követelményekhez igazítva alkalmazzák. (Martin Flower 2002, pp. 116-122.)

„*An object model of the domain that incorporates both behavior and data.*”

- Martin Flower

Tehát a Domain-nek egy olyan modellje objektumokkal, amely magába foglal működést is és adatot is. Mi is az a Domain? A Domain az gyakorlatilag az, amivel az üzlet foglalkozik, vagy amihez az üzlet ért vagy éppen, ami a témája egy adott projektnek. A modellt úgy tudjuk leírni, hogy a valóságnak olyan részeit reprezentáljuk, amik a mi szempontunkból fontosak vagy hasznosak.

A Domain model egy reprezentációja jelentőséggel bíró fogalmaknak arra a területre vonatkozóan, amit modelleznünk kell egy szoftverben.

A következő ábrán bemutatom, hogy is néz ki egy ilyen Domain model.

```
public class CarType : Entity<int>, IAggregateRoot
{
    3 references | Kovacs, Mate, 241 days ago | 1 author, 1 change
    public string CarTypeName {get; set;}

    1 reference | Kovacs, Mate, 241 days ago | 1 author, 1 change
    public CarType(string cartypeName)...

    1 reference | 0 changes | 0 authors, 0 changes
    public static async Task<CarType> Add(UserAggregate.AdminUser adminUser, string cartypeName)...

    1 reference | 0 changes | 0 authors, 0 changes
    public async Task Update(UserAggregate.AdminUser adminUser, string carTypeName)...

    1 reference | Oliver Nagy, 59 days ago | 2 authors, 3 changes
    public async Task Delete(CarType carType, AdminUser adminUser)...
}
```

4. ábra: Autó típus, mint Domain model
Forrás: saját kép

Amit itt látunk az az, hogy az Autó típus az egy dolog. Ezeket a dolgokat általában osztályokkal reprezentáljuk, ezen modellezük őket. Látjuk, hogy új autó típust tudunk létrehozni, ha tudjuk az autó típus nevét, amit szeretnénk felvenni a rendszerbe. Azt is látjuk, hogy már egy meglévő autó típust tudunk szerkeszteni, illetve törölni is a rendszerből, ha aki ezeket a műveleteket elakarja végezni rendelkezik Admin jogosultsággal. Amit itt látunk az konkrétan az, amit az üzlet igényként elvárt tőlünk. Ha

felolvassuk ezt a kódot, akkor az meg fog felelni az üzleti igényeknek. Ez a Domain model rész még nem tartalmaz adatbázis kapcsolatokat. Az, hogy ez a Domain model minél közelebb kerülhessen egy szoftverhez, ahhoz a tervezési minták fognak segíteni. Ehhez az első és legfontosabb lépés az adatbázis kapcsolat.

3.2. Tervezési minták (Design patterns)

A Design pattern, vagyis tervezési minta, egy ismert és bevált megoldási séma vagy sablon a szoftvertervezésben és -fejlesztésben. A tervezési minták olyan általános problémákra kínálnak megoldásokat, amelyek gyakran merülnek fel a szoftvertervezés során. Ezek a sablonok és megoldási séma segítik a fejlesztőket abban, hogy hatékonyan és strukturáltan tervezzenek, kódoljanak, és karbantartsanak szoftverrendszereket. (Dr. Ferenc Rudolf 2011, pp. 34-36.)

3.2.1. Repository pattern

A Repository Pattern a szoftverfejlesztésben általánosan használt tervezési minta, amely elválasztja az adatbázisból vagy bármely adatforrásból adatokat beolvasó logikát az alkalmazás többi részétől. Módot biztosít az adatokhoz való hozzáféréshez, miközben elválasztja a mögöttes adattárolási és visszakeresési mechanizmusokat. (https://learn.microsoft.com/en-us/dotnet/architecture)

A Repository egy osztály, ami magába foglalja az adatforrások eléréséhez szükséges logikát. Lehetővé teszik, hogy a perzisztencia függjön a Domain Modeltől, de a Domain Model létezzen és működjön bármilyen perzisztencia nélkül. (Ian Sommerville 2011, pp. 189-193.)

A Repository pattern első része a Repository interface, ez a Domain Modelnek a része. Ezzel a Domain Model kiköti, hogy milyen perzisztencia műveletek értelmesek az adott modellen. Ez az interface még nem függ a perzisztencia megvalósításától. A következő ábrán ennek működését fogom bemutatni.

```
8 references | Kovacs, Mate, 235 days ago | 1 author, 3 changes
public interface ICarTypeRepository : IRepository<CarType>
{
    2 references | Kovacs, Mate, 241 days ago | 1 author, 1 change
    CarType Add(CarType carType);
    3 references | Kovacs, Mate, 235 days ago | 1 author, 2 changes
    Task<CarType> GetCarType(int carTypeId);
    2 references | Kovacs, Mate, 241 days ago | 1 author, 1 change
    void Update(CarType carType);
    2 references | Kovacs, Mate, 241 days ago | 1 author, 1 change
    void Delete(CarType carType);

}
```

5. ábra: Repository interface
Forrás: saját kép

Itt látjuk, hogy ez a Repository interface az autók típusához tartozik. Ez definiálja, hogy le lehessen kérni egy autó típust egy Id alapján, hozzá lehet adni egy új típust egy típus név alapján, illetve szerkeszteni és törölni is lehet egy már létező autó típust a rendszerből.

A következő rész a Repository patternnek az implementáció. Itt a Repository interface-t egy Entity Framework DbContext segítségével valósítjuk meg. A következő ábrán ezt szemléltetem is.

```
public class CarTypeRepository : ICarTypeRepository
{
    private readonly AdministrationContext _context;
    67 references | Kovacs, Mate, 236 days ago | 1 author, 2 changes
    public IUnitOfWork UnitOfWork => _context;
    0 references | Kovacs, Mate, 236 days ago | 1 author, 2 changes
    public CarTypeRepository(AdministrationContext context)...
    2 references | Kovacs, Mate, 236 days ago | 1 author, 2 changes
    public CarType Add(CarType carType)
    {
        return _context.CarTypes.Add(carType).Entity;
    }
    3 references | Kovacs, Mate, 234 days ago | 1 author, 4 changes
    public async Task<CarType> GetCarType(int carTypeId)
    {
        var cartype = await _context.CarTypes.FirstOrDefaultAsync(x => x.Id == carTypeId);
        if (cartype == null)
        {
            cartype = _context.CarTypes.Local.FirstOrDefault(x => x.Id == carTypeId);
        }
        return cartype;
    }
    2 references | Kovacs, Mate, 236 days ago | 1 author, 2 changes
    public void Update(CarType carType)
    {
        _context.Entry(carType).State = Microsoft.EntityFrameworkCore.EntityState.Modified;
    }
    2 references | Kovacs, Mate, 236 days ago | 1 author, 2 changes
    public void Delete(CarType carType)
    {
        _context.Remove(carType);
    }
```

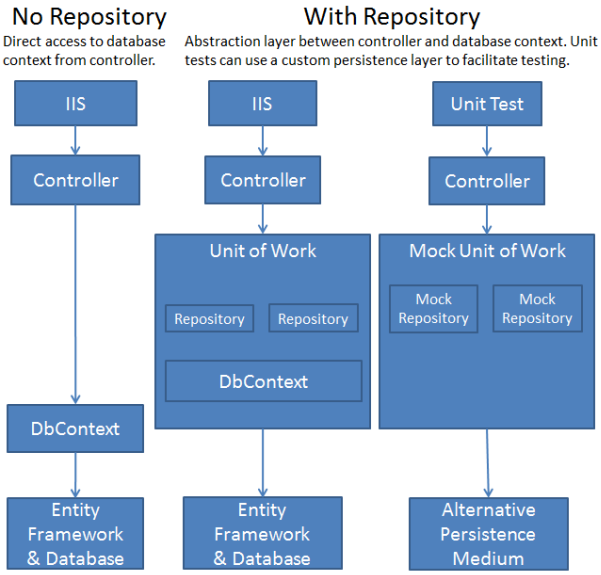
6. ábra: Repository implementáció
Forrás: saját kép

Ezen látjuk, hogy a CarTypeRepository implementálja az ICarTypeRepository interfacet és kap egy AdministrationContextet paraméterként a konstruktorban, amit le is ment. Utána ennek a Contextnek a CarTypes DbSetjét használja az interface metódusainak az implementációjára. Ugye ezek a lekérdezés, hozzáadás, szerkesztés és a törlés műveletek. Ez egy teljesen szokásos LINQ, amivel szoktunk találkozni, csak el van izolálva egy osztályba. Viszont egy interface-nek nem csak egy implementációja lehet, adhatunk egy úgynevezett "kamu" implementációt, ilyenkor a Repository interfacet egy egyszerű listával valósítjuk meg. Erre példát nem fogok mutatni, mert az én esetemben nem ez a megvalósítási forma történt.

Nézzük meg a Repository pattern használatát. Ez úgy fog történni, hogy a Controller alapesetben a DbContextre hivatkozik, vagy ha épp olyan az architektúránk akkor egy service interfacere. Itt a mi esetünkbe a DbContext helyett a Repository interface-re fog hivatkozni. A dependency injection dolga a Repository interface-hez egy megfelelő implementációt átadni, amit beállítunk a dependency injection konfigurációban.

Összefoglalva, a Domain Model nem függ a tárolástól (adatbázisról), és nem is tud róla. A Repository interface a Domain Model része. A Repository felelőssége és feladata, hogy a Domain Modelt betöltse, ismerve a tárolás módját.

Mi a különbség, ha van Repositorynk és ha nincsen?



7. ábra: Repository használatának különbsége
Forrás: learn.microsoft.com

Látjuk az ábrán, hogy ha nincs Repositorynk akkor a Controllerből közvetlen a DbContextet érjük el. Viszont, ha van, akkor ezen az ábrán is látszik, hogy Unit tesztelhető lesz a projektünk. Ebben az esetben a Controllerünk a Unit of Work-re hivatkozik, amire ugye több Repository is kapcsolódhat és a Unit of Work foglalja magába a DbContextet. De hogyha Unit tesztelünk akkor lehet egy úgynevezett Mock Unit of Workünk ami Mock Repositorykban jelenik meg és ez teszi lehetővé azt, hogy a Repository pattern segítségével Unit tesztelhetővé tegyük a kódunkat.

3.2.2. Unit of Work pattern

A Unit of Work pattern egy szoftvertervezési minta, amelyet a szoftverfejlesztésben használnak az adatbázis-központú alkalmazások feladatainak végrehajtására. Segít megőrizni az adatok konzisztenciáját és integritását az adatbázison belül, mivel biztosítja, hogy több műveletet egyetlen tranzakcióként kezeljenek. A Unit of Work mintát általában a Repository mintával együtt használják, és általában az Object-Relational Mapping (ORM) keretrendszerekhez társítják. A funkciója az, hogy minden változtatást egyszerre mentsünk, vagy egyszerre vonjunk vissza, ha közben bármikor hiba történne. (https://dotnettutorials.net)

Az alapötlet az, hogy a komponensek közvetlenül ne kommunikáljanak egymással, hanem egy központi objektumon keresztül érjék el egymást. Ez a központi objektum az úgynevezett "Mediator", amely felelős az összes kommunikáció koordinálásáért és vezérléséért. (https://dotnettutorials.net)

A Unit of Work nyomon követi mindazt az üzleti tranzakció során végzett tevékenységet, amely hatással lehet az adatbázisra. Amikor elkészült vele, kitalálja mindazt, amit meg kell tennie az adatbázis módosításához a munkája eredményeképpen. (https://martinfowler.com/eaaCatalog/unitOfWork.html)

Bemutatom a következő ábrán, hogy is néz ki ez az alkalmazáson belül.

```
public interface IUnitOfWork : IDisposable
{
    1 reference | Reka Acsadi, 243 days ago | 1 author, 1 change
    Task<int> SaveChangesAsync(CancellationToken cancellationToken = default(CancellationToken));
    71 references | Reka Acsadi, 243 days ago | 1 author, 1 change
    Task<bool> SaveEntitiesAsync(CancellationToken cancellationToken = default(CancellationToken));
}
```

8. ábra: Unit Of Work interface
Forrás: saját kép

Ezen az ábrán látjuk, hogy van egy IUnitOfWork interfaceünk, aminek van két darab függvénye, az egyik a SaveChangesAsync a másik pedig a SaveEntitiesAsync. A különbség a kettő között az, hogy míg a SaveChangesAsync-et a CommandHandlerek-ben hívjuk meg, addig a SaveEntitiesAsync pedig a DomainEventHandlerek-ben kap helyett. Mindkettő máshogyan kezeli a DomainEventek elküldését, ezért van rájuk szükség.

Ennek az interface-nek az implementációját szeretném szemléltetni a következő ábrán.

```
public abstract class BaseContext<TContext> : DbContext, IUnitOfWork where TContext : DbContext
{
    readonly IMediator _mediator;

    2 references | Reka Acsadi, 243 days ago | 1 author, 1 change
    protected abstract string DefaultSchema { get; }

    1 reference | Reka Acsadi, 243 days ago | 1 author, 1 change
    protected BaseContext(DbContextOptions<TContext> options) : base(options) { }

    1 reference | Reka Acsadi, 243 days ago | 1 author, 1 change
    protected BaseContext(DbContextOptions<TContext> options, IMediator mediator) ...

    1 reference | Reka Acsadi, 243 days ago | 1 author, 1 change
    protected override void OnModelCreating(ModelBuilder modelBuilder)...

    68 references | Reka Acsadi, 243 days ago | 1 author, 1 change
    public async Task<bool> SaveEntitiesAsync(CancellationToken cancellationToken = default)
    {
        // Dispatch Domain Events collection.
        // Choices:
        // A) Right BEFORE committing data (EF SaveChanges) into the DB will make a single transaction including
        // side effects from the domain event handlers which are using the same DbContext
        // with "InstancePerLifetimeScope" or "scoped" lifetime
        // B) Right AFTER committing data (EF SaveChanges) into the DB will make multiple transactions.
        // You will need to handle eventual consistency and compensatory actions in case of failures in any of the Handlers.
        await _mediator.DispatchDomainEventsAsync(this);

        // After executing this line all the changes (from the Command Handler and Domain Event Handlers)
        // performed through the DbContext will be committed
        var result = await base.SaveChangesAsync(cancellationToken);

        return true;
    }
```

9. ábra: Unit of Work implementáció
Forrás: saját kép

Szerencsénkre az Entity Framework DbContext osztálya alapból egy Unit of Work (vagyis tranzakciónálisan ment). Tehát amikor a DbContext SaveChanges-ét használjuk akkor minden, ami azzal a DbContextel betöltött entitáson változott, azt egyben és tranzakciónálisan fogja menteni. Tehát vagy az összes változtatást menti, vagy ha közben elakad, exception-re futott akkor rollbackkeli az addig mentett változtatásokat. A

DbContext-en az IUnitOfWork interfaceünket a DbContext saját SaveChanges metódusával implementáljuk. Ahhoz, hogy menteni tudjunk minden Repository implementáció rendelkezik egy IUnitOfWork property-vel, ami a DbContextet fogja visszaadni. Ezt láthattuk korábban a 6. ábrán is.

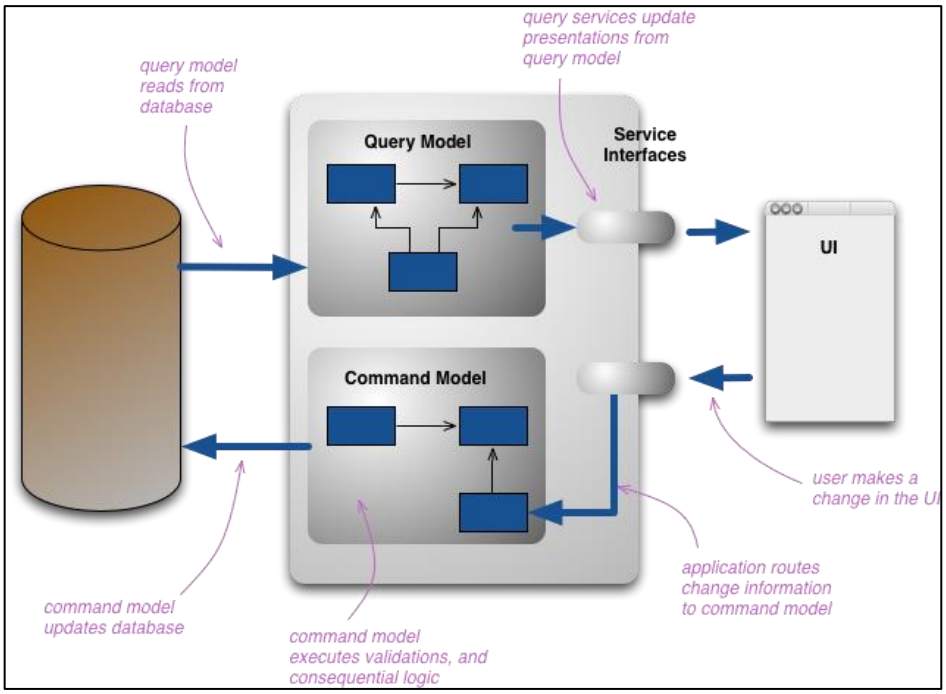
A Unit of Work pattern használatának a lényege az, hogy ezzel tudjuk menteni a Domain Modelben bekövetkezett változásokat.

3.2.3. CQRS (Command-Query Responsibility Segregation)

A CQRS, vagyis a Command-Query Responsibility Segregation egy szoftvertervezési minta, amelyet az alkalmazásarchitektúrák tervezésénél alkalmaznak. Az elv lényege, hogy különválassza a parancsokat (commands) és a lekérdezéseket (queries) a rendszerben, valamint különválassza az adatok írását (update) és olvasását (read) az alkalmazásban. Az alapötlet, hogy a rendszer műveleteit két csoportra bontjuk. (Greg Young 2012)

Ez a következőképpen fog kinézni:

Parancsok (Commands): A parancsok olyan kéréseket vagy utasításokat reprezentálnak, amelyek változtatják az alkalmazás állapotát vagy adatbázisát. Például létrehozhatnak, frissíthetnek vagy törölhetnek entitásokat az alkalmazásban. A parancsok az állapotváltoztatást és az írásos műveleteket kezelik. A Commandok a Domain Model függvényei.
Lekérdezések (Queries): A lekérdezések az adatok olvasását reprezentálják, de nem változtatják meg az állapotot. Például lehetőséget adnak az adatok lekérdezésére és visszaadására az alkalmazásból, hogy azok megjeleníthetők vagy felhasználhatók legyenek. A lekérdezések az adatok olvasását és lekérdezését kezelik. Eltérő modellt igényelnek, mint a Commandok. (https://martinfowler.com/bliki/CQRS.html)



10. ábra: CQRS bemutatása
Forrás: martinfowler.com

A CQRS esetén két külön modellünk van, amik egyébként épülhetnek ugyan arra az adatbázisra a lényeg az, hogy gondolatban és modell szinten is elválasztjuk a kettőt. Amit ezen az ábrán látunk az az, hogy a UI-ról az adatot az egyik modellen keresztül kapjuk és a UI-ról amikor jön valamilyen változtatás az pedig egy másik modellen keresztül megy vissza az adatbázisba. Ez lenne a CQRS-nek a vizuális formája.

A CQRS jól illik komplex domainekhez, ugyanazokhoz, amikhez a Domain-Driven Design is hasznos.

A **Dapper** segítségével SQL query stringeket írhatunk C#-ból, amiket könnyen mapelhetünk objektumokra. A következő ábrán látunk erre egy példát.

```
public async Task<List<CarTypeViewModel>> GetCarType(Administration.Domain.AggregatesModel.UserAggregate.AdminUser adminUser)
{
    if (_checkUserRightService.CheckIfAdminUser(adminUser))
    {
        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();
            var result = await connection.QueryAsync<CarTypeViewModel>(
                @"SELECT [Id] as CarTypeId ,CarTypeName
            FROM [Administration].[CarTypes]
            order by CarTypeName"

            );
            return result.ToList();

        }
    }
    else
    {
        throw new Exception("Ön nem rendelkezik a szükséges jogosultsággal!");
    }
}
```

11. ábra: SQL query string
Forrás: saját kép

Ezen látjuk, hogy paraméterként vár egy AdminUser-t, majd nyit egy SQL connection-t és a QueryAsync az egy Dapper-es extension function, amin a bekeretezett kérés fog lefutni. Ez vissza fogja adni az összes már létező autó típus nevet.

A CQRS használata a következő, létrehozunk egy Query interfacet amin megadjuk, hogy milyen lekérdezéseket fogunk alkalmazni (12. ábra). Ez úgy fog működni, hogy az interfaceből leszármazott Query osztály kap egy connection stringet paraméternek, amire minden egyes Queryben nyitunk neki egy SQL connection-t és a Dapper-en keresztül lefuttattunk egy SQL lekérdezést (13. ábra).

```
public interface ICarTypeQueries
{
    2 references | Oliver Nagy, 54 days ago | 1 author, 1 change
    Task<List<CarTypeViewModel>> GetCarType(Administration.Domain.AggregatesModel.UserAggregate.AdminUser adminUser);
    1 reference | Oliver Nagy, 75 days ago | 1 author, 1 change
    Task<bool> IsExistedCarTypeName(string cartypeName, AdminUser adminUser);
    1 reference | Oliver Nagy, 75 days ago | 1 author, 1 change
    Task<string> GetCarTypesNameByCarTypeId(int carTypeId, AdminUser adminUser);
}
```

12. ábra: Query interface
Forrás: saját kép

```
public class CarTypeQueries : ICarTypeQueries, ICheckExistingCarTypeNameService
{
    private readonly string _connectionString;
    private readonly ICheckUserRightService _checkUserRightService;
    2 references | Oliver Nagy, 73 days ago | 1 author, 1 change
    public CarTypeQueries(string connectionString, ICheckUserRightService checkUserRightService)...

    2 references | Oliver Nagy, 54 days ago | 1 author, 1 change
    public async Task<List<CarTypeViewModel>> GetCarType(Administration.Domain.AggregatesModel.UserAggregate.AdminUser adminUser)...

    1 reference | Oliver Nagy, 73 days ago | 1 author, 2 changes
    public async Task<string> GetCarTypesNameByCarTypeId(int carTypeId, AdminUser adminUser)...

    4 references | Oliver Nagy, 73 days ago | 1 author, 2 changes
    public async Task<bool> IsExistedCarTypeName(string cartypeName, AdminUser adminUser)...
}
```

13. ábra: Query osztály
Forrás: saját kép

3.2.4. Mediator pattern

A Mediator pattern egy szoftvertervezési minta, amely segít a komponensek közötti közvetlen kapcsolatok elkerülésében és azok közötti kommunikáció központosításában. A Mediator minta azt célozza meg, hogy csökkentse a komponensek közötti függőségeket és segítse elő a laza kapcsolatot, ami hozzájárulhat a szoftverrendszerek könnyebb karbantartásához és bővítéséhez. Az alapötlete az, hogy a komponensek közvetlenül ne kommunikáljanak egymással, hanem egy központi objektumon keresztül érjék el egymást. Ez a központi objektum az úgynevezett "Mediator", amely felelős az összes kommunikáció koordinálásáért és vezérléséért. A Mediator minta gyakran alkalmazható olyan rendszerekben, ahol a komponensek közötti kommunikáció összetett és szervezett. (Dr. Ferenc Rudolf 2011, pp. 34-36.)

Egy program sok osztályból, és az ő interakcióikból épül fel. Ahogy egyre több az osztály, az interakciók is komplexebbek lesznek. Milyen problémákat oldhat meg a Mediator?

- Objektumok közötti szoros függőséget (tight coupling) kerülni kell.
- Adott objektumok közötti interakciót az objektumok változtatása nélkül is lehessen módosítani.

Hogyan oldja meg a Mediator ezeket a problémákat?

- Definiálunk egy külön mediator objektumot, ami magába foglalja az objektumok közötti interakciókat.
- Az objektumok az interakciókat nem közvetlen egymással intézik, hanem átadják a mediátor objektumnak, és az hajtja végre őket.

A Mediator Pattern klasszikus példája a légiforgalmi irányító rendszer. Ebben a repülőgépek kommunikálnak egymással és az irányítótoronnyal, hogy biztosítsák a biztonságos felszállást, leszállást és a navigációt. Az irányítótorony koordinálja a repülőgépek közötti interakciókat, lehetővé téve a légi forgalom irányítását anélkül, hogy a repülőgépeknek közvetlenül kommunikálniuk kellene egymással. Ebben az esetben az irányítótorony lesz a mediator.

Ennek az implementációját egy NuGet package segítségével használjuk. A következő ábrákon bemutatom ennek a működését. Ez úgy néz ki, hogy van két interfaceünk, az egyik az ISender, a másik az IPublisher.

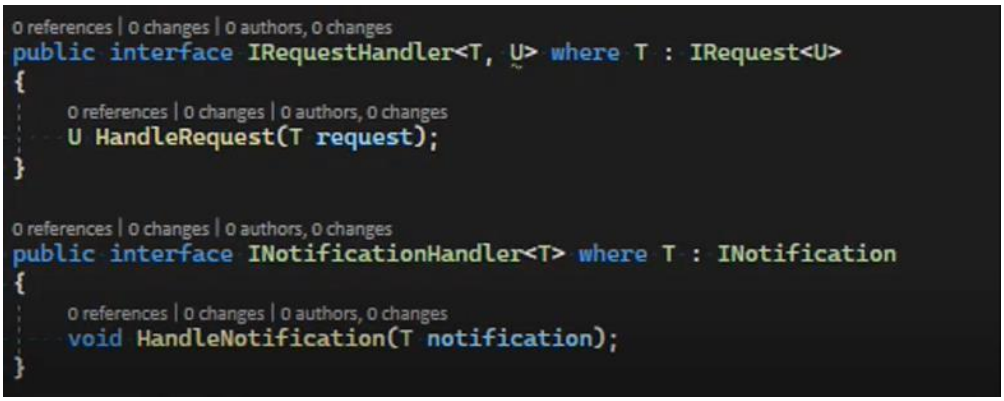
```
/// <summary>
/// Send a request through the mediator pipeline to be handled by a single handler.
/// </summary>
public interface ISender
{
    /// <summary>
    /// Asynchronously send a request to a single handler
    /// </summary>
    /// <typeparam name="TResponse">Response type</typeparam>
    /// <param name="request">Request object</param>
    /// <param name="cancellationToken">Optional cancellation token</param>
    /// <returns>A task that represents the send operation. The task result contains the handler response</returns>
    Task<TResponse> Send<TResponse>(IRequest<TResponse> request, CancellationToken cancellationToken = default);
```

14. ábra: ISender interface
Forrás: saját kép

```
/// <summary>
/// Publish a notification or event through the mediator pipeline to be handled by multiple handlers.
/// </summary>
public interface IPublisher
{
    Task Publish(object notification, CancellationToken cancellationToken = default);

    /// <summary>
    /// Asynchronously send a notification to multiple handlers
    /// </summary>
    /// <param name="notification">Notification object</param>
    /// <param name="cancellationToken">Optional cancellation token</param>
    /// <returns>A task that represents the publish operation.</returns>
    Task Publish<TNotification>(TNotification notification, CancellationToken cancellationToken = default)
        where TNotification : INotification;
}
```

15. ábra: IPublisher interface
Forrás: saját kép

Az ISender send művelete egy generikus IRequest paramétert vár, míg az IPublisher publish művelete egy INotification paramétert. Ezek lesznek az üzenetek, amiket majd tovább küldünk. Ebből már megtudjuk fogalmazni, hogy mi lesz egy Mediator. Egy Mediatornak vagy küldünk egy requestet ami azért generikus, hogy változhasson, hogy ez mivel tér majd vissza, vagy átadunk neki egy notification-t amit ő majd tovább küld. Ennek a másik oldala, hogy ez, hogy jut el ahhoz, akihez kell, a Medaitor package definiál két interface-t, az IRequestHandler-t és az INotificationHandler-t.



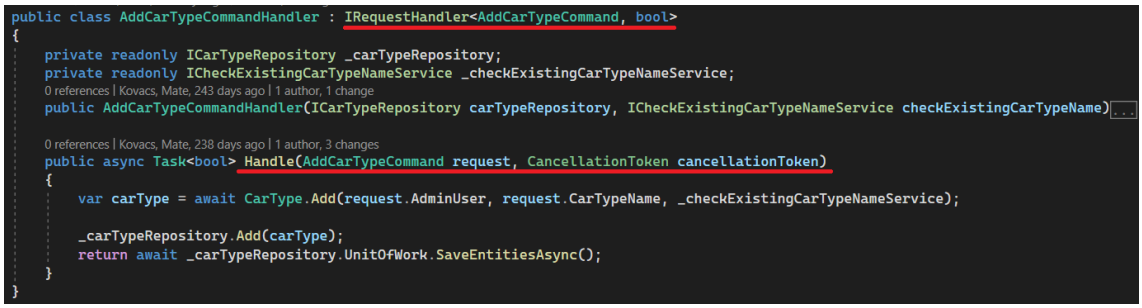
16. ábra: IRequest és INotification handler
Forrás: saját kép

Az IRequestHandler interface az azt köti ki, hogyha van egy T requestünk ami U-t ad vissza, akkor a RequestHandler tudja a T requestet kezelni és egy U-t visszaadni. Hasonlóképpen a NotificationHandler, hogyha kap egy T notification fajtát azt le tudja kezelni.

Szeretném bemutatni ennek a használatát a következő ábrán.

17. ábra: IRequest használata
Forrás: saját kép

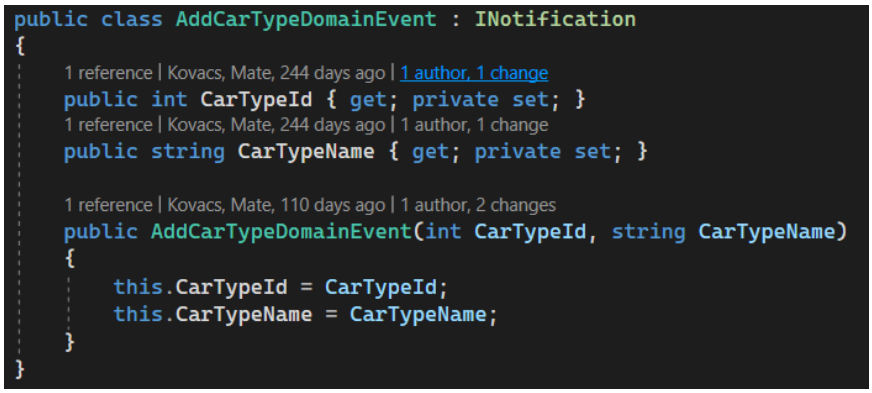
A Mediator pattern csatlakozni fog a CQRS-hez, olyan konkrét módon, hogy a Commandjainkhoz fog tartozni egy ilyen Command osztály, ami egy IRequest-et valósít meg. Itt jelezzük, hogy ez egy IRequest és azt is jelezzük, hogy milyen eredményt várunk tőle, ugye látjuk, hogy egy bool eredményt. Csinálunk egy Command osztályt, ebben az esetben az autó típus hozzáadása lesz a műveletünk. Ez paraméternek vár egy AdminUser-t és egy autó típus nevet. Ezután kell nekünk egy osztály a Command kezelésére.



18. ábra: IRequestHandler használata
Forrás: saját kép

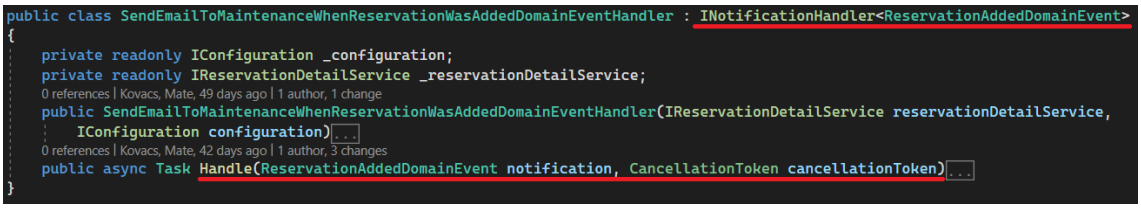
Ez az osztály megvalósítja az IRequestHandler interface-t, ami kezeli az AddCarTypeCommand-ot. A Handle függvény kap egy AddCarTypeCommand kérést és vissza kell adnia egy bool-t. Ezt úgy implementáljuk, hogy a RequestHandler az konstruktor paraméterben vár egy CarTypeRepository-t és amikor új autó típust hozunk létre, akkor lepéldányosítjuk, hozzáadjuk a repositoryhoz és mentjük a Unit of Work segítségével. Ez volt az IRequest használata.

Most szeretném bemutatni, hogy az INotificationöknek hol van a helye. Az INotificationök nagyon jól illenek a Domain Eventek-re a Domain-Driven Desgin-ban. Ez úgy fog kinézni, hogy lesz egy DomainEvent osztályunk és meg kell adnunk, hogy ez megvalósítja az INotification interface-t.



19. ábra: INotification használata
Forrás: saját kép

Egy jó példa a NotificationHandler-re az e-mail küldés. Létrehozunk egy DomainEventHandler osztályt, ami implementálja az INotificationHandler interface-t, ami kezeli a Domain Eventünket. Ebben az esetben a Handle függvény dolga az lesz, hogy egy foglalás feladásakor e-mailt küldjön a karbantartásnak erről az új igényről.



20. ábra: INotificationHandler használata
Forrás: saját kép

3.2.5. Factory method

A Factory Method egy szoftvertervezési minta, amelyet a szoftvertervezés során alkalmaznak. A Factory Method segítségével objektumokat hozunk létre anélkül, hogy közvetlenül hívással vagy példányosítással dolgoznánk az objektumok konkrét osztályainak konstruktorával. Ehelyett egy absztrakt interfész vagy osztály definiálja a létrehozandó objektumok típusát és a létrehozás módját, majd a konkrét osztályok implementálják ezt az interfészt vagy öröklik az absztrakt osztályt, és megvalósítják a létrehozás részleteit. A Factory Method minta segítségével az alkalmazások rugalmasak maradhatnak, mivel az objektumok létrehozására vonatkozó döntéseket a konkrét osztályokra hagyja. Ez lehetővé teszi az alkalmazások módosítását vagy bővítését anélkül, hogy megváltoztatnánk a meglévő kódot, ami segít a karbantartásban és az új funkciók hozzáadásában. (E. Gamma, R. Helm, R. Johnson, J. Vlissides 1994, pp. 107-117.)

A Factory Method minta előnyei közé tartozik:

Rugalmasság: A kliensek nem szorulnak rá a konkrét osztályok ismeretére, melyek objektumokat hoznak létre. Ez lehetővé teszi a kód könnyebb karbantartását és bővítését.
Kicserélhetőség: A Factory Method lehetővé teszi új osztályok bevezetését vagy meglévők kicserélését a kliens kód módosítása nélkül.
Továbbfejleszthetőség: A Factory Method lehetővé teszi a létrehozási logika továbbfejlesztését, például a konkrét objektumok előállításához szükséges konfigurációs adatok vagy paraméterek hozzáadását. (E. Gamma, R. Helm, R. Johnson, J. Vlissides 1994, pp. 139-151.)

A Factory Method-ra egy jó szemléltető példa az Office alkalmazásaiban jól ismert Új menüpont, amely mindegyik alkalmazásban létrehoz egy új dokumentumot és megnyitja

azokat. A megnyitásuk egyformán történik, de a létrehozásuk különböznek egymástól. Amíg a szövegszerkesztő program egy üres szöveges dokumentumot, addig a táblázatkezelő program esetén egy üres munkafüzetet hozunk létre. (https://hu.wikipedia.org)

A Factory Method egy függvény, ami példányokat hoz létre egy osztályból. A leggyakoribb ilyen függvény a konstruktor. Több konstruktor is létezhet egy osztályon belül, de ha ugyan az a nevük akkor ez nem lesz túl Clean Code, azaz nem túl könnyen értelmezhető. Ennek a használatára szeretnék mutatni egy példát a következő ábra segítségével. Mivel egy konstruktort nem tudunk aszinkronná tenni, ezért egy statikus aszinkron függvényt hozunk létre.

```
public static async Task<CarType> Add(UserAggregate.AdminUser adminUser, string cartypeName,
    ICheckExistingCarTypeNameService checkExistingCarTypeNameService)
{
    if (!adminUser.IsAdmin)
    {
        throw new FleetManagementDomainException("A hozzáadás sikertelen! Ön nem rendelkezzik a hozzáadáshoz szükséges jogosultsággal.");
    }

    var carTypeNameExist = await checkExistingCarTypeNameService.IsExistedCarTypeName(cartypeName, adminUser);
    if (carTypeNameExist)
    {
        throw new FleetManagementDomainException("A hozzáadás sikertelen! A megadott gépjárműtípus korábban már hozzá lett adva!");
    }

    return new CarType(cartypeName);
}
```

21. ábra: Factory Method használata
Forrás: saját kép

Ez a függvény egy CarType-ot fog visszaadni, ez azért egy Factory mert a CarType egy példányát fogja visszaadni. Megkapja a paramétereket és ebben az esetben egy await segítségével ellenőrzi, hogy létezik-e ilyen nevű autó típus már a rendszerben. Ha nem volt ilyen akkor létrehozza az újat, ha viszont talált azonos nevűt akkor egy kivételt dob és a hibaüzenetet megjeleníti nekünk. A statikus függvény létrehozása után a konstruktort privátra állítjuk, hogy véletlen se lehessen azon keresztül példányosítani, csak is az Add függvény segítségével.

3.2.6. Adapter pattern

Az Adapter pattern egy olyan szoftvertervezési minta, amely segít két inkompatibilis interfész közötti együttműködés kialakításában. Az Adapter pattern-t általában azért alkalmazzuk, hogy összekapcsoljunk két olyan rendszert vagy komponenst, amelyeknek eltérő interfészeik vannak, és nem tudnának közvetlenül együttműködni anélkül, hogy megváltoztatnánk az egyik vagy mindkettő kódját. Az Adapter pattern egy közvetítőréteget hoz létre, amely lehetővé teszi az inkompatibilis interfészek közötti átjárást és együttműködést. (E. Gamma, R. Helm, R. Johnson, J. Vlissides 1994, pp. 273-282.)

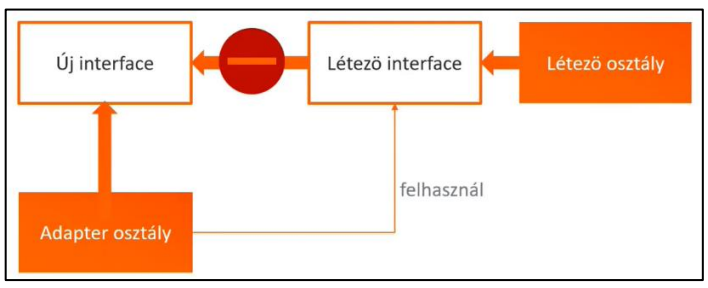
Példaként, gondoljunk egy olyan alkalmazásra, amely a mért adatokat egy különböző mérőműszerek által használt inkompatibilis interfészekkel gyűjti össze. Az Adapter pattern lehetővé teszi az alkalmazás számára, hogy a mért adatokat egységes interfészen keresztül gyűjtse be, miközben az egyes mérőműszerek specifikus interfészeit érintetlenül hagyja.

Milyen problémákat oldhat meg az Adapter pattern?

- Hogyan használhatunk egy létező osztályt olyan interfészként, amit alapból nem implementál?
- Hogyan egészíthetünk ki egy létező osztályt egy új interfésszel, ha az osztályt nem módosíthatjuk?

Hogyan oldja meg az Adapter pattern ezeket a problémákat? (22. ábra)

- Definiálunk egy Adapter osztályt, ami implementálja az új interfészt.
- Az Adapter osztály az új interfész implementációjához felhasználja a meglévő osztály funkcionalitását.



22. ábra: Adapter pattern működése
Forrás: saját szerkesztés

4. A FEJLESZTÉS MENETÉNEK BEMUTATÁSA

A fejlesztések a Scrum módszertan keretein belül zajlottak. Ez egy jól ismert és nagyon elterjedt fejlesztési módszertan, amit a KUKA Hungária is régóta használ.

4.1. A Scrum módszertan

A Scrum egy olyan projektmenedzsment és fejlesztési módszertan, amely eredetileg a szoftvertervezés területén született, de mára számos más területen is használják a projektvezetés hatékonyságának növelésére. A Scrum egyfajta iteratív és inkrementális fejlesztési folyamatot alkalmaz. A Scrum célja az, hogy növelje a projektek átláthatóságát, gyorsítson a fejlesztési folyamaton, és lehetővé tegye a gyakori visszajelzéseket és változtatásokat a projekt menetében. Azok a projektek, amelyek változékony követelményekkel rendelkeznek vagy agilis megközelítést igényelnek, gyakran a Scrumot választják projektmódszerként. A Scrum módszertan előre definiált elemekből áll, amelyek mind egy előre meghatározott célt szolgálnak és mind szükségesek ahhoz, hogy a módszertan hatékonyan működjön. (K. Schwaber, J. Sutherland 2020)

4.1.1. A Scrum csapat

A Scrum középpontjában a scrum csapat áll. Ehhez kapcsolódóan vannak meghatározva különböző szerepkörök, események, munkaanyagok és szabályok. A Scrum csapat tagjai, a Scrum Master, a Product Owner azaz a terméktulajdonos és a fejlesztők. Ők mind együtt felelősek az eredményekért.

A Product Owner képviseli a megrendelőt a fejlesztés során, feladata, hogy a csapat olyan feladatokkal foglalkozzon, amelyek üzleti szempontból valóban fontosak, és ezáltal maximalizálja a teremtett értéket.

A Scrum Master elsődleges feladata magának a Scrum módszertannak a betartása és a működésének támogatása. Segít a csapatnak, hogy megértse a Scrum Guide-ban megfogalmazott értékeket, szabályokat, és minél hatékonyabban alkalmazza a gyakorlatban.

A Fejlesztőcsapat felelős a termék elkészítéséért. Ez egy kis számú (3-9 fős) csapat, a csapatban minden olyan szakterület képviselője rendelkezésre áll, ami a termék sikeres fejlesztéséhez szükséges. A fejlesztőcsapat célja, hogy minden sprint végére egy működő, bemutatható, potenciálisan leszállítható terméket hozzon létre. (https://promanconsulting.hu/scrum-modszertan-alapok)

4.1.2. A Scrum események

A Sprint a Scrum szíve, ahol az ötleteket értékké alakítják. Ezek egy hónapos vagy annál rövidebb fix időtartamú események. Az előző Sprint befejezése után azonnal indul egy újabb Sprint. A termékcél eléréséhez szükséges összes munka, beleértve a Sprint tervezést, a napi scrumokat, a Sprint Review-t és a Sprint Retrospective, a Sprinteken belül történik.

Minden sprint előtt a Sprint tervező megbeszélés során a Product Backlog teendői közül kiválasztják az adott sprint során elvégzendő feladatokat. Ez a Product Owner és a Fejlesztőcsapat közös feladata, ahol megbecsülik a feladatok elvégzéséhez szükséges időtartamot, és ígéretet tesznek a feladatok elvégzésére. A sprint tervezés időtartama változó, attól függően, hogy a sprint hány hétből áll. Az én esetemben egy Sprint 3 hétből állt.

A Daily Scrum egy 15 perces esemény a Scrum csapat fejlesztői számára. A bonyolultság csökkentése érdekében a Sprint minden munkanapján ugyanabban az időben és helyen kerül megrendezésre.

A Sprint Review célja, hogy megvizsgálja a Sprint eredményét, és meghatározza a jövőbeli adaptációkat. A Scrum csapat bemutatja munkája eredményeit a biznisz résztvevőinek, és megvitatják a termékcél felé tett előrehaladást.

A Sprint Retrospective célja a minőség és a hatékonyság növelési módjainak megtervezése. A Scrum csapata megvizsgálja, hogyan zajlott a legutóbbi Sprint az egyének, interakciók, folyamatok, eszközök és azok teljesítésének meghatározása tekintetében. Azonosítják azokat a feltevéseket, amelyek félrevezették őket, és feltárják az eredetüket. A Scrum Team megvitatja, hogy mi ment jól a Sprint során, milyen

problémákkal találkozott, és hogyan oldották meg ezeket a problémákat. (K. Schwaber, J. Sutherland 2020)

4.2. Event Storming

Az Event Storming egy kollaboratív modellezési technika, amelyet a szoftverfejlesztés és az üzleti folyamatok modellezése területén használnak. Ez egy módszer komplex rendszerek és folyamatok feltárására, elemzésére és tervezésére. Az Event Stormingot Alberto Brandolini hozta létre, és népszerűségre tett szert a DDD és agilis szoftverfejlesztő közösségekben. Az Event Storming elsődleges célja a megosztott megértés elősegítése a csapattagok és az érintettek között, megkönnyítve a problémák azonosítását, a megoldások meghatározását és a szoftverrendszerek hatékony tervezését. Az Event Storming különféle kontextusokhoz illeszthető, például szoftverfejlesztéshez, üzleti folyamatok modellezéséhez vagy bármely olyan tartományhoz, amely összetett interakciókat és eseményeket foglal magában. Értékes eszköz a DDD-hez, az eseményvezérelt architektúrákhoz és az agilis fejlesztési gyakorlatokhoz. (https://creately.com/blog/meeting-visual-collaboration/event-storming)
Alapfogalmak:

1. Eventek: Mi történhet a rendszerben?
2. Commandok: Milyen szándék válthatja ki a történéseket? A felhasználó mire akarja utasítani a rendszert? (Általában ugyanúgy hívjuk, mint az eseményt, csak felszólító módban)
3. Read modelek: A user milyen információkra alapozza a szándékát?
4. Userek: Melyik command-ot melyik user válthatja ki? Írhatunk konkrét embert is (pl: Géza módosíthat telephely beállítást)
5. Policies: Ha egy eseményre a rendszer egy másik része is reagálni akar, a policy új commandot adhat ki. Elnevezésük kb "<kiváltott command neve>, amikor <kiváltó esemény neve>".
6. Conditions: Bármilyen feltétel, aminek megkell, hogy feleljen a command.

Ezek az alapfogalmak a hozzájuk tartozó adott színű cetlin lesznek kifejtve az Event Storming során, a fejlesztők végig mennek a biznisz által megfogalmazott igényeken és egyesével kifejtik azokat és felvezetik őket ezekre a cetlikre.

Egy tipikus Event Storming eszköz a miro, ez egy olyan platform, ami segíti a csapatkommunikációt és a projektmenedzsment megkönnyítését. A következő ábrán bemutatom egy funkció működését az Event Storming és a miro segítségével.

23. ábra: Egy funkció megfogalmazása a miro-ban
Forrás: saját kép

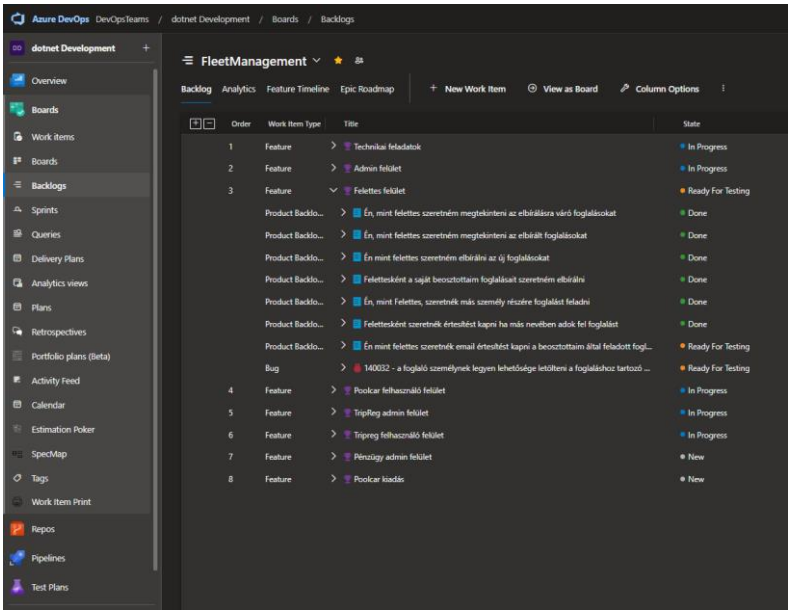
Az igény a következőképpen volt megfogalmazva: “Én, mint felhasználó, szeretnék a saját nevemben foglalást feladni.” Ezt az Event Storming során az ábrán látható módon lett kibontva. Láthatjuk, hogy a Command, azaz a parancs (kék cetli) a “Foglalás feladása” lesz, ami kifogja váltani a “Foglalás fel lett adva” Eventet, azaz eseményt (narancssárga cetli). Azt is kitudjuk olvasni az ábrából, hogy ez a funkció a Felhasználói hatáskörrel lesz elérhető. Illetve az ehhez tartozó Condition-t, azaz a feltételt (világoskék cetli), hogy ez a parancs lefusson. Ez nem más mint, hogy legyen szabad a gépjármű a kiválasztott időpontban. A zöld cetlin minden olyan információt látunk, amiket a felhasználó megad majd egy foglalás feladásához és amire alapozza ezt a szándékát.

4.3. A projekt indítása

A projekt indítása egy úgynevezett Kick-off meetinggel kezdődött. Ez egy olyan meeting, ahol a Scrum csapat és a biznisz oldal találkozik és bemutatkoznak egymásnak. Megbeszélik a projekt folyamatának menetét, lépéseit és különböző mérföldköveit, mint például az első Sprint Retrospective időpontját. A biznisz elmeséli a projekt létrejöttének indokát és célját. Itt veszi kezdetét a projekt. Ezután a Scrum csapat összeül és átnézik a biznisz által a Backlog-ba felvett összes igényt és tovább viszik az Event Storming-ra, hogy ott kifejtsék őket.

4.4. Projekt indító Event Storming

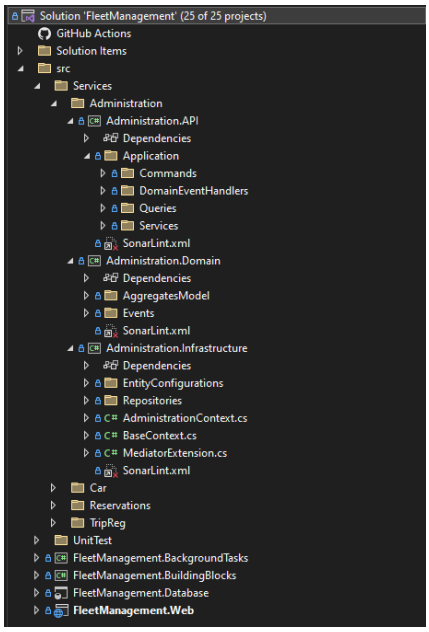
Először a biznisz által megfogalmazott igényeket néztük át az Event Storming során. A Backlog áttekintéséhez, ahol az igények vannak megfogalmazva, mi az Azure DevOps nevű szolgáltatását használtuk. Az Azure DevOps egy olyan rendszer, amely magában foglalja mindazon funkciókat, amely az alkalmazás teljes életciklusának kezeléséhez szükségesek. Különféle munkaszervezési modellek állnak rendelkezésre ahhoz, hogy a termék előállításában érdekeltek szervezhessék és nyomon követhessék a teljesítés állapotát. Az Azure DevOps-ban alkalmazott struktúra az organizációkra épül, amelyek magukban foglalják a szervezeti szintű beállításokat és hozzáféréseket, illetve a fejlesztési projekteket. A projektek biztosítják azt a funkcionalitást, amely a fejlesztési folyamatot jellemzi, kezdve a forráskód kezeléstől a buildelési és tesztelési tevékenységeken keresztül a szoftver verziók kiadásáig bezárólag.



24. ábra: Azure DevOps Backlog felülete
Forrás: saját kép

Az igények megfogalmazására van egy jól bevált szabály, amit a biznisznek követnie kell az igények leírására. A felhasználók szemszögéből és a program kívánt működését határozza meg a következő struktúra szerint: „Én mint <felhasználó/jogkör>, azt szeretném, ha <tevékenység>, azért, hogy <indoklás>.”

4.5. Könyvtár struktúra bemutatása

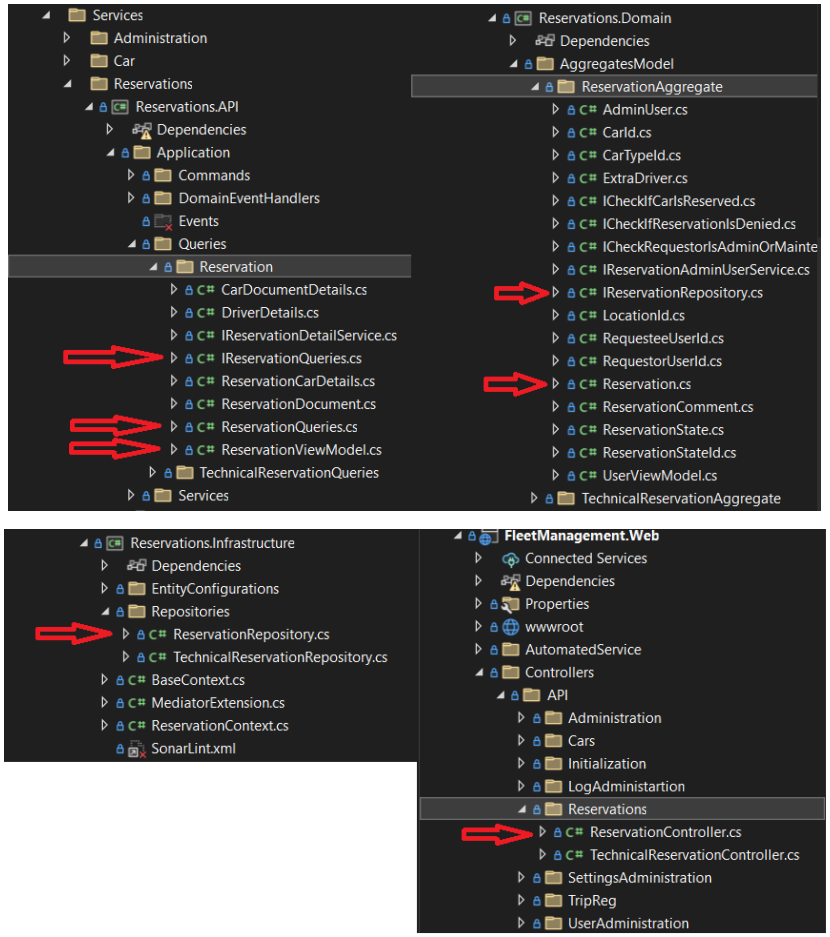


25. ábra: A projekt könyvtár struktúrája
Forrás: saját kép

Amit itt fontos látni, az az, hogy a Solution-ben van egy Web projekt. Ez ad otthont egy adag API cotnrollernek, illetve ez ad otthont egy adag MVC controllernek amik a view működését fogják biztosítani, az API controllerek pedig az alkalmazás szívével fognak kommunikálni, ez az ami a Domain Driven Design-ból jön. Ezen egy szinten találunk egy Services mappát, a service-k egy nagy egységei a programnak, amiket leginkább úgy lehet elkülöníteni, mint külön álló Bounded Context-ek. Látjuk, hogy van négy darab service-ünk, amiből az egyik az adminisztrációs részekkel foglalkozik, egy másik magával az autókkal, egy harmadik a foglalásokkal és a negyedik a TripReg funkciókkal. Ez utóbbi az a rész, amiről a jövőbeli fejlesztések részben fogok beszámolni. Azt is láthatjuk még, hogy három projekt van service-enként. Van egy Domain modell, egy Infrastruktúra projekt és van egy Application vagy röviden API projekt. Ez igy az alapfelállás, van egy BaseContext és egy MediatorExtansion fájl, ami benne van az architektúrában alapból. Mindegyik servicehez létre van hozva egy DbContext, az ábrán ezt AdministrationContext néven láthatjuk.

4.6. Aggregate implementálása

Az Aggregate Root-ok kialakításához először is tovább visszük az Event Storming alatt kidolgozott részeket az Aggregate modellezésre. Itt először csoportosítjuk a Commandokat/eseményeket az alanyuk szerint, majd ebből lesznek az Entitások. Ha megvannak az Entitások, akkor őket újra csoportosítjuk az alapján, hogy van-e felügyeleti kapcsolat egyik Entitás és másik Entitás között, ha van akkor ezek lesznek az Aggregate-ek. Ezek Entitások, amik szorosan összetartoznak. Minden Entitás egy Aggregate része.



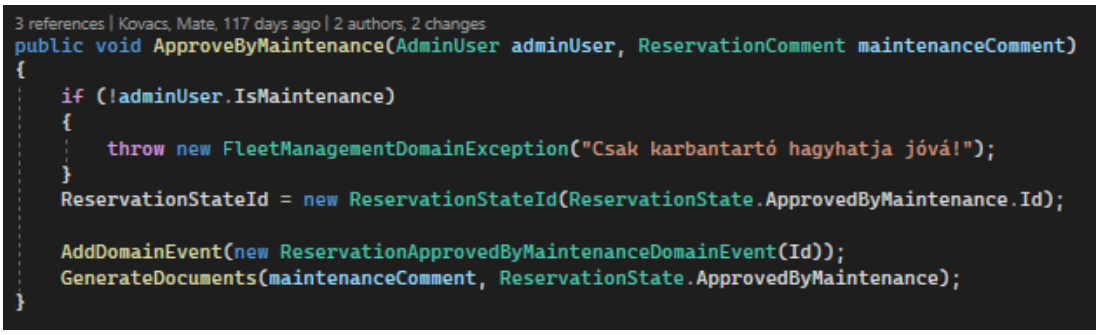
26. ábra: Foglalás, mint Aggregate
Forrás: saját kép

Ebben az esetben a Reservation azaz foglalás lesz az Aggregate Root. Ezért az Aggregate-et is róla nevezzük. Először is a Domainben létrehozunk az Aggregates Model mappa alá egy ReservationAggregate nevű mappát, amiben elkészítjük az Aggregatre Root-ot. Ehhez az Aggregate Root-hoz tartozik egy IRepository interfész. Ezután az Infrastrucutre projektben létrehozunk egy Repository implementációt. Aztán az Application projektben

létrejön hozzá egy Query interfész és ehhez egy Query implementáció, illetve egy ViewModel. Utoljára pedig a Web projektben a Controllerek közé a saját service mappájába készítünk egy controllert az Aggregate-nek.

27. ábra: Foglalás, mint Aggregate Root
Forrás: saját kép

Ezen az ábrán a Foglalások Aggregate Root-ját láthatjuk. Látjuk, hogy egy foglalás mennyi tulajdonsággal rendelkezik. Ezeknek a többségét mind megadjuk egy foglalás feladásakor. Egy ilyen Aggregate Root rendelkezik különböző Domain műveletekkel, ezeket tulajdonképpen különböző funkcióknak is nevezhetjük, ilyenek az alapműveletek mint, hozzáadás vagy létrehozás, módosítás és a törlés. Ezekkel a műveletekkel általában mindegyik Aggregate Root rendelkezik, ezen felül pedig tartalmazhatnak összetettebb műveleteket is. A foglaláshoz néhány ilyen példa a foglalás jóváhagyása Admin által, de lehetőség van a Felettesnek és a Karbantartásnak is jóváhagyására. Ennek ellentétje pedig a foglalás elutasítása, amire szintén mind a három jogkörnek van lehetősége. Az Administration service alatt lévő Aggregate Root-ok leggyakrabban csak az alapműveleteket foglalják magukba, hiszen ezek különböző adminisztrációs dolgokra lesznek használva, ilyen például az Autó típus, a Dokumentumok, a Telephelyek, az Autók tulajdonosai, akiktől a cég lízingeli őket stb. Egy ilyen művelet a következőképpen épűl fel:

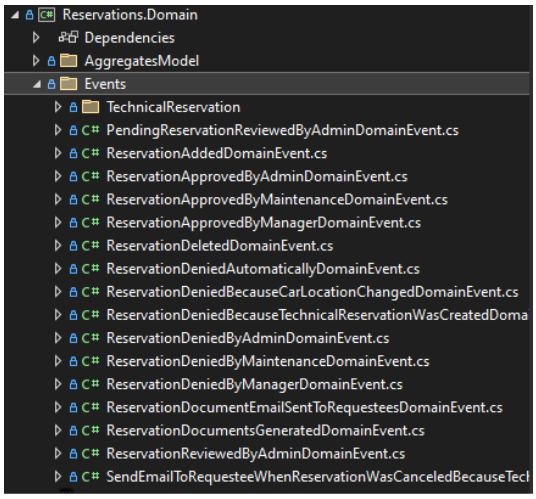


28. ábra: Domain művelet
Forrás: saját kép

A példa, amivel szemléltetni szeretném egy Domain művelet megvalósítását az a Karbantartás általi jóváhagyás lesz. Itt látjuk, hogy két paramétert vár a metódus, az egyik az egy adminUser ami azt mutatja meg, hogy az adott felhasználó aki ezt a műveletet szeretné végrehajtani milyen jogkörökkel rendelkezik, a másik pedig egy megjegyzés amit ez esetben a Karbantartás hagy jóváhagyás esetén. Ezen adatok tudatában először megnézzük egy Elágazáson belül, hogy az adott felhasználó rendelkezik-e Karbantartási jogkörrel, amennyiben nem, akkor egy Domain Exception-t dobunk, amit a felületen is látni fog a felhasználó. Ha mégis rendelkezik az adott jogkörrel a felhasználó, akkor ez esetben az adott foglalás státuszát „Karbantartás által jóváhagyott” -ra állítjuk, majd egy Domain Event-et hozunk létre értesítve a rendszert, hogy ez a művelet megtörtént. Erről a Domain Event-ről a következő fejezetben fogok részletesebben beszélni. Az utolsó lépése ennek a műveletnek a GenerateDocuments nevű szintén Domain művelet meghívása, ami azt a célt szolgálja, hogy létrehozza a géphasználati engedélyt, amit majd a foglaló személy fog megkapni és ezzel az engedéllyel az autót igénybe tudja venni.

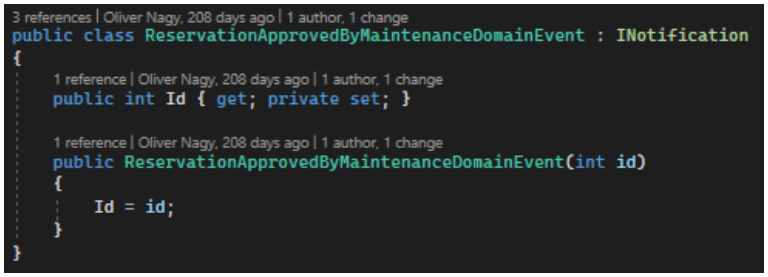
4.7. Domain Event implementálása

A Domain Eventek az Event Storming során eseményként megfogalmazott dolog lesznek kódban modellezve. Őket is a Domain projekt alatt a saját Events nevű mappájukba hozzuk létre őket.



29. ábra: Foglaláshoz tartozó Domain Eventek
Forrás: saját kép

Minden Domain Event az INotification-ből származik és minden olyan tulajdonsággal rendelkezik, amik fontosak lehetnek az esemény szempontjából. Egy példa erre az esetre az előző fejezetben bemutatott jóváhagyás karbantartás általi művelet eseménye. Ennek az eseménynek csak az adott foglalás Id-jára lesz szükség, hogy tudjuk melyik foglalás lett jóváhagyva. Minden Eventnek van konstruktora, tehát létre lehet hozni őket, viszont mivel a tulajdonságok private setter-rel rendelkeznek, ezért utólag nem lehet őket módosítani. Ezzel közöljük a Visual Studioval és az összes fejlesztővel azt, hogy egy Domain Event az egy olyan dolog, ami létrejött és utána nem változtatható. Igy modellezük azt, hogy ha egy Command végrehajtódik akkor az eseményt kiváltja.



30. ábra: Domain Event
Forrás: saját kép

4.8. Repository Interface implementálása

A Repository Interface szorosan az Aggregate-hez tartozik, mivel az Aggregate Root-tal címkézzük. Általában a hozzáadás, módosítás, törlés és a visszaadás műveletekkel

rendelkezik. Ezek implementálása az adott Repositorykban kapnak helyet. Ők az IUnitOfWork segítségével fognak működni.

```
32 references | Kovacs, Mate, 201 days ago | 3 authors, 6 changes
public interface IReservationRepository : IRepository<Reservation>
{
    4 references | ● 2/2 passing | Reka Acsadi, 300 days ago | 1 author, 1 change
    Reservation Add(Reservation reservation);
    10 references | Reka Acsadi, 300 days ago | 1 author, 1 change
    void Update(Reservation reservation);
    2 references | ● 1/1 passing | Reka Acsadi, 292 days ago | 1 author, 1 change
    void Delete(Reservation reservation);
    12 references | Reka Acsadi, 300 days ago | 1 author, 1 change
    Task<Reservation> GetAsync(int reservationId);
}
```

31. ábra: Repository interfész
Forrás: saját kép

4.9. Value Object használata

```
84 references | 0 changes | 0 authors, 0 changes
public record AdminUser(string SapNumber,
    bool IsAdmin, bool IsMaintenance,
    bool IsManager, bool IsEmployee) : ValueObject;

19 references | 0 changes | 0 authors, 0 changes
public record ReservationStateId(int Value) : ValueObject;
```

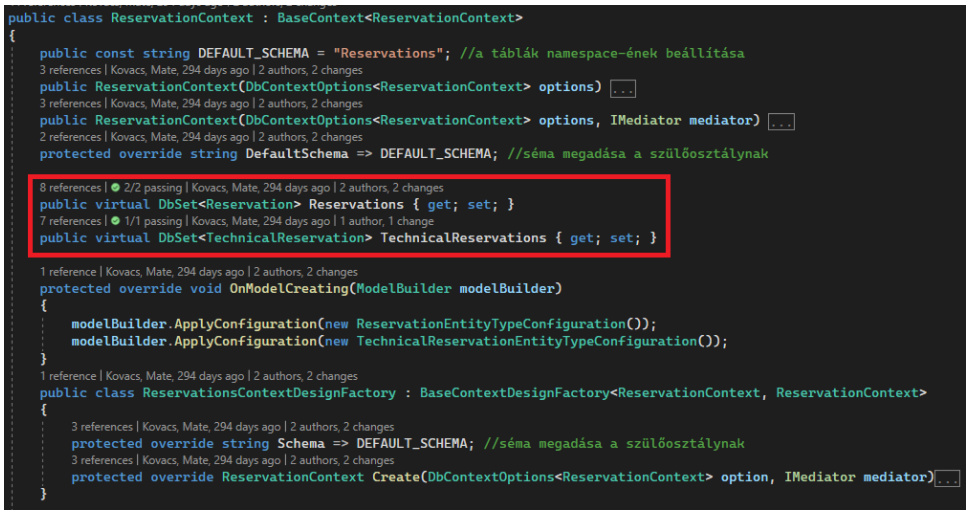
32. ábra: AdminUser és Foglalás státusz, mint Value Object
Forrás: saját kép

A Value Object az gyakorlatilag a duálisa az Entitásnak. Ami alatt azt értem, hogy az Entitásnak van egy Id-ja és a tulajdonságai változhatnak össze vissza, ő mindig ugyanaz az Entitás marad, neki van egy ilyen folyamatos létezése. A Value Object valahol ennek az ellenkezője, neki is lehetnek tulajdonságai, de a Value Object csak és kizárólag a tulajdonságainak az összesége. Tehát ha megváltoztatjuk egy tulajdonságát, akkor az azt jelenti, hogy ez egy tök más Value Object. Illetve két Value Object akkor egyenlő, hogy ha minden egyéb tulajdonságuk is egyenlő. Ezek úgy viselkednek, mint az Entitások egy Aggregate Root-on belül. A foglalások esetében, ilyen Value Objecteket használunk, mint a CarId(az autó id-ja), RequestorUserId(a foglaló id-ja), LocationId(a telephely id-ja) és az ábrán is látható ReservationStateId(a foglalás státuszának id-ja), illetve az AdminUser ami az összes Service-ben megtalálható, ez szolgál arra, hogy megtudjuk az adott felhasználóról, hogy milyen jogkörökkel rendelkezik. A Car service alatt lévő Car Aggregate Root-ban is használunk Value Objecteket, ilyenek a CarTypeId, ami az autó típusát jelöli, itt is használjuk a LocationId-t, hiszen a kocsik is különböző telephelyeken

érhetőek el. Az OwnerId mutatja a céget, akitől az adott autó lízingelve van, illetve a CostCenterId, ami szintén, mint a telephely, ez a telephelyen belüli költség helyet fogja mutatni.

4.10. Infrastructure rész implementálása

Itt már az elkészült Domain Model-ből dolgozunk. Nézzük meg milyen fájlok tartoznak a Perzisztenciához. Először is minden Service-hez van egy DbContext amiben minden egyes Aggregate-hez fog tartozni egy DbSet, így kezdjük az EF Core konfigurálását.



33. ábra: Reservation DbContext
Forrás: saját kép

Ezután tartozni fog mindegy egyes Entitáshoz egy EntityTypeConfiguration nevű fájl. Ez tárolja azt, hogy hogyan fogjuk az adott Entitást adatbázis táblává alakítani. Itt fontos, hogy megadjuk a sémát, ami alapján létrehozza majd a táblát, ez gyakorlatilag a namespace-e az adatbázis tábláknak és ez szervízenként különböző, ezt láthatjuk is a 33. ábra első sorában. Mindenképp az Id-t megjelöljük kulcsként és beállítjuk, hogy nem akarjuk letárolni a DomainEvents tulajdonságot. Majd beállítjuk a navigation és private tulajdonságokat. Ahogy megvan az elkészült EntityTypeConfiguration fájl ezt a DbContext-nek az OnModelCreating függvényébe betudjuk regisztrálni.

```
class ReservationEntityTypeConfiguration : IEntityTypeConfiguration<Reservation>
{
    0 references | Kovacs, Mate, 204 days ago | 3 authors, 5 changes
    public void Configure(EntityTypeBuilder<Reservation> reservationConfiguration)
    {
        reservationConfiguration.ToTable("Reservations", ReservationContext.DEFAULT_SCHEMA);

        reservationConfiguration.HasKey(a => a.Id);

        reservationConfiguration.Ignore(a => a.DomainEvents);
```

34. ábra: Reservation EntityTypeConfiguration
Forrás: saját kép

Ezután implementáljuk a már korábban említett IRepository interfészt. Ez ugyan úgy néz ki minden egyes Aggregate Root-ra. Az Add művelettel hozzáadjuk a DbSet-hez az új objektumot, az Update az az, hogy a DbContextnek beregisztráljuk, hogy modified lett az objektum, hogy észrevegye ezt majd a mentésnél. A GetAsync pedig lekérdezi az adott objektumot és visszaadja. A Delete pedig törli az objektumot a DbSet-ből. Ezzel az Infrastructure projekt el is készült.

4.11. Application rész implementálása

Az Application projektbe fognak tartozni a Query-k, a Command-ok és a hozzájuk tartozó Handler-ek, illetve, ha használunk akkor a Domain Event Handler-ek is. Ezek a saját nevű mappájukban érhetők el, külön Aggregate-re bontva. Először a Query-ket fogom bemutatni, ezeket a Dapper segítségével használjuk. A Query-nek van egy implementációja és egy hozzá tartozó Interfésze.

```
private async Task<List<int>> GetAllDeniedReservations(SqlConnection connection,
    DateTime pickupTime, DateTime submissionTime)
{
    var deniedState = GetDeniedStates();
    string dateRangesOverlap = $"( @pickupTime < SubmissionTime AND @submissionTime > PickupTime )";
    var result = await connection.QueryAsync<int>(
        @"SELECT CarId
            FROM [Reservations].[Reservations]
            WHERE" + dateRangesOverlap +
            "AND ReservationStateId IN @deniedState", new { pickupTime, submissionTime, deniedState }
        );
    return result.ToList();
public async Task<bool> RequestorIsAdminOrMaintenanceUser(string sapNumber)
{
    using (var connection = new SqlConnection(_connectionString))
    {
        connection.Open();
        var result = await connection.QueryAsync<string>(
            @"SELECT SapNumber
            FROM [Administration].[Users] u
            inner join [Administration].[UserRoleIds] r on u.Id=r.UserId
            where SapNumber=@sapNumber", new { sapNumber }
        );
        return result.Any();
```

35. ábra: Foglaláshoz tartozó lekérdezések
Forrás: Saját kép

A 35. ábrán két különböző lekérdezést láthatunk. A GetAllDeniedReservation nevezetű arra a célra szolgál, hogy visszaadja mindazon foglalást, amik el lettek utasítva. Láthatjuk, hogy ez egy privát metódus, ami annyit jelent, hogy ezt egy másik lekérdezésben fogjuk használni, hivatkozva rá. Illetve a RequestorIsAdminOrMaintenanceUser lekérdezés pedig azt adja vissza, hogy a foglalást feladó személy rendelkezik-e Admin vagy Karbantartás jogkörrel. Ez már egy publikus metódus, tehát itt megtörténik az adatbázishoz való kapcsolódás. Ezt connection string segítségével érjük el és a connection.Open() sorral nyitjuk meg a csatlakozást. Minden publikus lekérdezésünk ezekkel a sorokkal kezdődik. Ezután jön a Dapper segítségével az SQL stringek létrehozása, az ábrán ezeket láthatjuk narancssárga színezéssel. Ide kerülnek be a különböző SQL kódok. A két lekérdezésben az is különbözik, hogy míg az egyik egy int típusú listát ad vissza, hiszen itt több foglalás Id-ját fogjuk tárolni, addig a másik csak egy bool, azaz igaz vagy hamis értéket fog nekünk visszaadni. Attól függően, hogy milyen típusú eredményt várunk, úgy állítjuk be a lekérdezést. Ezek a lekérdezések általában a felületen megjelenített dolgokhoz kellenek, de vannak, amelyek kapcsolódnak különböző funkciókhoz és ezek működéséhez. Rengeteg féle lekérdezést készítünk, de a leggyakoribb lekérdezés, amit használunk az egy Aggregate-nek az összes példányát visszaadni.

A következő része az Application projektnek az a Command-ok és a hozzájuk tartozó Command Handler-ek. Az Event Storming alatt megfogalmazott Commandok-hoz fog egy Command nevű fájl és egy Command Handler kapcsolódni.

```
public class ApproveReservationByAdminCommand : IRequest<bool>
{
    [JsonProperty]
    2 references | Daniel Kovari, 280 days ago | 1 author, 1 change
    public int ReservationId { get; private set; }
    [JsonProperty]
    2 references | Daniel Kovari, 280 days ago | 1 author, 1 change
    public AdminUser AdminUser { get; private set; }
    [JsonProperty]
    2 references | Molnar, Benedek, 151 days ago | 1 author, 1 change
    public ReservationComment AdminComment { get; private set; }
    3 references | Molnar, Benedek, 151 days ago | 1 author, 1 change
    public ApproveReservationByAdminCommand(int reservationId,
        AdminUser adminUser, ReservationComment adminComment)
    {
        ReservationId = reservationId;
        AdminUser = adminUser;
        AdminComment = adminComment;
    }
```

36. ábra: Foglalás jóváhagyása Command
Forrás: Saját kép

Először is a Command fájlban lesznek mind azok a paraméterek, amiket az Aggregate modellezésnél összegyűjtöttünk, hogy ezekre lesz szükségünk. A Command-okra is igaz az, hogy egyszer létrehozhatóak, de nem módosíthatóak már. Ezt a private set részből tudhatjuk. Fontos az architektúra szempontjából, hogy a Mediator-nek az IRequest interfészét implementálják a Command-ok. Ez azért fontos, hogy a Command Handlerek kezelni tudják ezeket a kéréseket.

```
public class ApproveReservationByAdminCommandHandler : IRequestHandler<ApproveReservationByAdminCommand, bool>
{
    private readonly IReservationRepository _reservationRepository;
    2 references | Molnar, Benedek, 199 days ago | 2 authors, 4 changes
    public ApproveReservationByAdminCommandHandler(IReservationRepository reservationRepository)
    {
        _reservationRepository = reservationRepository;
    }
    2 references | Molnar, Benedek, 151 days ago | 2 authors, 8 changes
    public async Task<bool> Handle(ApproveReservationByAdminCommand request,
        CancellationToken cancellationToken)
    {
        var reservation = await _reservationRepository.GetAsync(request.ReservationId);
        if (reservation is null) { return false; }
        reservation.ApproveByAdmin(request.AdminUser, request.AdminComment);
        _reservationRepository.Update(reservation);
        return await _reservationRepository.UnitOfWork.SaveEntitiesAsync();
    }
```

37. ábra: Foglalás jóváhagyása CommandHandler
Forrás: Saját kép

Egy Command Handler mindig megvalósítja az IRequestHandler interfészt, amiben megadjuk a hozzá tartozó Comamnd-ot és hogy egy bool értéket kérünk vissza a végén. Amit a 37. ábrán látunk az gyakorlatilag az összes Command Handler-re vonatkozik, paraméternek megkapja az adott Aggregate repository-át és amikor a Commandot kezeli, betölti a Command-ban kapott Id alapján az Aggregate Root-ot, ha nem sikerül akkor visszatér egy hamis eredménnyel, ha pedig sikerült betölteni, akkor meghívja az Aggregate Root-on a Command-hoz tartozó függvényt. Ebben a példában az adott Id-hoz tartozó foglalásnak hívja meg az ApproveByAdmin nevű függvényét. Ha ez sikerült akkor ezután egy Update-et hívunk a repository-ra és a Unit Of Work-on mentünk. Ezeket a Handlereket elkészítjük mindegyik Command-hoz.

A fent említett Domain Event Handler-ek is ugyan ilyen sémával léteznek, annyi küllönbséggel, hogy ők inkább az INotification interfészt valósítják meg. Itt abban az esetben hozunk létre Domain Event Handler-t, ha szeretnénk egy eseményre feliratkoztatni valamilyen történést. A szoftver esetében ilyen Domain Event Handlerekben készítjük el az e-mail küldést bizonyos eseményekre, mint például: értesítés, ha a foglalás jóvá lett hagyva, gép használati engedély kiküldése stb.

Ezek implementálásával el is készült az Application projektünk, ahhoz, hogy ezek a Commandok és lekérdezések működjenek a felületen szükségünk lesz egy ASP.NET Core Web app projektre.

4.12. Web projekt implementálása

A Web projekten belül, amiket használunk és létrehozunk azok a Java script-ek, az API Controllerek és a felületet biztosító cshtml view-k. Először az API Controllereket szeretném bemutatni. Elsősorban az API Controllerekről lesz szó. Őket a Web projekten belül a Controller mappában, saját Aggregate-re bontva hozzuk létre.

```
[Route("api/v1/Administration/[controller]")]
[ApiController]
3 references | Oliver Nagy, 113 days ago | 2 authors, 15 changes
public class CarTypeController : ControllerBase
{
    private readonly IMediator _mediator;
    private readonly IAdminUserService _adminUserService;
    private readonly ICarTypeQueries _cartypeQueries;
    private readonly ILogger<CarTypeController> _logger;

    0 references | Oliver Nagy, 120 days ago | 1 author, 1 change
    public CarTypeController(IMediator mediator, ICarTypeQueries cartypeQueries, ILogger<CarTypeController> logg

    [Route("add")]
    [HttpPost]
    [ProducesResponseType((int)HttpStatusCode.OK)]
    [ProducesResponseType((int)HttpStatusCode.BadRequest)]
    0 references | Oliver Nagy, 120 days ago | 2 authors, 5 changes
    public async Task<IActionResult> Add([FromBody] Services.Administration.Commands.AddCarTypeCommand command)
    {
        try
        {
            var adminUser = await _adminUserService.GetAdminUser();
            var newcommand = new FleetManagement.Services.Administration.API.Application.Commands
                .CarTypeCommands.AddCarTypeCommand(adminUser, command.CarTypeName);
            var commandResult = await _mediator.Send(newcommand);

            if (!commandResult)
            {
                return BadRequest();
            }

            return Ok();
        }
        catch (Exception ex)
        {
            return BadRequest(ex.Message);
        }
```

38. ábra: Controller és Action
Forrás: Saját kép

Amint látjuk az ábrán paraméterben kap egy Mediator-t és az adott Aggregate-hez tartozó queryket. A Controller-ben úgynevezett Actionöket hozunk létre, mindegyik Action rendelkezik headerökkel ahol meg kell adni a route-ot, illetve, a művelettől függ, hogy éppen HttpGet vagy HttpPost-ot használunk. Amelyik Action létrehoz, az HttpGet-et kap, amelyik pedig módosít az HttpPost-ot fog kapni. Minden Command-hoz tartozni fog egy Action. Az Actionök JSON-ként fogják megkapni a Command paramétereit, ezt látjuk a „[FromBody]” tag-ből. Egy Action paramétere mindig a hozzá tartozó Command lesz. Lényegében egy Action annyit csinál, hogy a Mediator-be elküldi a Command-ot, a

Command-nak a Handler-e kezeli majd elvégzi a műveletet. Majd az eredménynek megfelelően „BadRequest” -et vagy „Ok” státusz kódot küld vissza. Egy Action tartozhat egy queryhez is, erre egy példa a következő ábra.

```
[Route("")]
[HttpGet]
[ProducesResponseType(typeof(CarTypeViewModel[]), (int)HttpStatusCode.OK)]
[ProducesResponseType((int)HttpStatusCode.NotFound)]
0 references | Oliver Nagy, 120 days ago | 2 authors, 6 changes
public async Task<IActionResult> Index(DataSourceLoadOptions loadOptions)
{
    try
    {
        var adminUser = await _adminUserService.GetAdminUser();
        var result = await _cartypeQueries.GetCarType(adminUser);
        return Ok(DataSourceLoader.Load(result, loadOptions));
    }
    catch (Exception ex)
    {
        return NotFound();
    }
}
```

39. ábra: Query Action
Forrás: Saját kép

Egy try catch-be berakjuk a lekérdezést és hogyha sikeres akkor itt is „Ok” státuszt kap, ha pedig sikertelen volt akkor egy „Not Found” státuszt kapunk. Hasonló, mint a Command Action-je, annyi különbséggel, hogy itt nem vesszük igénybe a Mediator segítségét.

Az API Controllereken kívül használunk még Frontend kontrollereket is, ezek a felület megjelenítését fogják biztosítani. Ők kapcsolják össze a Controller-t a view-val. Példa egy ilyen Controller-re:

```
public class CarController : Controller
{
    0 references | Patkos, Mate, 115 days ago | 1 author, 1 change
    public IActionResult Car()
    {
        return View();
    }
}
```

40. ábra: Frontend Controller
Forrás: Saját kép

Ahhoz, hogy a szoftvernek legyen egy megjeleníthető formája is, létrehozunk minden egyes felülethez és menüponthoz egy Razor view fájlt. Ezekben a view-kban különböző DevExtreme-es kontrollokat használunk. Ezekhez a cég hozzáférést biztosit a fejlesztői számára. A leggyakrabban használt kontroller az a DataGrid, ez úgy viselkedik, mint egy

táblázat, vannak sorai és oszlopai. Ilyet használunk a különböző adminisztrációs Entitásokhoz, mint például a telephelyek, tulajdonosok, autó típusok, dokumentum típusok, jogkörök kezelése, illetve az autó flotta kezelése is egy ilyen DataGrid által valósult meg. A szoftver egyik fő funkciója, ahol a foglalásokat lehet megtekinteni és ezen a felületen foglalni, az pedig egy úgynevezett Scheduler, azaz egy naptár szerű ütemező kontrollerrel valósul meg.

```
<script>
    var myLocationStore = DevExpress.data.AspNet.createStore({
        key: "LocationId",
        loadUrl: siteUrl + "/api/v1/Administration/Location",
        insertUrl: siteUrl + "/api/v1/Administration/Location/add",
        deleteUrl: siteUrl + "/api/v1/Administration/Location/delete",
        updateUrl: siteUrl + "/api/v1/Administration/Location/update",
        onBeforeSend: kukaDefaultBeforeSend("LocationId"),
        onAjaxError: kukaDefaultOnAjaxError});
</script>
@(Html.DevExtreme().DataGrid<LocationViewModel>()
    .ID("gvLocations")
    .Columns(column =>
    {
        column.AddFor(row => row.LocationName)
            .Caption(new JS("formatMessage('Location')"))
        .HeaderFilter(headerFilter => headerFilter.AllowSearch(true))
        .AllowFiltering(true)
        .AllowResizing(true)
        .AllowSearch(true)
        .AllowEditing(true);
    })
    .DataSource(new JS("myLocationStore"))
    .LoadPanel(l => l.Enabled(false))
    .HeaderFilter(headerFilter => headerFilter.Visible(true))
    .SearchPanel(searchPanel => searchPanel.Visible(true))
    .NoDataText("Nincs megjeleníthető adat.")
    .ColumnAutoWidth(true)
    .WordWrapEnabled(true)
    .ShowBorders(true)
    .ShowRowLines(true)
    .ShowColumnLines(true)
    .OnRowRemoving("gvLocationsOnRowRemoving")
    .Editing(editing =>
    {
        editing.Mode(GridEditMode.Form);
        editing.AllowUpdating(true);
        editing.AllowAdding(true);
        editing.AllowDeleting(true).ConfirmDelete(false);
    })
)
```

41. ábra: Telephelyek view
Forrás: saját kép

Az API Controllerrel úgy tud a DevExtreme-es kontroll kommunikálni, hogy meg kell neki adni egy DataSource-t, ahol meg kell adni mi a LoadUrl, az InsertUrl, az UpdateUrl, a DeleteUrl. Látjuk, hogy itt az API Controller-ben felvett Actionök routejai fognak szerepelni. Itt még meg kell adni egy Key-t, ami általában az adott Aggregate Id-ja lesz, amivel a felület foglalkozni fog. Ami még itt fontos az az OnBeforeSend függvény hívása, mivel ez felelős azért, hogy konvertáljon a DevExtreme által értett módszer között meg ahogy a DDD-s API-kat létrehozzuk a között. Az ábrán ezt egy myLocationStore nevű változóba tároljuk és azt adjuk át a DataGrid DataSource property-nek.

Ezeken a kontrollereken különböző eseményekre vagy a felületen való kattintás eseményeire készítünk JavaScript functionöket, ilyenek lehetnek például az adott telephelyhez való kocsik megjelenítése egy legördülő menüben, a DataGrid-ből való sor törlése után a Grid újbóli betöltése vagy a foglalás sorára kattintva jóváhagyjuk azt.

```
function onRowApproveClick(e, approveUrl) {
    rowKey = e.row.data.ReservationId;
    popupCommandUrl = approveUrl;
    let commentPopup = getCommentPopup();
    commentPopup.show()
}
function setLocationValue(rowData, value) {
    selectedLocation = value;
    selectedLocationForScheduler = value;
    rowData.LocationId = value;
    rowData.CarId = null;
}
```

42. ábra: JavaScript függvények
Forrás: saját kép

Minden felület és menüpont felkonfigurálása után elkészül a Web projektünk is.

4.13. A szoftver tesztelése

A fejlesztési szakasz után pedig következnek az átvételi tesztek, de ez általában már nem a fejlesztőcsapat feladata. A komponenstesztek azok, amelyek a rendszer legkisebb részeit, azaz magukat a funkciókat/metódusokat tesztelik. Ezeknek a metódusoknak ismerjük a működését, így meg tudjuk határozni, hogy adott bemenetek esetén milyen kimeneti választ várunk. Így egységtesztek segítségével megvizsgálhatjuk, hogy az elvárt eredmény megegyezik-e a kapott eredménnyel. A komponens tesztek közé sorolhatjuk még a modul teszteket is, ezek viszont nem az elvárt kimenetet vizsgálják, hanem a metódus nem-funkcionális tulajdonságait nézik (pl.: gyorsaság). Az integrációs tesztek segítségével tudjuk vizsgálni az interfészen keresztüli kommunikációt, együttműködéseket. Segítségükkel azt vizsgálhatjuk, hogy a programban megfelelően hajtódnak-e végre az egymás után következő folyamatok, jól tudnak-e kommunikálni egymással a különböző komponensek, rendszerek. Ezek a tesztelésnek már egy magasabb szintjét képviselik. A rendszertesztek segítségével már azt vizsgálják, hogy a készülő rendszer megfelel-e a specifikációnak és a rendszertervnek. Ezt sokszor már egy dedikált teszt csapat végzi, mert a fejlesztőcsapat sokszor elfogult a saját munkájával kapcsolatban. Ide soroljuk a UI teszteket is, amikor az elkészült felületet tesztelik végig, hogy sikerül-e kezeletlen hibát kiváltaniuk a szoftverből.

Az egység tesztek írásához Microsoft .NET környezetben sok jó keretrendszer áll a rendelkezésünkre. A mi választásunk az xUnit keretrendszer használatára esett. Ami

közös ezekben a keretrendszerekben, az a tesztek felépítése. Minden esetben szükségünk lesz egy teszt projekt hozzáadására. Itt foglalnak majd helyet a tesztesetek, elszeparálva a tesztelni kívánt kódtól. Ezekben a projektekben teszt osztályokat találunk majd, amelyek a megnevezésben feltüntetett kódrészeket hivatottak tesztelni. Ezek a teszt osztályok foglalják majd magukba az általunk megírt teszteseteket. Mindig a tesztelni kívánt kódhoz illeszkednek a tesztosztályok és a tesztesetek elnevezései, ezzel biztosítjuk a könnyű átláthatóságot és értelmezhetőséget.

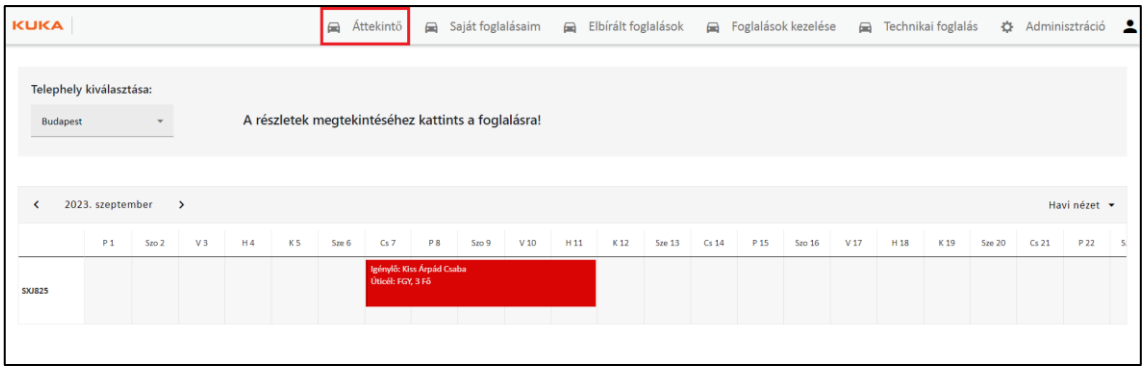
```
public class AddCarCommandHandlerTest : RepositoryTestBase
{
    [Fact]
    0 references | Molnar, Benedek, 138 days ago | 1 author, 3 changes
    public async Task AddCarCommand_Success()
    {
        // Arrange
        var carRepository = await CreateCarRepositoryAsync();

        var adminUser = new AdminUser(true, false, false);
        var begin = new DateTime(2023, 4, 1, 8, 30, 52);
        var end = new DateTime(2023, 4, 2, 9, 30, 50);
        var techDate = new DateTime(2023, 12, 1, 8, 30, 52);
        var document = new Document("test", "pdf", new byte[1], DateTime.Today, 1, false, 1);
        Document[] documents = { document};

        AddCarCommand c = new AddCarCommand(adminUser,1,1,1,"plate","chasis","contract",begin,end,1,1,techDate,1,1,1,"userid", documents);
        AddCarCommandHandler ch = new AddCarCommandHandler(carRepository);

        // Act
        var result = await ch.Handle(c, CancellationToken.None);

        // Assert
        Assert.True(result);
    }
```

43. ábra: Autó felvétele unit teszt
Forrás: saját kép

Minden tesztnek lesz egy (keretrendszertől függő) attribútuma, amivel meg tudjuk jelölni, hogy ez egy teszteset. A teszt neve egy a névkonvencióval meghatározott formátumú elnevezés lesz. A teszt törzse pedig mindig az alábbi 3 részből épül fel: Arrange, Act, Assert. Az arrange részben definiáljuk általában azt a tesztobjektumot, amin a tesztelni kívánt funkciót el tudjuk végezni. Ha több ilyen objektumra van szükségünk, az mind az Arrange részbe kerül. Az act rész az, ami a tesztelendő funkció meghívását jelenti. Az assert részben hasonlítjuk össze az elvárt eredményt az act részben kapott eredménnyel. Ha ezek megegyeznek, a tesztünk sikeresen lefut. Ha nem egyeznek meg, a tesztünk elbukik, és meg kell vizsgálnunk, hogy mi a bukás oka. Egy teszt bukása esetén lehetőségünk van a Visual Studio segítségével debuggolni a teszt kódot. Ez a folyamat ugyanúgy működik, ahogy a kód debuggolása is, így nem okoz különösebb nehézséget a hiba okának felderítése.

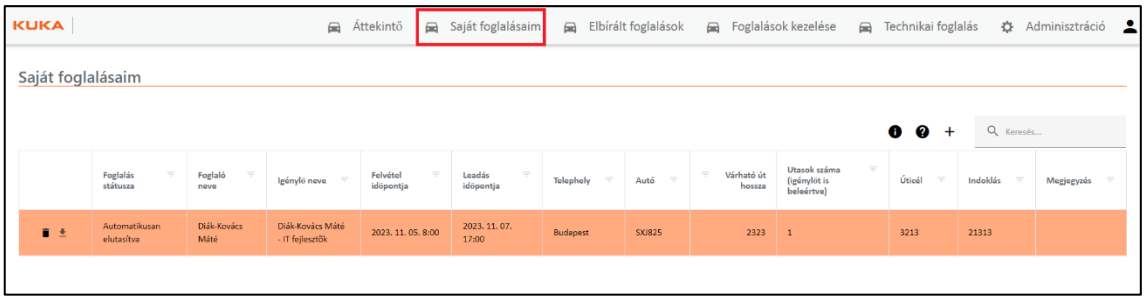
A fejlesztés során a sprintenként kiadott verziókat a biznisz feladata volt tesztelni. Itt tudtak kiderülni különböző bugok és hibák. A fejélesztőcsapat pedig a Unit tesztek írásával segítette a tesztelést.

4.14. Az elkészült szoftvert bemutatása



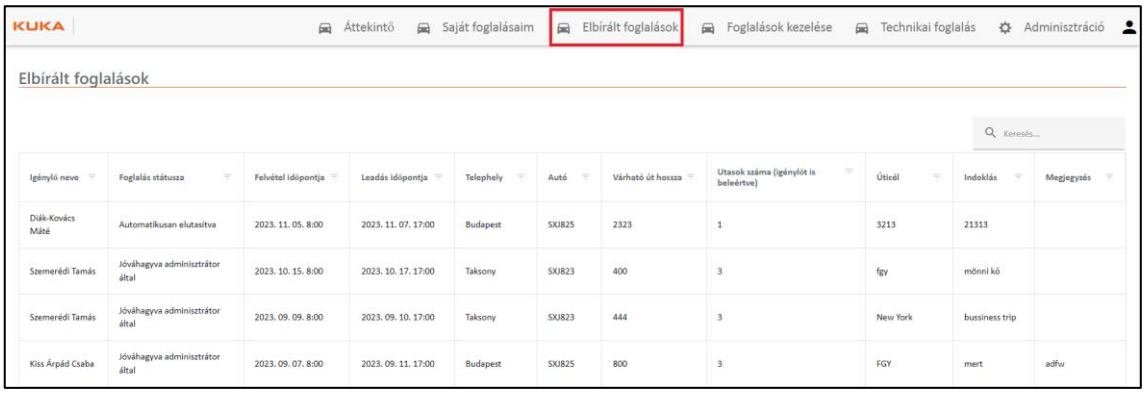
44. ábra: Áttekintő felület
Forrás: saját kép

A weblapra bejelentkezve egyből az Áttekintő menü fogad minket. Ez a szíve a szoftvernek, itt történnek a foglalások feladása, illetve itt látszódnak a szabad és a lefoglalt időpontok is. Ez a felület telephelyre van bontva és azon belül is az elérhető kocsikra, egy sor egy kocsihoz tartozik. A foglalások státusza különböző színekkel van megjelenítve, a piros a foglalt és jóváhagyott, a sárga a foglalt, de még jóváhagyásra váró. Heti, illetve havi nézet között tudunk váltani, igény szerint ki mit preferál. A foglalás részleteinek megtekintéséhez duplakattintással megjelenik egy felugró ablak, ahol láthatjuk minden fontos paraméterét. Ezen a felületen is van módunk letölteni a géphasználati engedélyt amennyiben a foglalásunk jóváhagyásra kerül, egyébként pedig jóváhagyás után automatikusan e-mailben is elküldi a rendszer ezt a dokumentumot.



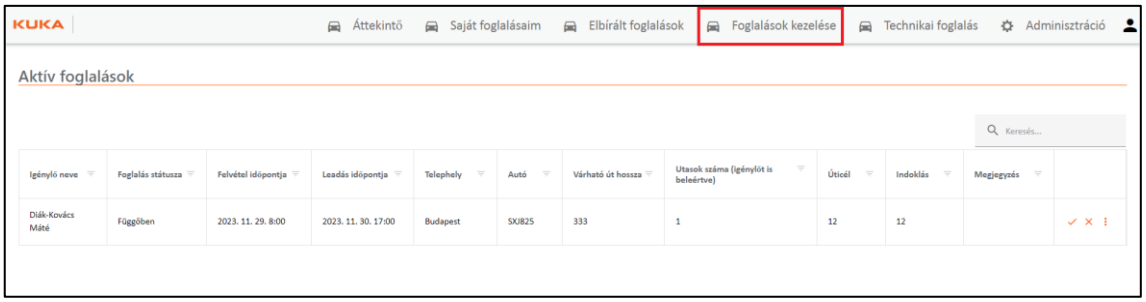
45. ábra: Saját foglalásaim menü
Forrás: saját kép

A következő menüpont a Saját foglalásaim, itt is lehetőségünk van feladni foglalást annyi kivétellel, hogy itt nem egy naptárban intézzük, hanem a plusz gomb megnyomásával egy felugró ablakban töltjük ki a formot. Ezen a felületen csak a saját feladott foglalásainkat látjuk, másét nem. Itt elérhető két darab dokumentum, ami a szoftver leírását és a gépkocsik használatát tartalmazza.



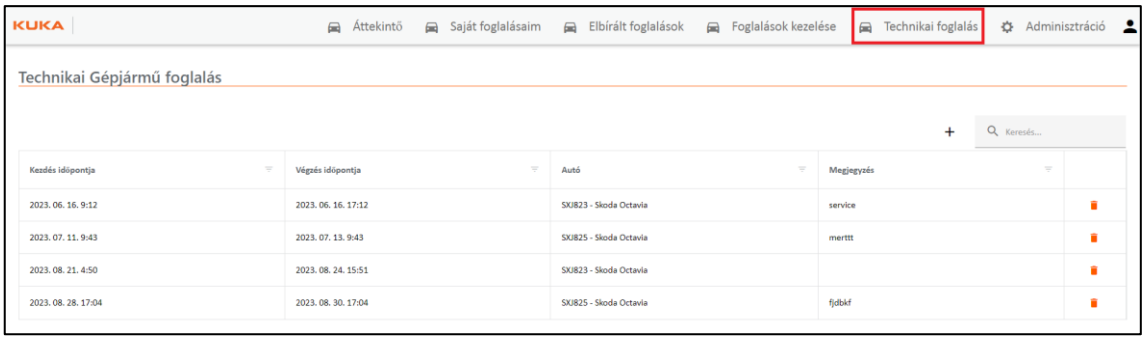
46. ábra: Elbírált foglalások menü
Forrás: saját kép

Az elbírált foglalások menüben látszódnak a korábban és aktuálisan feladott foglalások státusztól függetlenül. Ez úgy viselkedik, mint egy böngészési előzmény. Ehhez csak az Adminok és a Karbantartási joggal rendelkezők férnek hozzá.



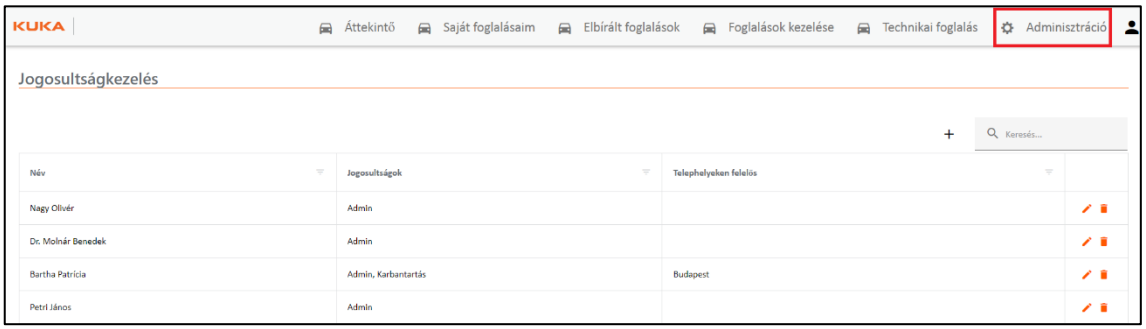
47. ábra: Foglalások kezelése menü
Forrás: saját kép

A foglalások kezelése menüpontban lehetősége van az Adminoknak, Karbantartási és felettesi joggal elfogadni, illetve elutasítani a foglalásokat. Karbantartás csak a saját telephelyéhez feladott foglalást látja, a felettes pedig csak a saját alkalmazottai foglalását tudja elbírálni. Itt is elérhető egy részletező felugró ablak, ha több információra lenne szüksége a felhasználónak a foglalás elbírálásának döntéséhez.



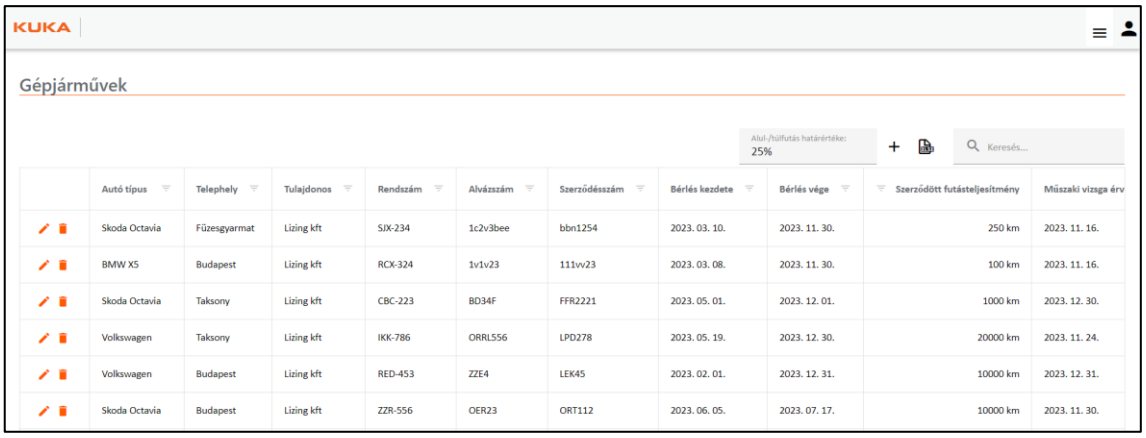
48. ábra: Technikai foglalás menü
Forrás: saját kép

A technikai foglalás felületen az Adminoknak és a Karbantartási joggal rendelkező felhasználóknak lehetőségük van a flottához tartozó gépkocsik szervizelésének a foglalására, ilyenkor a kocsira egy technikai foglalást adnak fel, ami megjelenik a többi foglalás között így a felhasználók láthatják, hogy egy adott kocsi mikor less szervízelve és mikor tudják újból lefoglalni maguknak. Ugyan ezen a felületen van lehetőségük van egy technikai foglalás törlésére.



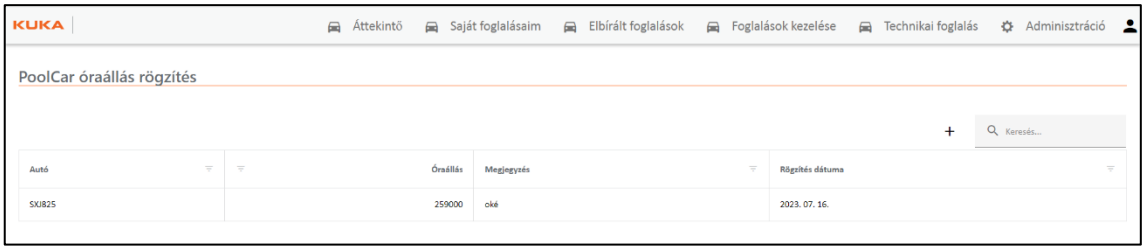
49. ábra: Jogosultságkezelés menü
Forrás: saját kép

Az adminisztráció alatt található jogosultságkezelés menüben lehetősége van az Adminoknak beállítani felhasználóknak különböző jogosultságokat, mint például: Admin vagy Karbantartás. A karbantartási jogkörhöz megkell adni egy telephelyet is, ahova tartozni fog. Ezt bármikor tudják módosítani, illetve törölni egy felhasználót.



50. ábra: Gépjárművek menü
Forrás: saját kép

Ugyan csak az adminisztráció alatt található gépjárművek menüben van lehetősége az Adminoknak és a Karbantartásnak kezelni a flottához tartozó gépkocsikat. Itt tudják felvenni az aktuális és új kocsikat a rendszerbe, illetve itt tudják módosítani vagy törölni is őket. A kocsikat és a hozzájuk tartozó adatokat kitudják exportálni Excelbe. Betudják állítani az alul-/túlfutási értéket, amit a rendszer a számolásokhoz használ.



51. ábra: Óraállás rögzítése menü
Forrás: saját kép

Szintén az adminisztráció alatt találunk egy óraállás rögzítés menüt, ahol az Adminok és a Karbantartás tudja rögzíteni a gépkocsikhoz tartozó kilométer óraállást. Ezzel tudják naprakészen tartani a kocsikat.

5. ÖSSZEFOGLALÁS ÉS TOVÁBBI FEJLESZTÉSI LEHETŐSÉGEK

5.1. A projekt összefoglalása

A projekt és maga a fejlesztés januártól egészen augusztusig tartott. Ez idő alatt rengeteg különböző tanult ismereteket kellet alkalmaznom. Meg kellett értenem a szoftver valamilyen szintű működését a megfelelő programkódok fejlesztéséhez. Természetesen az informatikai ismeretek nélkülözhetetlenek voltak a probléma megoldása során. Merítettem, a tanult C# és Java fejlesztésekből és web fejlesztési ismeretekből, de a szoftverhez kapcsolódó architektúra professzionális tervezése egy teljesen új kihívás volt számomra. Sok új ismerettel bővültem, mind a szoftverfejlesztés és mind pedig a projektmenedzsment területein belül. Azáltal, hogy végig jelen voltam egy projekt indítása és befejezése során, egy teljes összképet kaptam egy projekt folyamat lefolyásáról, amit a jövőben hasznos tapasztalattal tudok majd tovább vinni magammal. Nagyon sok új felülettel és szolgáltatással találkoztam és ismertem meg amik egyszerűsítik és segítik a fejlesztést, illetve a projekt nyomon követését. Ezeket a jövőben is folyamatos használattal fogom kezelni. A fejlesztés során próbáltunk a szoftverhez igazodó programot készíteni, amin a jövőbeli változások könnyedén eszközölhetőek, legyen szó jövőbeli fejlesztésekről vagy hibák javításáról. Elképzelhető, hogy az átadás után olyan hibák és észrevételek kerülnek felszínre, amelyek a jelen tervezési szakaszban még nem merültek fel. A dolgozat során bemutattam az elkészítendő szoftvert, majd bemutattam röviden a csapatot és a kitűzött célt. A kutatómunka során utána jártam a C# programozási nyelv sajátosságainak, a .NET keretrendszer működésének és az Entity Framework jelentőségének. A tervezés során bemutattam az architektúra egyes részeinek a működését és mivoltját, a fejlesztői környezet tulajdonságait és különböző eszközeit. A fejlesztést a projekt tervezésével és indításával kezdtem, hogy a folyamatok logikája könnyen értelmezhető és átlátható legyen. A szoftver különböző részeire bontva átláthatóbb volt a tényleges fejlesztési folyamat. Az elvégzett munka és a dolgozatban leírtak segítenek nekem abban, hogy meghatározzam, merre szeretnék tovább lépni a jövőben. Az elért eredmények és a projekt során megszerzett készségek alkalmassá tesznek arra, hogy aktív résztvevője legyek a szoftvertervezésnek, fejlesztésnek és üzemeltetésnek azokban a projektekben, amelyek komplex problémákra adnak megoldást. A projekt során történt tapasztalatok által megtanultam a csapatmunka,

kommunikáció és hatékony időmenedzsment fontosságát. Ezek a készségek nemcsak szakmailag, hanem személyes szinten is fejlesztették a kompetenciáimat, és a mindennapi életben is alkalmazhatóvá váltak. A projekt befejezése után rájöttem, hogy a folyamatos tanulás és fejlesztés elengedhetetlen ahhoz, hogy lépést tudjak tartani az informatikai világ dinamikus változásaival. Úgy érzem, hogy a projekt során szerzett készségek és az elért eredmények nem csupán egy konkrét projekt sikerét, hanem egy szélesebb perspektívát is nyitottak számomra az informatikai szakma területén. A dolgozat és a projekt végére érve rájöttem, hogy a folyamatos tanulás és a szakmai fejlődés kulcsfontosságú az informatikai szakmában való előrehaladáshoz. Úgy érzem, hogy megszilárdultak azok a kompetenciák és tapasztalatok, amelyekre építve további sikeres projekteken dolgozhatok és hozzájárulhatok az informatikai terület fejlődéséhez. A lefejlesztett szoftver következő nagy fázisa a projekt átadása és a szoftver utóélete, ahol elő jöhetnek esetleges hibák, illetve újabb igények is felmerülhetnek.

5.2. Fejlesztési lehetőségek

Az előző alfejezetben említett újabb igények és a bevezetésben bemutatott funkció, ami a szakdolgozat írása idejében még nem készült el az úgynevezett TripReg felület. Az új TripReg felületek két fontos részből állnak: az Adminok által használt adminisztrációs felületből és a cégen belüli vezetők által használt felhasználói felületből. Az Admin felület elsősorban azoknak a személyeknek szól, akik felelősek a céges autók kezeléséért és az ezzel kapcsolatos adminisztratív feladatokért. Itt az Adminok képesek új dolgozókat felvenni a rendszerbe, akik rendelkeznek céges autóval, és ezáltal TripReg felhasználóként vehetik igénybe a szolgáltatást. Az Adminoknak lehetőségük van céges autókat hozzárendelni, módosítani vagy épp törölni azokat. Az Adminok ugyancsak rögzíthetnek utakat és óraállásokat, valamint láthatják ezeket a műveleteket naplószerűen. Ez a funkció segíti az adminisztrációt és a nyomon követést. A másik felület, a TripReg felhasználói felület, a vezető beosztású személyeknek biztosítja a lehetőséget, hogy egyszerűen rögzítsék a céges autóikkal megtett utakat és óraállásokat. Ezen a felületen a vezetők aktuális információkat láthatnak az éves privát kilométer mennyiségükről, az összes eddig megtett út számáról, valamint a még rendelkezésre álló kilométerekről is. Emellett a rendszer továbbfejleszthető, hogy kielégítse a jövőbeli igényeket, és a használat során felmerülő hibák javítását is figyelembe veszi. Az új TripReg felületekkel

a céges autók kezelése és a vezetők számára elérhető információk naprakészen és könnyen kezelhetők lesznek, ezáltal hozzájárulva a hatékonyabb flottakezeléshez és az üzleti folyamatok optimalizálásához. A TripReg rendszer továbbfejlesztése érdekében a jövőben számos további igény és funkció is felmerülhet. Elsődlegesen fontos lehet a felhasználói élmény optimalizálása, hogy mind az Adminok, mind a vezetők könnyedén és hatékonyan tudják használni a rendszert. Ezen kívül elképzelhető, hogy a felhasználók igénylik majd olyan jelentéseket vagy statisztikákat, amelyek segítik az üzleti döntéshozatalban és a flottamenedzsment optimalizálásában. Az esetlegesen felmerülő hibák javítása is kulcsfontosságú feladat lesz a rendszer fenntartása és zavartalan működése érdekében. Fontos figyelemmel kísérni a felhasználók visszajelzéseit, hogy az esetleges problémákra gyorsan reagálhassunk és javíthassuk azokat. A rendszer biztonsága is kiemelten fontos, így a megfelelő adatvédelmi intézkedéseket és védelmi mechanizmusokat is be kell építeni a fejlesztések során. Az együttműködés és kommunikáció megkönnyítése érdekében lehetőség lehet olyan értesítési rendszer bevezetése, amely automatikusan értesíti az érintetteket, például amikor egy új autó kerül hozzárendelésre, vagy amikor egy vezető új utat rögzít. Ez tovább növelheti a rendszer átláthatóságát és felhasználóbarátságát. A jövőben kialakítható lenne akár egy mobilalkalmazás is, amely lehetővé teszi a felhasználók számára, hogy bárhol és bármikor hozzáférjenek a rendszerhez, például utak rögzítése, módosítása, kocsik foglalása vagy értesítések fogadása céljából. Az akadálymentesítés további fontos szempontként merül fel a szoftver továbbfejlesztése során. A felhasználóknak elérhetővé téve az alkalmazást és annak funkcióit azok számára is, akik esetleg különféle fogyatékosságokkal küzdenek, a rendszer még inkluzívabbá válik. A rendszer fejlesztése során érdemes figyelmet fordítani arra, hogy a felhasználói felület könnyen kezelhető legyen látássérült vagy más fogyatékossággal élő személyek számára is. Például, hangutasítások vagy nagyobb gombok alkalmazásával. A rendszerben bevezetett hangvezérlési lehetőségek segíthetnek azoknak, akiknek nehezére esik az érintőképernyők vagy a hagyományos gombok használata. A felhasználói felület színkombinációinak és betűméreteinek beállításával a látássérült vagy színtévesztő felhasználók is könnyen használhatják a rendszert.

IRODALOMJEGYZÉK

1. Eric Evans (2003): Domain-Driven Design: Tackling Complexity in the Heart of Software. pp. 24-301.
2. Martin Flower (2002): Patterns of Enterprise Application Architecture. pp. 116-122.
3. Dr. Ferenc Rudolf (2011): Szoftverkarbantartás, Egyetemi tananyag - Szegedi Tudományegyetem Természettudományi és Informatikai Kar Szoftverfejlesztés Tanszék. pp. 34-36.
4. Erich Gamma - Richard Helm - Ralph Johnson - John Vlissides (1994): Design Patterns Elements of Reusable Object-Oriented Software. pp. 107-117., 139-151., 273-282.
5. Ken Schwaber - Jeff Sutherland (2020): The Definitive Guide to Scrum: The Rules of the Game.
6. Ian Sommerville (2011): Software Engineering 9th Edition. pp. 56-234.

Elektronikus hivatkozás

1. What is Visual Studio?, 2023. május 05. https://learn.microsoft.com/hu-hu/visualstudio/get-started/visual-studio-ide?view=vs-2022 (Utolsó letöltés: 2023. szeptember 12.)
2. Robert Sheldon: C# (C-Sharp), 2022. december https://www.techtarget.com/whatis/definition/C-Sharp (Utolsó letöltés: 2023. szeptember 12.)
3. A tour of the C# language, 2023. május 04. https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (Utolsó letöltés: 2023. szeptember 12.)
4. Overview of .NET Framework, 2023. március 30. https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview (Utolsó letöltés: 2023. szeptember 13.)
5. What is .NET Framework?, https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework (Utolsó letöltés: 2023. szeptember 13.)
6. What is Entity Framework?, https://www.entityframeworktutorial.net/what-is-entityframework.aspx (Utolsó letöltés: 2023. szeptember 13.)

7. Design the infrastructure persistence layer, 2023. február 21. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design (Utolsó letöltés: 2023. szeptember 14.)
8. Pranaya Rout: Unit Of Work in Repository Pattern, 2019. https://dotnettutorials.net/lesson/unit-of-work-csharp-mvc/ (Utolsó letöltés: 2023. szeptember 14.)
9. Martin Flower: Unit of Work, https://martinfowler.com/eaaCatalog/unitOfWork.html (Utolsó letöltés: 2023. szeptember 14.)
10. Martin Flower: CQRS, 2011. július 14. https://martinfowler.com/bliki/CQRS.html (Utolsó letöltés: 2023. szeptember 15.)
11. Gyártó metódus programtervezési minta, 2020. december 6. https://hu.wikipedia.org/wiki/Gy%C3%A1rt%C3%B3_met%C3%B3dus_programtervez%C3%A9si_minta (Utolsó letöltés: 2023. szeptember 15.)
12. Scrum módszertan érthetően: Ezek a Scrum legfontosabb lemei, 2021. szeptember 3. https://promanconsulting.hu/scrum-modszertan-alapok/ (Utolsó letöltés: 2023. szeptember 16.)
13. Guide to Conducting an Event Storming Session, 2023. január 10. https://creately.com/blog/meeting-visual-collaboration/event-storming (Utolsó letöltés: 2023. szeptember 21.)
14. Greg Young: CQRS Documents, 2012. december 28. https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf (Utolsó letöltés: 2023. szeptember 16.)

TARTALMI KIVONAT

A "Flottakezelő és foglalást lebonyolító webalkalmazás fejlesztése" című szakdolgozat azt a célt tűzi ki, hogy bemutassa egy modern, felhasználóbarát webalkalmazás tervezését és fejlesztését, melynek fő funkciója a flottakezelés és járműfoglalás könnyű és hatékony módjának biztosítása. A szakdolgozat bemutatja a webalkalmazás tervezési fázisát, amely magában foglalja a felhasználói igények és követelmények elemzését. A tervezés során hangsúlyt fektetnek a felhasználói élmény javítására és az intuitív használhatóságra, hogy a flottakezelők és a járműfoglalók könnyedén ki tudják használni az alkalmazást. A fejlesztés részletezi a technológiai megoldásokat, amelyekre az alkalmazás építése során támaszkodtak. Az alkalmazás keretrendszerét, felhasználói felület kialakítását és egyéb fontos technikai döntéseket részletesen ismerteti. Végül, a dolgozat összegzi az elvégzett munkát, és megvizsgálja az eredményeket, beleértve az alkalmazás előnyeit a flottakezelés és járműfoglalás területén. Emellett röviden értékeli az alkalmazás jövőbeli fejlesztési lehetőségeit és kihívásait. A "Flottakezelő és foglalást lebonyolító webalkalmazás fejlesztése" című szakdolgozat olyan értékes információkat nyújt, amelyek segíthetnek a flottakezelés és járműfoglalás hatékonyabbá tételében, valamint az ilyen típusú webalkalmazások tervezésében és fejlesztésében érdekelt szakembereknek és vállalkozásoknak.

SUMMARY

The thesis aims to present the design and development of a modern, user-friendly web application, the main function of which is to provide an easy and efficient way of fleet management and vehicle reservation. The thesis presents the design phase of the web application, which includes the analysis of user needs and requirements. The design focuses on improving the user experience and intuitive usability so that fleet managers and vehicle bookers can easily use the app. The development details the technological solutions that were relied upon to build the application. It describes in detail the framework of the application, the design of the user interface and other important technical decisions. Finally, the paper summarizes the work done and examines the results, including the benefits of the application in the field of fleet management and vehicle reservation. It also briefly evaluates the future development opportunities and challenges of the application. The thesis provides valuable information that can help professionals and businesses interested in making fleet management and vehicle reservation more efficient, as well as in the design and development of this type of web application.