Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott InformatikaiTanszék

Princz Tamás

TOTAL REACT: REAKTÍV MINTA ALKALMAZÁSA ADATVEZÉRELT .NET ALKALMAZÁSOK MINDEN RÉTEGÉBEN

KONZULENS

Simon Gábor

BUDAPEST, 2022

Összefoglaló

Összefoglaló

Napjainkban nagyon elterjedtek a különböző reaktív jellegű architekturális minták, komponensek, melyek alapja, hogy az alkalmazáslogika az adatok aszinkron folyamára reagál. Ilyen adatfolyamok lehetnek felhasználói interakciók vagy akár más alkalmazásoktól származó üzenetek is. Míg a klasszikus többrétegű kialakításban a felhasználó az állapot egy statikus pillanatfelvételén dolgozik, addig a reaktív modellben élő, gyakorlatilag azonnal frissülő adathalmazon. Mindez sokkal gazdagabb, élőbb, interaktívabb felhasználói élményt eredményez.

Dolgozatomban arra törekedtem, hogy megalkossak egy olyan architektúrát, ahol az adatáramlás minden rétegben és a rétegek között is reaktív módon történik az adatbázis rétegtől kiindulva egészen a felhasználói felületig. Ezt követően meg is valósítottam az architektúrát .NET platformon. Az alkalmazás bemutatásakor ismertetem, hogy pusztán a kialakított reszponzív architektúrából és az ezt támogató felhasznált komponensekből adódóan milyen rendszerszintű képességeket szerez a rendszer. Ezen túl gyakorlati példákon keresztül szemléltetem, hogy milyen teljesítménynövelő, felhasználói élményt gazdagító funkciók bevezetésére ad lehetőséget az alkalmazott megközelítés.

Végül összevetem ezen előbbi megoldást egy üzletileg hasonló funkcionalitást megvalósító, hagyományos háromrétegű architektúrát követő referencia megvalósítással különböző mérnöki szempontok, többek között reszponzivitás, karbantarthatóság, forráskódmennyiség és teljesítmény alapján.

Tartalom

Összefoglaló **3**

1 Reaktív programozás **6**

1.1 Összehasonlítás az imperatív programozással 6

1.2 Change propagation 7

1.3 Eseményfolyamok 8

1.4 Reaktív keretrendszerek 10

2 Reaktív adatkezelés megvalósítása **12**

2.1 Reaktív adatkezelés szerver oldalon 13

2.2 Kliens-szerver kommunikáció reaktív módon 17

2.2.1 Websocket 17

2.2.2 Fusion – Replica Service 18

2.2.3 Egyéb alternatívák 20

2.3 Backend-adatbázis kommunikáció reaktív módon 21

2.3.1 Change Data Capture 21

2.3.2 Debezium és Kafka 23

2.3.3 Egyéb alternatívák 24

3 Felhasznált technológiák **26**

3.1.1 ASP.NET Core 26

3.1.2 Hangfire 27

3.1.3 Kafka .NET 27

3.1.4 EF Core 27

3.1.5 ASP .NET Core Identity 27

3.1.6 Blazor WASM 27

3.1.7 RestEase 29

3.1.8 Docker, docker-compose 29

4 Esettanulmány **30**

4.1 Architektúra 31

4.2 Projektek és futtatási környezet bemutatása 33

4.3 Backend funkciók implementálása 34

4.3.1 Termék alapdadatok lekérdezése és szerkesztése 34

4.3.2 Több termék alapadatainak listázása 36

4.3.3 Komplex lekérdezések megvalósítása - termékek közti keresés 37
4.3.4 Időalapú invalidáció – akciók kezelése 40
4.3.5 Adatbázis változás külső forrásból - termék kezelő CRM 41
4.3.6 Jogosultágtól függő adatok kezelése – saját megrendeléseim lekérdezése ... 43
4.4 Frontend funkciók implementálása 45
4.4.1 Websocket invalidációs mechanizmus bekapcsolása 45
4.4.2 Adatok megosztása komponensek között - kosárkezelés 47
4.4.3 Felület állapota mint lokális függőség – termék keresés 52
5 Összefoglalás, kitekintés 55
5.1 Előnyök, rendszerszintű többletfunkciók 55
5.1.1 Cachelés szerver és kliens oldalon 55
5.1.2 UI komponensek függetlensége, egyszerűbb API kommunikáció 55
5.1.3 Jobb felhasználói élmény, reszponzivitás 56
5.1.4 Adatbázis szintű integráció más alkalmazásokkal 56
5.2 Hátrányok, architektúra korlátai 57
5.2.1 Nagyobb backend komplexitás 57
5.2.2 Nagyobb memória használat 58
5.2.3 Fusion könyvtár 58
5.2.4 Kafka infrastruktúra felállítása nagy plusz üzemeltetési teher 58
5.3 Milyen esetekben ajánlanám ezt az architektúrát 59
5.4 Kitekintés, továbbfejlesztési lehetőségek 59
6 Irodalomjegyzék 61
Függelék 67
6.1 docker-compose.yaml 67

1 Reaktív programozás

Az adatvezérelt alkalmazások fejlesztésben rejlő legnagyobb kihívást gyakran az állapot kezelésének nehézsége okozza. Reagálni a különböző forrásokból történő adatváltozásokra és végigvezetni azokat az alkalmazás különböző részein gyakran igen nagy komplexitással jár. Ezt a komplexitást egyrészt az okozza, hogy ugyanazt az adatot az alkalmazáson belül több helyen jelenítjük meg, illetve több különböző reprezentációban tároljuk. Ez azt eredményezi, hogy az adat változásakor sok módosító műveletet kell végrehajtanunk ahhoz, hogy az adatunk mindenhol konzisztens legyen. Másrészről pedig a különböző adatrészek között, illetve az egyes adatrészek és az alkalmazás futás közbeni állapota között is lehetnek különböző függések és egymásra hatások. Például egy adatérték megváltozásának hatására egy adott felület teljesen átalakulhat, gombok tűnhetnek el, jelenhetnek meg vagy tiltódhatnak le. Ezen egymásra hatások kezelése gyakran eredményezhet nehezen olvasható és karbantartható forráskódot. Dolgozatomban azt vizsgáltam, hogy a reaktív programozás mintája hogyan nyújthat segítséget az itt felvázolt nehézségekre, illetve hogy milyen előnyökkel járhat, ha ezt a mintát nem csak komponensek szintjén használjuk, hanem kiterjesztjük azt egy N-rétegű architektúra minden rétegére.

1.1 Összehasonlítás az imperatív programozással

Reaktív programozás [1] egy programozási minta, ami elsősorban eseményfolyamokkal és az adat változásainak továbbításával foglalkozik. Ennek a mintának a segítségével függőségeket adhatunk meg az adataink között és automatikusan újra számíthatjuk azok értékét, a függőségek megváltozása esetén. Ahhoz, hogy megértsük, hogy ez pontosan mit jelent érdemes lehet összevetnünk ezt a mintát a sokszor már alapértelmezésként használt imperatív szemlélettel.

A legtöbb ma használt programozási nyelv az imperatív megközelítést követi, ez azt jelenti, hogy az alkalmazás állapotának módosítása az egyes utasítások végrehajtásának hatására történik. Például egy az „$a = b + c$” kifejezés azt jelenti, hogy $\boldsymbol{a}$ változó legyen egyenlő b és c változók az utasítás kiértékelésekor vett értékeinek összegével. De ez nem jelent semmilyen kapcsolatot a három változó között tehát, ha a későbbiekben változik b vagy c értéke az nem fogja befolyásolni az $\boldsymbol{a}$ változó értékét.

Ezzel szemben reaktív programozás használata esetén ugyanez az „`a = b + c`” kifejezés magában hordozza azt a tulajdonságot, hogy ***a*** változó minden időpillanatban meg kell, hogy egyezzen *b* és *c* változók értékével, tehát ha a későbbiekben ***b*** vagy ***c*** változik azt akkor *a* értéke is újraszámításra kerül. Az alábbi egyszerű kód részleten is jól látható ez a különbség a két megközelítésben.

```
var b = 1
var c = 2
var a = b + c
b = 10

// Imperatív esetben
console.log(a) // 3

// Reaktív esetben
console.log(a) // 12
```

1.2 Change propagation

Reaktív programozás alkalmazása során tehát függőségeket fogalmazunk meg az adataink között, ezeket a függőségeket gyakran tároljuk valamilyen gráf formájában, ami segít annak a meghatározásában, hogy egy adott érték változása esetén mely más értékeket kell újra számolnunk. Fontos kérdés viszont, hogy hogyan tudunk értesülni egy adott adat megváltozásáról. Ennek a problémának a kezelésére többféle megközelítés is létezik.

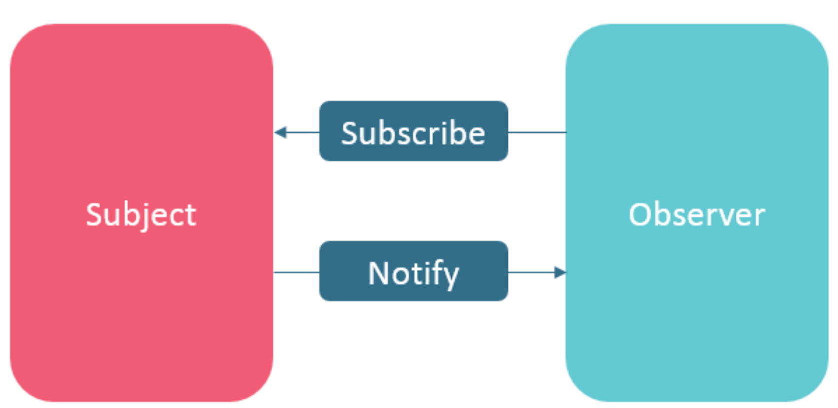
- **Pull:** A pull megközelítés esetén a fogyasztó (előző példában az ***a*** változó) rendszeresen lekérdezi a függőségeinek az aktuális értékét és ez alapján újra számítja a saját értékét, ha szükséges. Ezt a megközelítést más szóval szokás még **polling**-nak is nevezni.
- **Push:** A fogyasztó értesítést kap a függőségeitől, ha azok értéke megváltozik. Ezek az értesítések tartalmazzák a függőség új értékét is, így további kommunikáció nem szükséges.
- **Push-pull:** A fogyasztó szintén értesítést kap a függőségeitől, azonban ez az értesítés nem tartalmazza az új értéket, csak annyit, hogy „valami változás történt”, ezért a fogyasztónak az értesítés után még le kell kérdeznie az új értéket. Ez értelemszerűen több kommunikációt eredményez viszont azzal az előnnyel jár,

hogy a fogyasztó csak abban az esetben fogja lekérdezni a legújabb értéket, ha tényleg szüksége van rá, ezzel pedig csökkenthetjük az összességében átvitt adatmennyiséget.

Alapvetően a **push** alapú megközelítéseket szoktuk előnyben részesíteni, illetve „igazán reaktív" -nak tekinteni. A **pull** megközelítés hátránya, hogy sokszor fölöslegesen kérdezzük le a függőségeink értékét, ha azok nem változtak, sokkal előnyösebb, ha ők tudnak értesíteni minket a változásról.

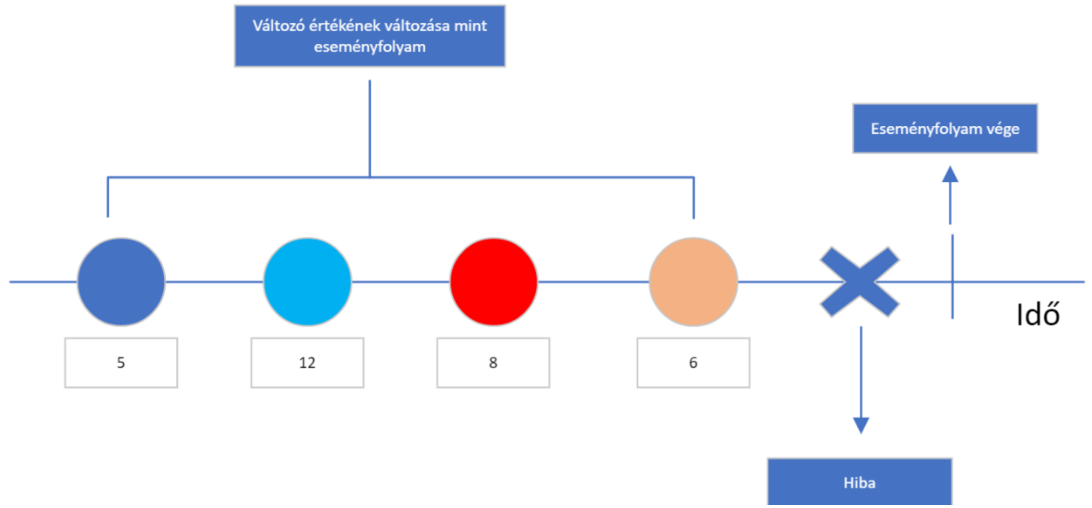
1.3 Eseményfolyamok

A reaktív programozás egyik legfontosabb absztrakciója az eseményfolyam fogalma. Egy eseményfolyam nem más, mint bekövetkezett események időben rendezett sorozata. Ilyen eseményfolyam lehet például egy HTML klikk esemény amire fel tudunk iratkozni és a bekövetkezés hatására különböző műveleteket végezni, de az előző példában vett ***a***, ***b*** és ***c*** változók is kezelhetőek úgy, mint egy-egy eseményfolyam, ahol az egyes események nem mások, mint a változók új értékei. Ez a reaktív programozás egyik alapötlete, hogy (szinte) mindent megpróbálunk eseményfolyamként kezelni és ezeket az esemény folyamokat aszinkron módon dolgozzuk fel azáltal, hogy függvényeket definiálunk melyek az egyes események bekövetkezésekor futnak le. Ez gyakorlatilag tekinthető az **Observer** [2] tervezési minta megvalósításának, ahol az eseményfolyam a *Subject* a feldolgozó függvények pedig az *Observer* szerepét töltik be.



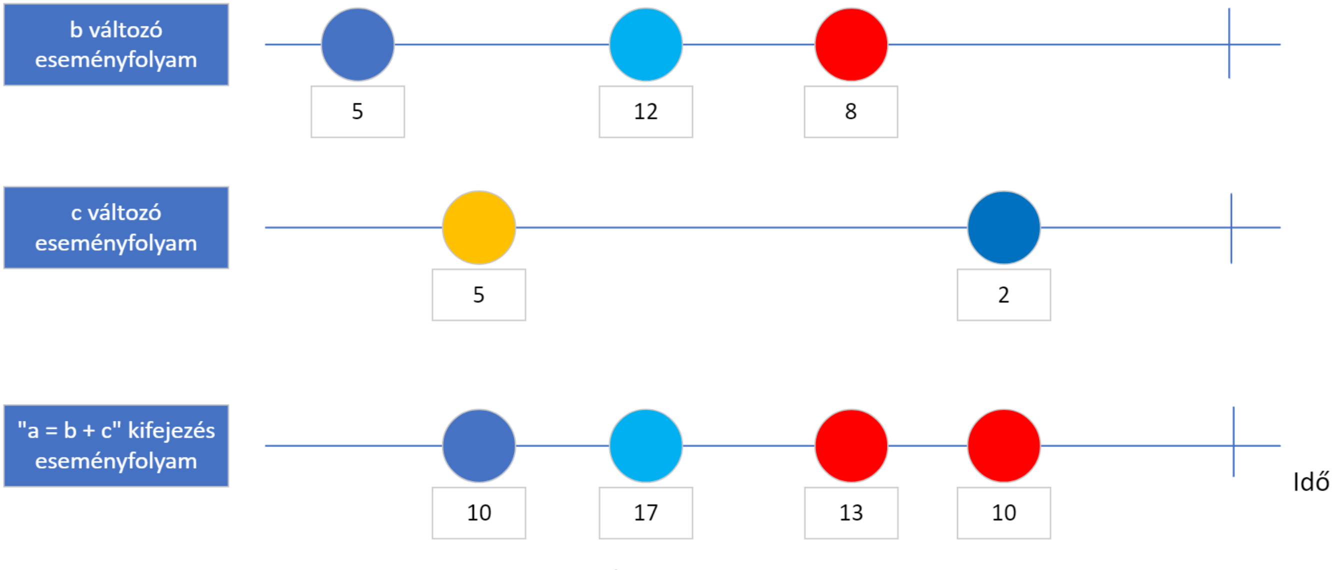
1. ábra Observer tervezési minta [42]

Annak, hogy a különböző adatokat eseményfolyamként kezeljük több előnye is van. Az eseményfolyamokkal különböző műveleteket tudunk végrehajtani, például megadhatunk egy transzformációt, ami az esemény folyam minden elemére lefut, ezzel egy új esemény folyamot kapva. Ezen kívül leszűrhetjük csak a számunkra fontos eseményeket, vagy csoportosíthatjuk a bekövetkezett eseményeket a bekövetkezés ideje szerint, például egy klikk eseményfolyamot idő szerint csoportosítva szintén kaphatunk egy új eseményfolyamot, ami dupla klikk eseményeket tartalmazza. Az egyes eseményfolyamok bemenetként is szolgálhatnak más eseményfolyamok számára, mint az „$a = b + c$” kifejezés példájában.



2. ábra Egy változó, mint eseményfolyam

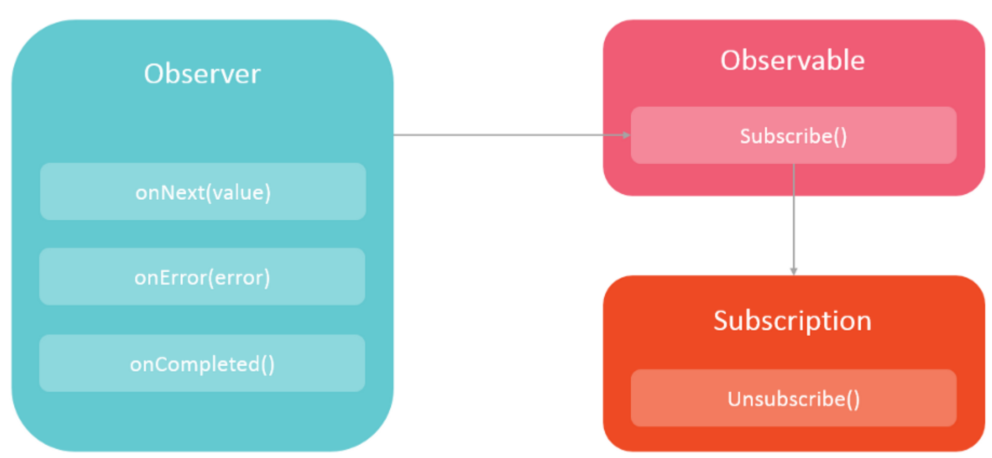
Az eseményfolyamokat és a velük végzett műveleteket ún. marble diagramok [3] segítségével szokták szemléltetni. A fenti ábrán is egy ilyen diagram látható, amin egy darab változó szerepel, mint egy eseményfolyam. Általában 3 féle eseményt szoktunk megkülönböztetni, az **új értékeket** tartalmazó eseményeket, a **hibát jelző** eseményeket és az **eseményfolyam végét** jelző eseményt. Az előző példákban használt „$a = b + c$” kifejezés is könnyen ábrázolható ebben a formában.



3. ábra „`a = b + c`" eseményfolyam

1.4 Reaktív keretrendszerek

A reaktív programozás mintája napjainkban igen népszerű. Léteznek teljesen reaktív programozási nyelvek, mint például az **Elm** [4] amelyet leginkább webes felületek leírására használnak. Vannak keretrendszerek melyek bevezettek reaktív szintaxis elemeket, mint például a **Svelte** [5] ahol létezik a „reaktív értékadás" a „`$`" operátorral, például „`$: a = b * 2`" esetén `a` értéke mindig `b` duplája lesz. De a legnépszerűbb könyvtár, ami a reaktív programozás mintáját követi valószínűleg a **ReactiveX** [6].



4. ábra ReactiveX Observable [42]

A Reactive X könyvtár bevezeti az `Observable` nevű absztrakciót, ami nagyon hasonló az előzőekben tárgyalt eseményfolyamokhoz. A könyvtár rengeteg eszközt ad a

kezünkbe ahhoz, hogy különböző adatokat egységesen `Observable`-ként tudjunk kezelni és hogy feliratkozzunk ezen adatok változására a `subscribe` függvény segítségével. Az ilyen `Observable` adatokkal nagyon sok féle műveletet tudunk végezni, itt található egy lista a ReactiveX által definiált operátorokról [7] melyekhez az előző példákban használt marble diagramokkal készült szemléltetések is tartoznak. Maga a könyvtár 17 programozási nyelven elérhető, talán a Javascript nyelven készült implementációja az RxJs a legnépszerűbb, de a Java, Swift és C# nyelven készült implementációkat is sok helyen használják.

Sok könyvtárat találhatunk tehát ami a reaktív megközelítést követi, azonban ezeket általában csak egy adott rétegen belül szokták használni. Például nagyon népszerűek ezek a minták a kliens oldali alkalmazásokban, ahol ugyanaz az adat a felületen több helyen jelenik meg és ezt egy eseményfolyamként kezelve könnyű konzisztensen tartanunk. Ezzel szemben én a továbbiakban azt próbálom megvizsgálni, hogy hogyan lehetne ezt a mintát nem csak egy adott rétegen belül, hanem több réteg között lévő adatáramlás esetén felhasználni.

2 Reaktív adatkezelés megvalósítása

Az előző fejezetben ismertettem a reaktív programozás főbb jellemzőit és absztrakcióit, de a konkrét implementációjukról nem beszéltem. Pedig fontos kérdés, hogy hogyan történik az egyes függvények közti függőségek kezelése vagy, hogy mégis hogyan tudunk értesülni egy adott adat megváltozásáról, főként akkor, ha az valamilyen külső forrásból származik. Ebben a fejezetben ismertetem azokat a reaktív megközelítést támogató plusz komponenseket és technológiákat, amelyeket felhasználtam az elkészült architektúrámban, illetve megemlítek pár alternatív választási lehetőséget is, amelyek hasonló célokat szolgálnak és más kontextusban szintén jó választások lehetnének.

Ahogy korábban is említettem, a reaktív minták alkalmazása viszonylag egyszerű amikor azt csak egy adott rétegen belül használjuk, vagy például, ha egy offline asztali alkalmazást készítünk, ahol az adatainkat a memóriában tároljuk és azok csak a felületi elemekkel való interakció hatására változhatnak meg. Egy klasszikus N-rétegű webalkalmazás esetén a helyzet már sokkal bonyolultabb, hiszen az általunk használt és megjelenített adatok gyakran egy másik rétegből érkeznek valamilyen hálózati kommunikáció útján. Ilyenkor nehezebben tudunk értesülni ezen adatok megváltozásáról, mert ezt a különböző kommunikációs protokollok nem feltétlenül támogatják az ilyen jellegű értesítéseket.

Törekedtem arra, hogy a kialakított reaktív architektúra részben hasonló építőlemeket használjon, mint a tradicionális N-rétegű megoldás. Ennek megfelelően az egyes rétegeket itt is egy frontend alkalmazás, egy **REST API**-t tartalmazó backend alkalmazás és egy **relációs adatbázis** segítségével alkottam meg, a különbség mindössze a rétegek közti kommunikációs megoldásokban jelenik meg. Ennek egyrészt az az előnye, hogy ez a felépítés már sok fejlesztő számára ismert lesz, és ezáltal számukra ez a megoldás is könnyebben érthető és elsajátítható lehet. Másrészről pedig sok alkalmazás készült már el, ami ezt a megközelítést követi és ilyen módon ezek átalakítása könnyebb lehet, amennyiben úgy döntünk, hogy az itt felvázolt megközelítést akarjuk követni.

2.1 Reaktív adatkezelés szerver oldalon

Reaktív programozás esetén a program loginkánkat általában sok kisebb függvény kompozíciójából építjük fel, amelyek rendelkeznek ún. **tiszta függvény** [8] (pure function) tulajdonsággal. Ez a tulajdonság annyit tesz, hogy az adott függvény azonos bemeneti paraméterekre mindig ugyanazt az eredményt produkálja. Ennek alapvetően két főbb előnye van.

- Ha egy adott függvény függőségei megváltoznak, azt újra kell számítanunk, ha ezek a függőségek tiszta függvények akkor ezeknek a kiszámítási sorrendje tetszőleges lehet, illetve akár szabadon párhuzamosíthatjuk is őket.
- Ezek a tiszta függvények könnyen cachelhetőek, ami a gyakori újraszámítás miatt praktikus lehet

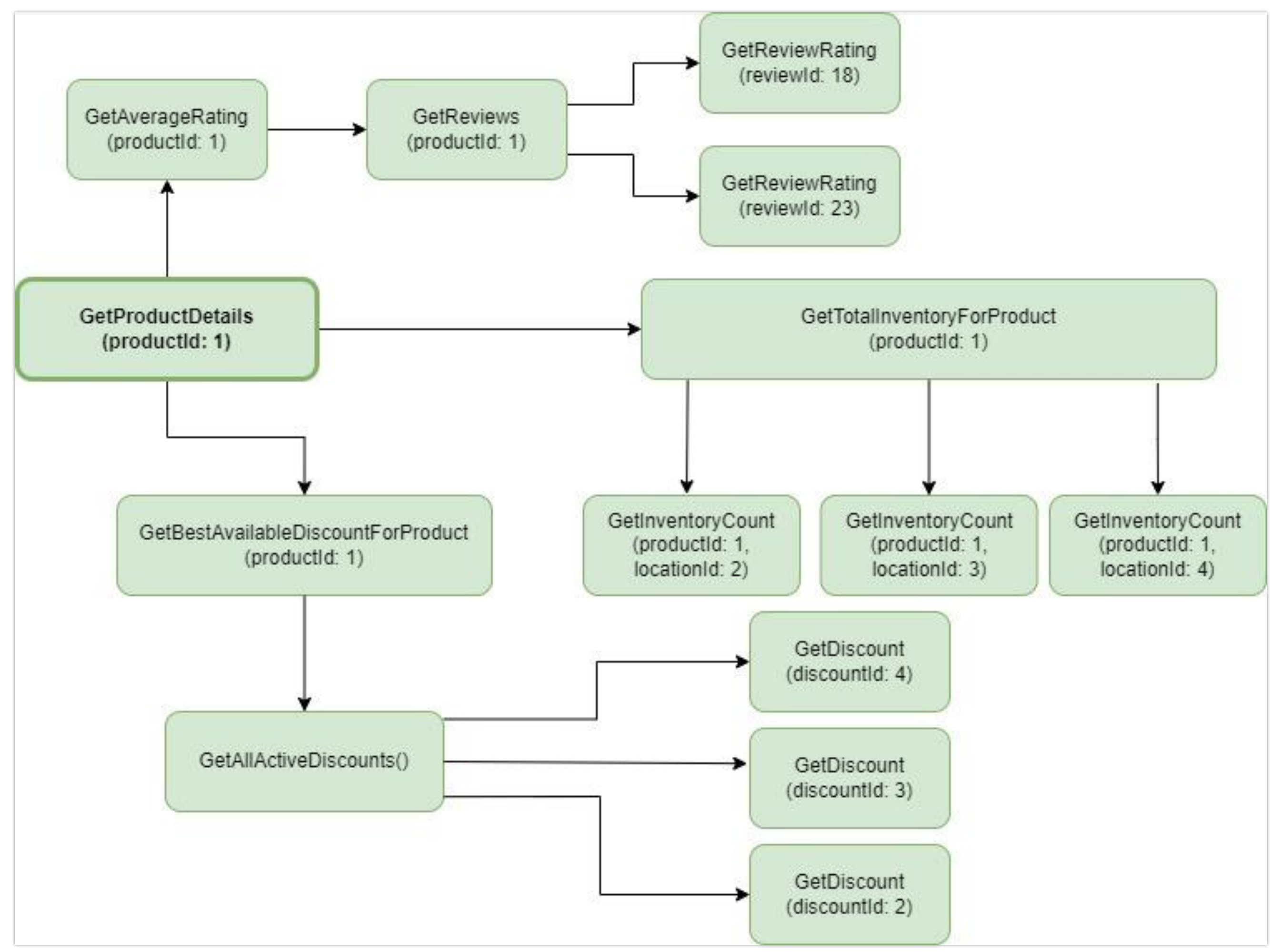
Viszont az a probléma, hogy egy backend alkalmazás esetén általában egy külső adatbázisból kérdezünk le adatokat, így ez a tulajdonság nehezen tud teljesülni. Ha van egy függvényünk, ami mondjuk egy ID bemeneti paraméter alapján kérdezi le egy felhasználó adatait adatbázisból, akkor a konkrét visszatérési érték nem csak a paraméterektől, hanem az adatbázis tartalmától is függeni fog. A relációs adatbázist azonban nem tudjuk klasszikus értelemben vett függőségként kezelni mert nem nyújt olyan interfészt, ahol értesíteni tudna minket egy általunk lekérdezett adat megváltozásáról.

Emiatt nekünk kell kezelnünk, hogy az általunk végzett[1] adatmódosító műveletek milyen lekérdező függvények eredményét befolyásolhatják és ezeket a függvényeket meg kell jelölnünk, hogy szükséges az újra számításuk. Backend alkalmazás esetén ez általában azt fogja jelenteni, hogy a hozzájuk tartozó adatbázis lekérdezést újra le kell majd futtatnunk. Erre a megjelölésre fogok a későbbiekben **invalidáció** néven hivatkozni. Ezt az invalidációt és újraszámítást könnyíti meg a **Fusion** osztálykönyvtár.

[1] Fontos, hogy itt azt az esetet próbálom kezelni amikor mi végzünk adatbázis módosítást, ha az adatbázis tartalmát más is módosítja akkor kicsit más megoldás kell, erről 2.3 fejezetben beszélek bővebben.

Fusion – Compute Service

A Fusion [9] egy nyílt forráskódú .NET osztálykönyvtár, ami a reaktív adatkezelésre és a valós idejű UI készítésre ad újszerű megoldást. Itt röviden ismertetem a könyvtár főbb funkcióit és hogy azok hogyan segítik a függőségek kezelését és az invalidációt. Egyik legfontosabb Fusion fogalom a `ComputeMethod` [10], amit egy attribútum szintjén tudunk használni. Egyrészt az ezen attribútummal ellátott függvények kimenete automatikusan cachelésre kerül, másrészt a Fusion eltárolja függőségként azt, hogy ez a függvény milyen más `ComputeMethod` függvényeket hív meg és ha ezek értéke változik, akkor a függvény eredményét invalidáljuk. Az alábbi ábrán látható egy ComputeMethod függvények közti vázlatos függőségi gráf, ami egy webshopban (későbbi példa alkalmazás) lévő termék adatait lekérdező függvényből (GetProductDetails) indul ki.



5. ábra Fusion függőségi gráf

Ha egy osztályunk ilyen függvényeket használ, akkor azt regisztrálnunk kell a Fusion számára, mint egy *ComputeService*. Az ilyen *ComputeService* osztályokból a Fusion futásidőben **dinamikus proxy osztályokat** fog generálni, melyek leszármaznak az általunk definiált osztályokból. Ezekben a generált osztályokban a *ComputeMethod* attribútummal ellátott függvények visszatérési értékét a Fusion egy *Computed* [11] nevű generikus wrapper osztályba csomagolja.

Ez az osztály valósítja meg igazából a függvényünk visszatérési értékének cachelését, illetve tárolja az adott függvény függőségeit és képes értesülni arról, ha azok változnak. Itt látható ezen osztály forráskódjának egy részlete, ami segíthet a működés megértésében.

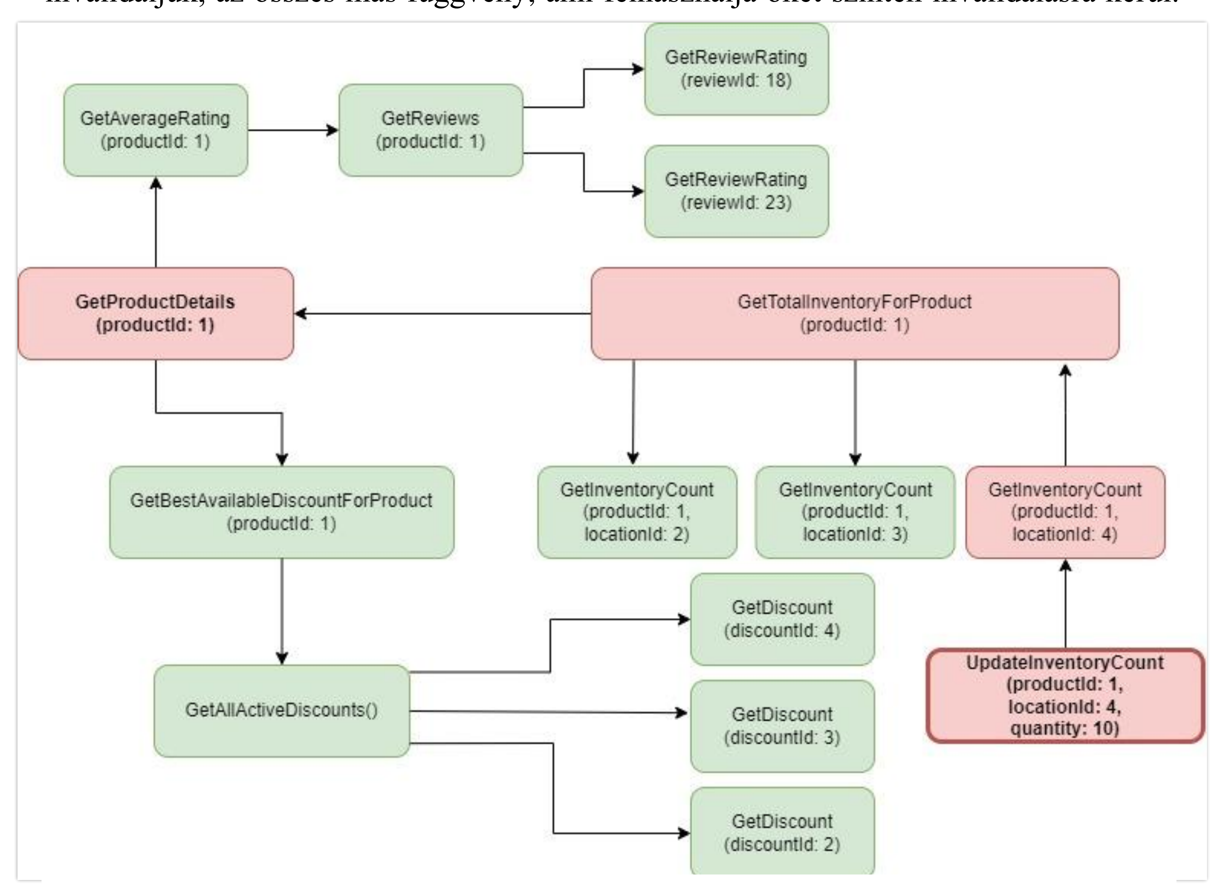
```
public abstract class Computed<T> : IComputed, IComputedImpl, IResult<T>
{
    private Result<T> _output;
    private RefHashSetSlim3<IComputedImpl> used;
    private HashSetSlim3<(ComputedInput Input, LTag Version)> _usedBy;
    private event Action<IComputed>? invalidated;

    public ComputedInput Input { get; }
    public ConsistencyState => (ConsistencyState) _state;
    public IFunction<T> Function => (IFunction<T>) Input.Function;
    public bool Invalidate() { //... }

}
```

A Computed objektumok rendelkeznek állapottal, ez tárolja a *ConsistencyState* mező, ennek három lehetséges értéke van: *Computing*, azaz éppen számítás alatt van az új érték, *Consistent*, azaz az utoljára kiszámított érték a lehető legfrissebb ha valaki lekérdezi ezt vissza is adhatjuk egyből, illetve *Invalidated*, ami azt jelzi, hogy tárolt érték már elavult mert vagy ez a konkrét Computed objektum, vagy valamelyik függősége invalidálásra került, ezért ha valaki lekérdezi akkor újra ki kell számítanunk. Ennek megfelelően ez az implementáció igazából a **push-pull** modellt követi, hiszen a függőségeink csak az változás tényéről értesítenek minket, a tényleges újraszámítás az valamilyen szinten a fogyasztón múlik.

Az utoljára kiszámított értéket konkrétan az *_output* mező tárolja. A *_used* és *_usedBy* kollekciók azt tárolják, hogy milyen más Computed példányoktól függünk, illetve, hogy kik azok, akik pedig tőlünk függenek. A *Function* tulajdonság egy

hivatkozás az eredeti függvényünkre, amivel ki tudjuk számítani a legújabb értéket, ez szintén tartalmazza a függvénynek **átadandó paramétereket**, ezt a cache bejegyzés kulcsának előállításához is felhasználjuk majd. Az `_invalidated` egy esemény, ami akkor fog elsülni amikor `Invalidated` állapotba kerül az adott objektum, szükség esetén erre manuálisan is feliratkozhatunk kívülről. Az `Invalidate()` pedig a függvény, ami az invalidációt elvégzi, de ezt általában nem konkrétan mi hívjuk meg, hanem különböző segédfüggvényeken keresztül automatikusan meghívásra kerül.

Visszatérve az eredeti problémára, lesznek tehát olyan függvényeink, amelyek külső adatforrást, jellemzően adatbázist használnak, ezeket nekünk kell **invalidálnunk.** A Fusion által javasolt [12] megközelítés az, hogy ezek a külső adatforrást lekérdező függvények legyenek egyszerűek (low level methods), például ID alapján kérdezzenek le egy adott rekordot, illetve helyezkedjenek el a függőségi gráf utolsó szintjén. Az első tulajdonság azért előnyös, mert megkönnyíti az invalidációt, egy adott ID-val rendelkező elem szerkesztésekor invalidáljuk a hozzá tartozó adat lekérdező függvényt, ugyanazzal az ID-val. A második tulajdonság pedig azért jó számunkra, mert a Fusion biztosítja a **Cascading invalidation**-t, tehát azzal, hogy ezeket az egyszerű függvényeket invalidáljuk, az összes más függvény, ami felhasználja őket szintén invalidálásra kerül.



6. ábra Fusion Cascading Invalidation

Az fenti ábrán látható egy ilyen invalidáció, egy termék raktárkészletet módosító függvény (`UpdateInventoryCount`) határásra. Az ilyen invalidáló függvényeket szintén egy attribútummal kell ellátunk, erre szolgál a `CommandHandler` osztály. Az ezzel az attribútummal ellátott függvények törzsében egyszerűen felsorolhatjuk, hogy milyen ComputeMethod függvények, milyen paraméterekre kiszámolt eredményét akarjuk invalidálni, erre a felsorolásra a későbbiekben **invalidációs blokk** néven fogok hivatkozni.

Egy másik fontos Fusion osztály, ami a későbbi implementációs részletekben is meg fog jelenni az az `EntityResolver`. Ez gyakorlatilag ID alapján adatbázis entitásokat lekérdező függvényeket ad nekünk, de a háttérben képes ezeket függvény hívásokat egy adatbázis kérésben végrehajtani. Ez azért hasznos mert ezzel az előbbi ábrákhoz hasonló függőségi gráfokat kevesebb tényleges adatbázis kérésben tudunk kiépíteni.

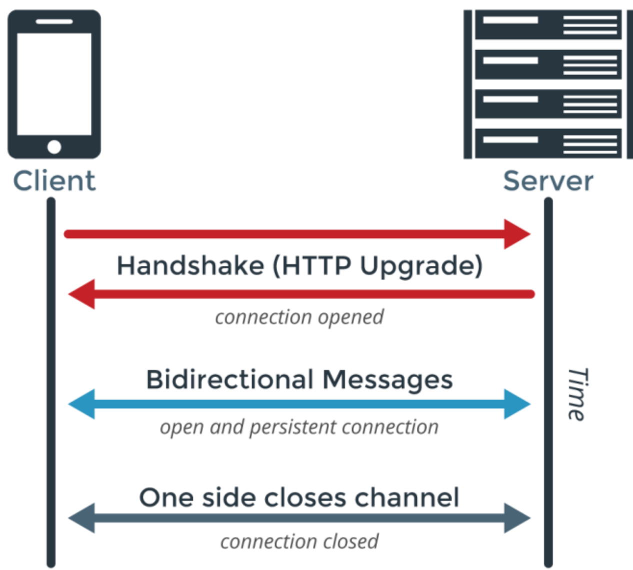
2.2 Kliens-szerver kommunikáció reaktív módon

Tekintsük azt az esetet amikor a frontend alkalmazásunkban használunk a backendről származó adatokat, ezeket ilyenkor általában egy HTTP hívás segítségével kérdezzük le. A korábban már említett RxJs könyvtár lehetőséget ad arra, hogy egy ilyen HTTP lekérdezést úgy kezeljünk mintha az egy eseményfolyam (Observable) lenne, azonban ez csak látszólagos megoldás. Ugyanis a HTTP az egy kérés-válasz protokoll, az általunk küldött adat lekérdezésre pontosan 1 válasz fog érkezni, eköré felépíthetünk egy absztrakciót és kezelhetjük úgy kliens oldalon mintha ez egy esemény folyam lenne, de attól még nem fogunk tudni értesítést kapni az új értékekről mert erre a kommunikációs protokoll jelenleg elterjedt és széles körben támogatott 1.1-es verziója nem ad lehetőséget. HTTP/2 vagy HTTP/3 használatával már megvalósítható lenne a két irányú kommunikáció azonban ennek a böngésző támogatottsága [13] jelenleg elég alacsony. Persze a polling lehetősége mindig adott, de ha igazán reaktív és push alapú megoldást akarunk keresni akkor valami más protokoll után kell néznünk ezzel leváltva, vagy akár kiegészítve a HTTP-t.

2.2.1 Websocket

Websocket [14] egy kommunikációs protokoll, ami kétirányú kommunikációt tesz lehetővé a böngésző és egy távoli szerver között egy perzisztens kommunikációs csatornát használva. A kapcsolat kiépítésére kezdetben a böngésző egy speciális HTTP

kérést küld a szervernek, ha a szerver szintén támogatja a Websocket protokollt, akkor ennek megfelelően válaszol erre a kérésre és kiépítésre kerül ez a kommunikációs csatorna, ami egy egyszerű TCP kapcsolatot használ.



7. ábra Websocket kommunikáció [15]

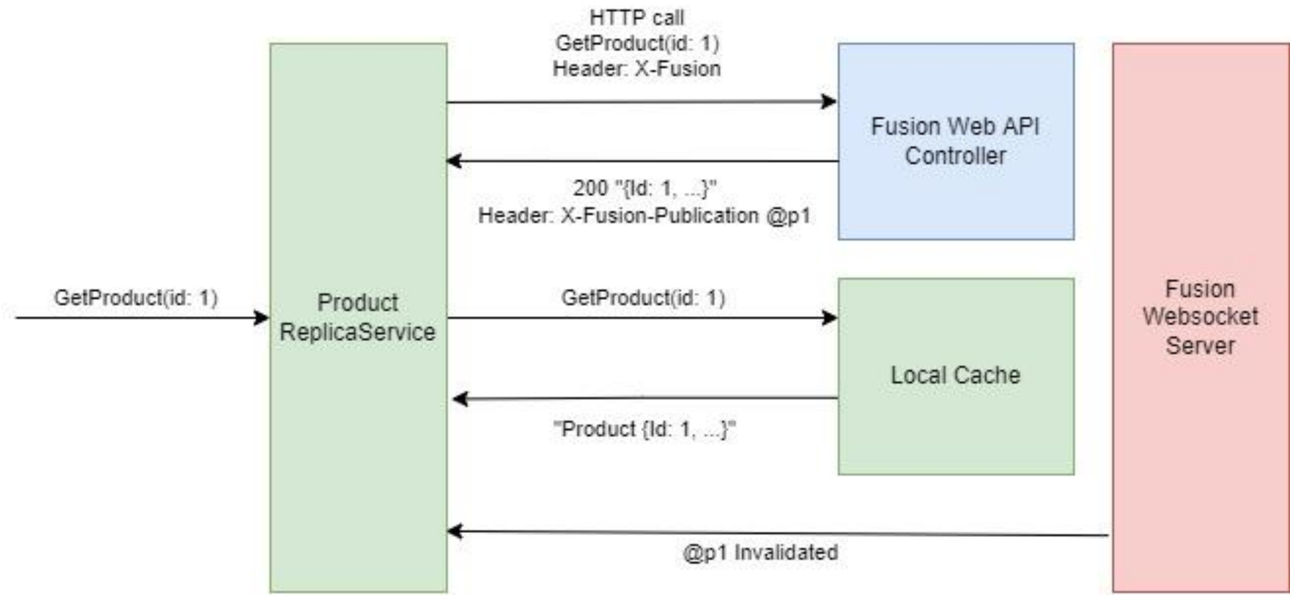
Innentől kezdve a böngésző és a szerver szabadon küldhetnek üzeneteket egymásnak. Ennek a protokollnak a segítségével a szerver már képes lesz értesíteni a klienst arról, hogy az általa lekérdezett adatok megváltoztak, ezzel jobban megfelelve a push alapú reaktív szemléletnek. Websocket esetén az átküldött üzeneteket „frame”-eknek nevezzük, az ezekben átküldött adat szöveges, illetve bináris formátumú lehet. Az adat frame üzeneteken kívül kezelnünk kell még két speciális üzenetet a „ping”-et ami a kapcsolat fenntartására szolgál, illetve a kapcsolat lezárását kezdeményező „close” üzenetet, amelyet a szerver és a kliens is küldhet.

2.2.2 Fusion – Replica Service

A Fusion osztálykönyvtár abban is segítséget nyújt, hogy a kliens alkalmazás értesülni tudjon az általa lekérdezett adatok módosításáról. Erre a szolgálnak a **Replica Service** [16] elnevezésű osztályok. Az elnevezés onnan ered, hogy egy Replica Service olyan, mint egy Compute Service kliens oldali másolata, vele azonos interfészt valósít meg, azonban ennek a konkrét implementációja nem egy lokális függvény, hanem egy HTTP hálózati hívás lesz. A pontos működés lényege, hogy adat lekérdezésekor automatikusan megtörténik a feliratkozás is az adott adat változásaira. Ez a gyakorlatban úgy néz ki, hogy a Replica Service egy speciális HTTP header-rel jelzi a szerver számára,

hogy a későbbiekben értesülni is akar majd a lekérdezett adat megváltozásáról. A lekérdezett adatok kliens oldalon cachelésre kerülnek ugyanúgy, mint egy szerver oldali Compute Service esetén, illetve Fusion visszaküld egy azonosítót a lekérdezett adatok mellé, amivel később hivatkozni fog rájuk az invalidáció során és felépít egy Websocket kapcsolatot, ha az korábban még nem került felépítésre.

Amikor valamely adat szerver oldalon invalidálásra került akkor a Fusion Webscoket Server értesítést küld az összes kliensnek erről, aki ezt az adatot lekérdezte és van még vele aktív websocket kapcsolatunk. Az értesítésben elküldjük a korábbi azonosítót, hogy a kliens tudja, hogy melyik általa lekérdezett adat változott meg. Az új adat azonban magában az értesítésben nincs benne. Ezzel gyakorlatilag itt is a push-pull modellt követjük, ennek itt az az előnye, hogy a kliens döntheti el, hogy számára fontos-e, hogy az adott adat legfrissebb verzióját lássa, ezzel megelőzve az esetlegesen fölösleges hálózati kommunikációt. Illetve a kliens akár megadhatja a lehetőséget a felhasználónak, hogy ő döntsön arról, hogy mikor kérje le az új értéket. Gondoljunk bele például abba az esetbe amikor egy közösségi oldalon járunk és megtekintünk valamilyen bejegyzést, ilyenkor általában a bejegyzés alatt számszerűen láthatjuk a rá beérkezett reakciók számát. Ez egy népszerű poszt esetén igen gyakran változhat, azonban azzal, hogy ezt az értéket valós időben frissítjük sok plusz hálózati kommunikációt iktatunk be, azonban nem biztos, hogy jobbá tesszük a felhasználói élményt.



8. ábra Invalidációs üzenet Websocket segítségével

2.2.3 Egyéb alternatívák

Server Sent Events (SSE) [17]

Az egyik legelső technológia, ami lehetővé tette push alapú kommunikációt a szervertől a böngésző felé. Hátránya azonban, hogy erősen korlátozott az egyidejű aktív kapcsolatok száma, illetve az is, hogy a kommunikáció csak egy irányban, szervertől a kliens felé tud működni. Azonban jó választás lehet, ha régi böngészőket kell támogatnunk, ahol Websocket nem elérhető.

gRPC Stream [18]

Ez egy bináris kommunikációs protokoll, ami HTTP/2-t használ a háttérben. A gRPC lehetővé teszi az ún. stream alapú kommunikációt, amivel akár két irányú push alapú kommunikáció lehetséges kliens és szerver között. Ez jó megoldásnak tűnhet, azonban ahogy korábban is említettem a böngészők jelenleg nem nyújtanak olyan részletességű API-t a HTTP/2 kérések küldéséhez, amivel megvalósítható lenne egy gRPC kliens böngészőben. Azonban, ha a későbbiekben ez a támogatottsági kérdés megoldódik, vagy esetleg két szerver oldali alkalmazás közötti kommunikációra akarjuk használni akkor ez is jó megoldás lehet.

SignalR [19]

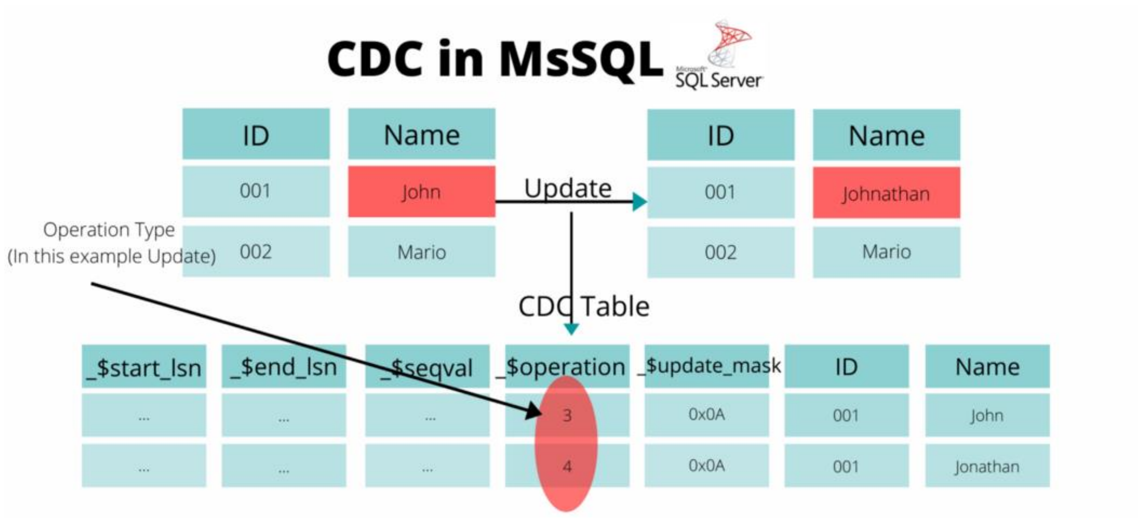
Nyílt forráskódú .NET keretrendszer szerver és kliens közti push alapú kommunikáció megvalósítására. A háttérben Websocket-et használ, ha az támogatott, ha nem akkor más alternatív technológiákra (mint például SSE) vált át. Jól használható, a .NET-es világban szinte standard megoldás valós idejű funkciók implementálására. Jelen esetben főként azért választottam a Fusion keretrendszert helyette, mert ott azzal, hogy az adat lekérdezés és a változás értesítés egy lépésben történik sokkal kevesebb kóddal elérhető ugyanaz a működés. SignalR esetén minden lehetséges adatváltozásra eseményeket kellett volna definiálnom, megírni a hozzájuk tartozó kliens oldali feldolgozó logikát, illetve kezelnem az ezekre való fel és le iratkozást. Viszont az is elmondható, hogy a Fusion egy elég új könyvtár melyet pár fejlesztő tart fent, a hosszú távú támogatottsága még kérdéses, ha egy hosszútávú projektbe vágunk bele akkor a SignalR ilyen szempontból egy biztosabb választás lehet.

2.3 Backend-adatbázis kommunikáció reaktív módon

A 2.1-es fejezetben már láthattuk, hogy invalidáció által hogyan tudjuk az adatbázisból származó adatokat újra lekérdezni és naprakészen tartani a Fusion cache tartalmát. Azonban ez a megoldás csak akkor működik, ha a mi alkalmazásunk az egyetlen, aki módosítja ezt az adatbázist. Gyakorlatban azonban könnyen előfordulhat, hogy több backend alkalmazás közös adatbázist használ, ezért keresnünk kell valami olyan megoldást, amivel más alkalmazás által végzett módosításokról is értesülni tudunk. A probléma itt is hasonló, mint a HTTP esetén, ugyanis a relációs adatbázisok nem tesznek elérhetővé olyan API-t, ahol adatok változásáról tudnánk értesülni push jelleggel. Ahhoz, hogy értesülni tudjak az adatbázisban tárolt adatok változásáról, illetve, hogy az itt tárolt adatokat is egy eseményfolyamként tudjam kezelni több komponens együttes használatára volt szükség, nincs olyan konkrét technológia, ami ezt egy lépésben lehetővé tenné számunkra, mint például az előző esetben a Websocket.

2.3.1 Change Data Capture

A Change data capture [20] (CDC) lényege, hogy az adatbázisban történt módosításokat feljegyezzük külön erre a célra készített ún. Change table (CT) táblákban. A CDC technológiát szinte az összes népszerű relációs adatbázis támogatja, azonban az engedélyezés menete, illetve a CT táblák felépítése adatbázisonként eltérő lehet, a továbbiakban én az SQL Server specifikus működést ismertetem. CDC-t először engedélyezünk kell adatbázis szinten, majd meg kell adnunk, hogy mely táblák változásait akarjuk ilyen módon eltárolni. Ez SQL Server esetén a *`sys.sp_cdc_enable_db`* és a *`sys.sp_cdc_enable_table`* tárolt eljárások segítségével lehetséges. A változásokat magából a **tranzakciós log**-ból tudjuk kinyerni, ezt konkrétan az **SQL Server Agent** folyamat végzi el.



9. ábra Adatváltozások rögzítése [43]

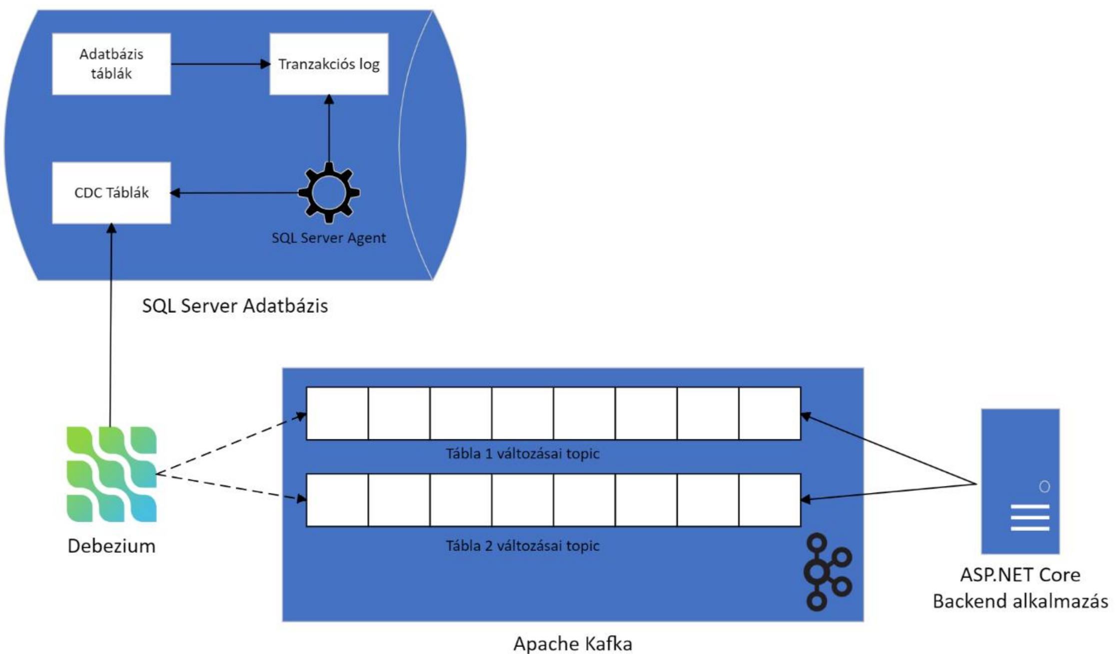
A fenti ábrán látható a CT táblák vázlatos felépítése, illetve hogy egy példa Update művelet esetén ezek hogy kerülnek kitöltésre. A táblák felépítése láthatóan hasonló azokhoz a táblákhoz melyek módosításait tárolják, kiegészítve egy-két plusz információval. Ezek közül talán a legfontosabb az `_$operation` ami a művelet típusát tárolja (Insert/Update/Delete), illetve a `_$start_lsn` ami a módosító művelethez kapcsolódó tranzakció azonosítója a tranzakciós logban.

CDC használatával eljutottunk oda, hogy a képesek vagyunk az adatbázisban történt változásokat elérni azzal, hogy a változásokat tartalmazó CT táblákat felolvassuk, azonban ezzel még három főbb probléma van.

- A CT táblák szerkezete adatbázisonként eltérő lehet, ha erre építünk akkor az alkalmazás logikánkat erősen hozzákötjük egy konkrét DB szoftverhez.
- A változásokat feldolgozni és átadni a backend számára kezelhető formátumban elég komplex lehet, tárolnunk kell mely változásokat dolgoztunk már fel és kezelnünk kell, ha a backend alkalmazás egy adott változást nem volt képes feldolgozni és ilyenkor újra próbálkozni. Ha itt hibát ejtünk az inkonzisztens adatokhoz vezethet az egész alkalmazásunkban.
- Ezeket a táblákat ugyan fel tudjuk mi olvasni bizonyos időközönként, de ez még mindig polling alapú megközelítés, ami szembe megy a dolgozatban felvázolt push alapú reaktív szemlélettel.

2.3.2 Debezium és Kafka

Az előbb felvázolt problémákra képes megoldást adni a **Debezium** [21] szoftver azzal, hogy képes a CDC táblákban lévő változásokat eseményekre leképezni, ezzel támogatva a reaktív szemléletet mi szerint az egyes adatforrásokat eseményfolyamként kezeljük. A Debezium támogatja az összes népszerűbb relációs adatbázist [22] és az általuk CDC táblákban rögzített változásokat képes azonos formátumú eseményekre leképezni, ezzel függetlenné téve az architektúránkat a konkrét adatbázis szoftver típusától. Ezeket az eseményeket pedig képes továbbítani egy **Kafka** [23] eseménybuszra, amit a backend alkalmazásunk már könnyedén fel tud dolgozni.



10. ábra CDC, Debezium és Kafka együttes működése

Apache Kafka egy nyílt forráskódú, elosztott *publish-subscribe* jellegű üzenetbusz. Kafka segítségével könnyedén továbbíthatunk eseményeket különböző alkalmazások között és ezeket aszinkron módon, push jelleggel tudjuk feldolgozni a fogadó oldalon. Többféle **topic**-ot hozhatunk létre, amelyek gyakorlatilag egyfajta esemény kategóriaként szolgálnak és a kliensek csak a nekik fontos kategóriákra iratkoznak fel. Debezium alapértelmezett esetben minden tábla változásaira egy külön Kafka topic-ot hoz létre. Kafka ezen felül jól tudja kezelni nagy mennyiségű esemény átvitelét (akár milliós nagyságrendben), illetve biztosítani tudja, hogy az események átvitele közben ne vesszenek el adatok.

Szinte minden népszerűbb programozási nyelven léteznek [24] Kafka kliens könyvtárak, amelyekkel fel tudunk iratkozni az üzenetbuszra és feldolgozni az érkező adatokat. Fontos megemlíteni, hogy bár ezzel a megoldással a mi alkalmazásunk tényleg push jelleggel tudja feldolgozni az adatbázisban történt változásokat, azonban a háttérben az SQL Server CDC, illetve a Debezium is használ polling jellegű adat lekérdezéseket. Ezek viszont nagyon elterjedt és megbízható megoldások, amikre fel lehet építeni egy éles alkalmazást, illetve az is elmondható, hogy relációs adatbázisok használata esetén sajnos ennél jelentősen jobb nem ismert. Dolgozatomban a tradicionális N-rétegű architektúrából indultam ki, ami relációs adatbázisokat használ, a NoSQL világban azonban találhatunk ennél jobb megoldást, melyet a következő fejezetben röviden ismertetek.

2.3.3 Egyéb alternatívák

MongoDB Change stream

MongoDB az egyik legnépszerűbb adatbázis a NoSQL világban. A change stream [25] funkcióval feliratkozhatunk egy adott kollekció (tábla) vagy akár egy egész adatbázis változásaira. A háttérben a change stream-ek ugyanazt a megoldást használják, amit a MongoDB a különböző adatbázis példányok (node) közötti replikációra használ. Ennek az értesítésnek a használatával nagy mértékben egyszerűsödni tudna a dolgozatban felvázolt architektúra. Viszont a relációs adatbázisokról MongoDB-re, vagy valamilyen más NoSQL megoldásra áttérni nagyon nagy mértékű átalakítás, ami sok egyéb kihívást rejt magában. Ha viszont az alkalmazásunk tervezésekor eleve egy ilyen adattárolási megoldás mellett döntöttünk akkor érdemes lehet a change stream technológia használatát megfontolnunk.

DML Trigger

A trigger az SQL világban egy speciális tárolt eljárás, ami egy adott táblán végrehajtott INSERT, UPDATE vagy DELETE művelet után fut le. Ezen az eljáráson belül elérjük az tábla módosítás előtti és utáni verzióját is, így megkaphatjuk, hogy pontosan milyen változások történtek, Az előbbiekben felvázolt működés megvalósítható lett volna triggerek segítségével is, léteznek keretrendszerek is melyek ilyen trigger alapú szinkronizációt, illetve az így kinyert adatok továbbítására a backend alkalmazás felé még valami más plusz megoldást kellene keresnünk.

PostgreSQL Subscription

PostgreSQL adatbázis 10-es verziójától kezdve elérhető a CREATE SUBSCRIPTION utasítás, amivel feliratkozhatunk az adatbázisban történt változásokra. Ez jó alternatíva lehet a felvázolt Debezium alapú megoldásra, ha nem akarunk egy teljes Kafka infrastruktúrát üzemeltetni, illetve eleve PostgreSQL adatbázist akartunk használni, azonban az implementáció itt is jóval komplexebb lenne.

3 Felhasznált technológiák

Ebben a fejezetben röviden ismertetem azokat a technológiákat, amelyek bár a reaktív megközelítéshez szorosan nem kapcsolódnak, de a konkrét implementáció során felhasználtam őket.

3.1.1 ASP.NET Core

Az ASP.NET Core [26] egy platformfüggetlen, nyílt forráskódú web alkalmazásfejlesztési keretrendszer. A reaktív és tradicionális példaalkalmazások backend részét ezzel a keretrendszerrel készítettem el. Az ASP .NET Core nagyon sok funkcióval rendelkezik, itt csak pár fogalmat ismertetnék röviden, amiket a fejlesztés során használtam.

- Controller: Az egyes végpontokat, ahova a kliens alkalmazásunk HTTP kéréseket tud majd küldeni ilyen Controller osztályok segítségével tudjuk definiálni.
- Dependency Injection: Ez egy tervezési minta [27] melyet a keretrendszer beépítetten támogat. Ami a további forráskód részletek megértése szempontjából fontos az igazából annyi, hogy emiatt tudtam konstruktor paraméterként egyszerűen átvenni a különböző függőségeket amire épp szükségem volt.
- HostedService: Egy folyamatosan futó háttérfolyamat adható meg vele, én konkrétan egy ilyen folyamatot használtam arra, hogy a Kafka eseményeket feldolgozzam.
- WebSocket: A korábban ismertetett Websocket technológiát szintén támogatja az ASP.NET, ezt konkrétan egy Middleware formájában tudjuk bekapcsolni.
- Swagger: A Controller osztályokból képes egy API leírót készíteni, amiből HTTP kliensek generálhatóak, ezzel megkönnyítve a fejlesztést.
- Autentikáció: Szintén egy Middleware formájában kapcsolható be, ezután pedig Controller végpont szinten, attribútumok formájában megadhatóak az adott végpont eléréséhez szükséges jogosultságok.

3.1.2 Hangfire

A Hangfire keretrendszer [28] sokféle job futtatását teszi lehetővé, én konkrétan egy **RecurringJob** [29] futtatására használtam, ami minden nap adott időpontban végrehajtódik.

3.1.3 Kafka .NET

A Kafka események feldolgozására és küldésére használható .NET alapú könyvtár. Nekem csak az esemény fogadó részére volt szükségem, így elsősorban a könyvtár **Consumer** [30] osztályát és a szorosan hozzákötődő osztályokat használtam.

3.1.4 EF Core

Az EF Core egy Object-relational mapping (ORM) eszköz, melynek segítségével C# kódból tudunk adatbázis sémát készíteni, ezt az EF Core szoftveren belül Code first megközelítésnek nevezzük. Az adatbázissal egy DbContext osztály segítségével tudunk kommunikálni, itt fogalmazhatóak meg utasítások az egyes táblákból kiindulva. adatbázis lekérdezéseket Language Integrated Query (LINQ) kifejezések segítségével tudunk megadni. Ezeket a lekérdezéseket az EF Core majd a háttérben az adott adatbázis szervernek megfelelő Structured Query Language (SQL) lekérdezésre fordítja. IQueryable típusú objektumok segítségével a lekérdezéseket memóriában tudjuk felépíteni és tetszőleges manipulálni, mielőtt azokat tényleges elküldenénk az adatbázis szervernek.

3.1.5 ASP .NET Core Identity

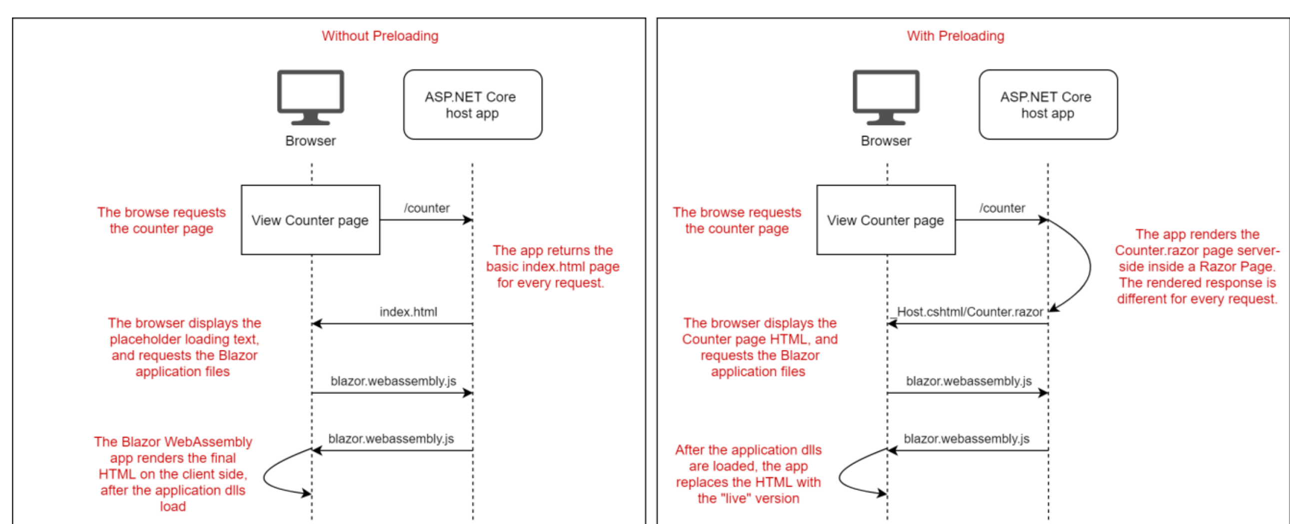
Az Identity keretrendszer megkönnyíti a felhasználókezelést azzal, hogy modelleket és függvényeket ad a gyakran használt funkciókhoz. A felhasználói adatok tárolása, mint például az email, felhasználónév, jelszó hash stb. az IdentityUser elnevezésű osztályban történik. Az felhasználókezelésért a UserManager osztály felel, például vele lehet felhasználót létrehozni egy IdentityUser objektum és a kívánt jelszó megadásával, ami természetesen majd hashelés után kerül tárolásra.

3.1.6 Blazor WASM

Blazor egy .NET alapú keretrendszer webalkalmazások UI rétegének elkészítésére. Ennek a keretrendszernek a segítéségével böngészőben futó kliens alkalmazásokat készíthetünk Javascript helyett C# nyelven, ezzel használhatjuk a

megszokott .NET-es könyvtárakat kliens oldalon is, illetve potenciálisan kódot oszthatunk meg a kliens és szerver alkalmazásaink között. Az egyes komponenseinket a **Razor** elnevezésű templating motor segítségével készíthetjük el, ami lehetővé teszi, hogy HTML fájlokba C# kódot ágyazzunk be. A keretrendszer alapvetően két megközelítést támogat, a szerver oldalon renderelt **Blazor Server** és a kliens oldalon renderelt **Blazor Webassembly** működési módokat, ebből én az utóbbi megoldást választottam. A C# kód, amit ilyenkor készítünk Webassembly formátumba kerül átalakításra, amit már támogatnak a böngészők, így készíthetünk SPA alkalmazásokat a .NET világban.

Webassembly mód használata esetén adott a lehetőség, hogy a kliens oldali alkalmazásunkat statikus fájlok (.html, .js, .css) formájában szolgáljuk ki, ennek előnye, hogy maga a hostolás nagyon olcsó, vagy sok helyen akár ingyenes is. Ezzel viszont az a probléma, hogy az oldal első betöltése eléggé lassú lehet, mert először meg kell várnunk amíg az összes fájlt letöltődik és az oldal csak utána kerül renderelésre. A HTML oldal, amit visszakapunk (index.html) gyakorlatilag csak egy placeholder, amit a javascript fájlok fognak tartalommal feltölteni amikor azok letöltődnek, amíg ez meg nem történik addig viszont a felhasználó gyakorlatilag egy üres oldalt lát. Erre a problémára megoldás a **szerver oldali előrenderelés**[2], melynek lényege, hogy az első oldalbetöltéskor szerver oldalon kirendereljük az oldal tartalmát.



11. ábra Szerver oldali előrenderelés [45]

[2] Ennek a mechanizmusnak az ismerete azért fontos mert a reaktív alkalmazás autentikációs megoldása támaszkodik rá, a _Host.cshtml-ben helyeztem el ezzel kapcsolatos logikát, amit erről fontos tudni az az, hogy ez minden oldalbetöltés előtt le fog futni, még szerver oldalon.

Ez Blazor esetén akkor elérhető, ha a kliens alkalmazásunkat egy ASP.NET backend alkalmazáson belül hostoljuk, ez praktikusan lehet ugyanaz az alkalmazás, ahol a REST API-nk elérhető. A tényleges előrenderelés egy konvenció szerint **_Host.cshtml** elnevezésű oldal segítségével történik, ahol a kliens alkalmazás komponenseihez hasonlóan szintén a Razor template motor megoldásait tudjuk használni, csak itt ez szerver oldalon fog lefutni.

3.1.7 RestEase

RestEase egy C# nyelven készült HTTP kliens könyvtár, alapvetően azért van rá szükség mert a Fusion Replica Service megoldás jelenleg csak ezt a könyvtárat támogatja. A használata alapvetően kényelmes, az egyes kliens osztályok által megfogalmazandó kéréseket egy interfész formájában tudjuk leírni, amiből a könyvtár majd konkrét implementációt fog generálni.

3.1.8 Docker, docker-compose

Docker segítségével az alkalmazásunkat egy izolált környezetben tudjuk futtatni, ezt nevezzük **konténernek**. Ezek a konténerek egy virtuális géphez hasonló módon működnek, azonban több konténer futtatása esetén a konténerek osztoznak a host gép kernelén, így összességében kisebb az erőforrás igényük. Egy konténer előállításához szükségünk van egy docker rendszerképre (image). A rendszerkép egy read-only template, ami tartalmazza a szükséges adatokat egy konténer létrehozásához. A legtöbb szoftver komponenshez elérhetőek hivatalos rendszerképek, amelyeket felhasználhatunk. Én konkrétan a Kafka infrastruktúra elemeit (Zookeeper, Connect, Kafka) és az adatbázist (SQL Server) futtattam ezek segítségével.

Az ilyen több konténerből álló alkalmazások készítését és futtatását teszi egyszerűbbé a **Docker Compose**. Ennek a technológiának a használatával egy YAML konfigurációs fájlban felsorolhatóak az egyes konténerek egy-egy kész rendszerkép (vagy Dockerfile) használatával. Alapértelmezetten ezek a konténerek egy hálózatba kerülnek, így tudnak kommunikálni egymással, sőt a konfigurációs fájlban megadható hostname használatával tudnak hivatkozni egymásra a hálózaton, a beépített Domain Name System (DNS) szolgáltatás segítségével.

4 Esettanulmány

Példa alkalmazásnak egy webshop implementálását választottam. Ennek igazából az architektúra szempontjából nincs nagy szerepe, azért esett erre a választásom mert ilyen alkalmazást már valószínűleg mindenki használt, így az egyes funkciókat nem kell túl nagy részletességgel bemutatnom. Konkrétan az alábbi funkciók megvalósítását tűztem ki célnak.

- Keresés, szűrés termékek között:
  - Szűrés: adott kategóriájú termékek, akciós termékek, készleten lévő termékek, adott értékelésnél jobb termékek, adott ár tartományon belüli termékek, név alapján szűrés
  - Rendezés: értékelés szerint, ár szerint, hozzáadás dátuma szerint
- Értékelések írása termékekhez: Szöveges leírás és számszerű értékelés 1-5 között. Ezen értékelések száma és átlaga megjelenik a termékeknél
- Kosárkezelés, egyszerű foglalási rendszerrel. Egy raktárkészleten lévő termék több felhasználó kosarában is benne lehet, de csak az tudja megrendelni, aki először kosárba helyezte, neki van **lefoglalva**. Ha valaki számára lefoglalt terméket eltávolít a kosarából, akkor ez a foglalási jog átszáll a kosárba helyezés időpontja szerint a következő felhasználóra, ha van ilyen.
- Rendelés elküldése, korábbi rendelések megtekintése
- Integráció egy külső alkalmazással, ami a termékek adatainak szerkesztését, és a raktárkészlet kezelést teszi lehetővé és az ezzel kapcsolatos módosításokat adatbázis szinten végzi el.

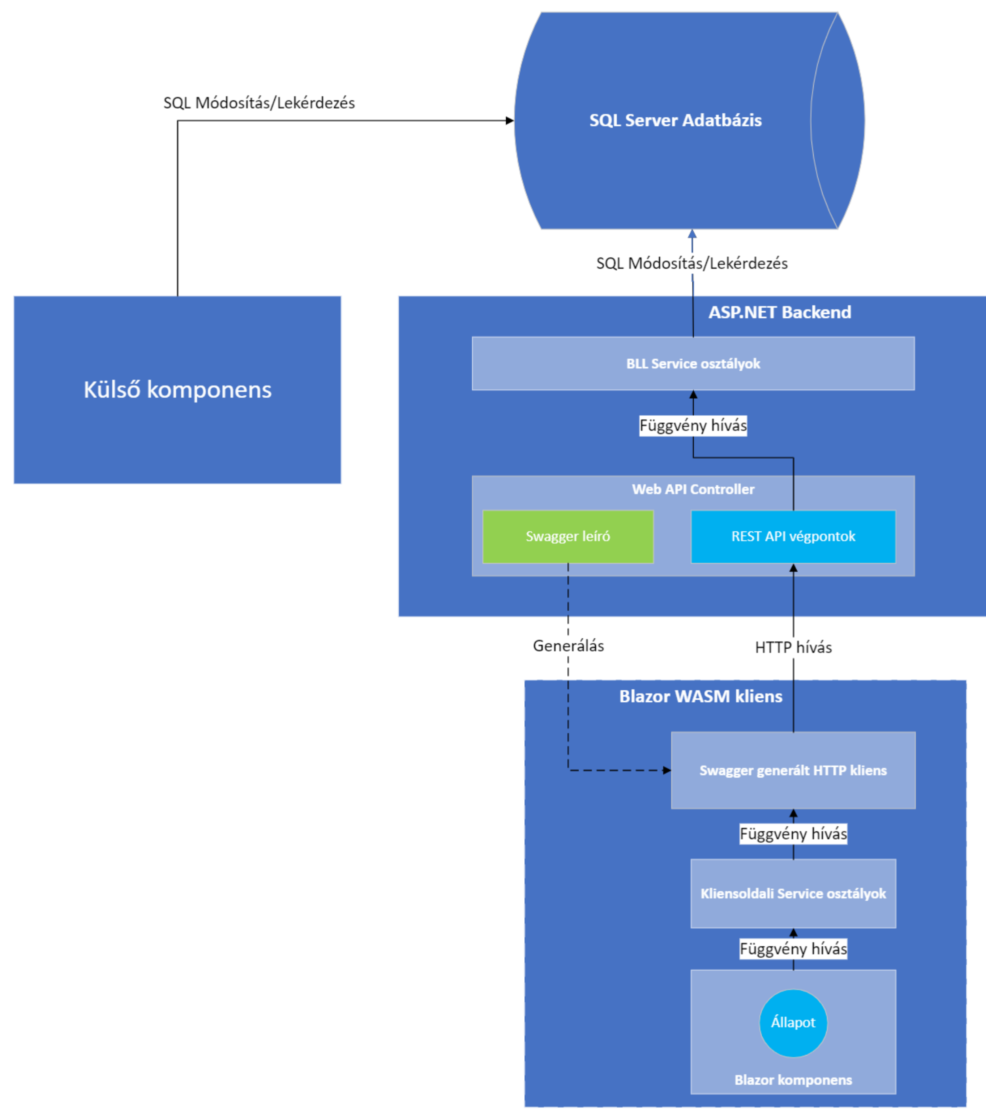
Ezeket a funkciókat két külön alkalmazásban is megvalósítottam ezekre a későbbiekben reaktív és tradicionális implementációként fogok hivatkozni. A következő fejezetekben ezen alkalmazások felépítését mutatom be, majd egy pár konkrét üzleti funkció megvalósítását ismertetem. Igyekeztem olyan funkciókat választani a szemléltetéseknél, ahol a két alkalmazásban a követett elvek, illetve a felhasznált komponensek miatt eltérő megközelítést kellett követnem.

4.1 Architektúra

Ebben a fejezetben röviden ismertetem a két alkalmazás magas szintű architekturális felépítését. A tradicionális alkalmazás az N a rétegű architektúrát követte, ennek megfelelően konkrétan a következő komponensekből állt:

- **ASP.NET Backend**
- **Blazor WASM Frontend**
- **SQL Server adatbázis**

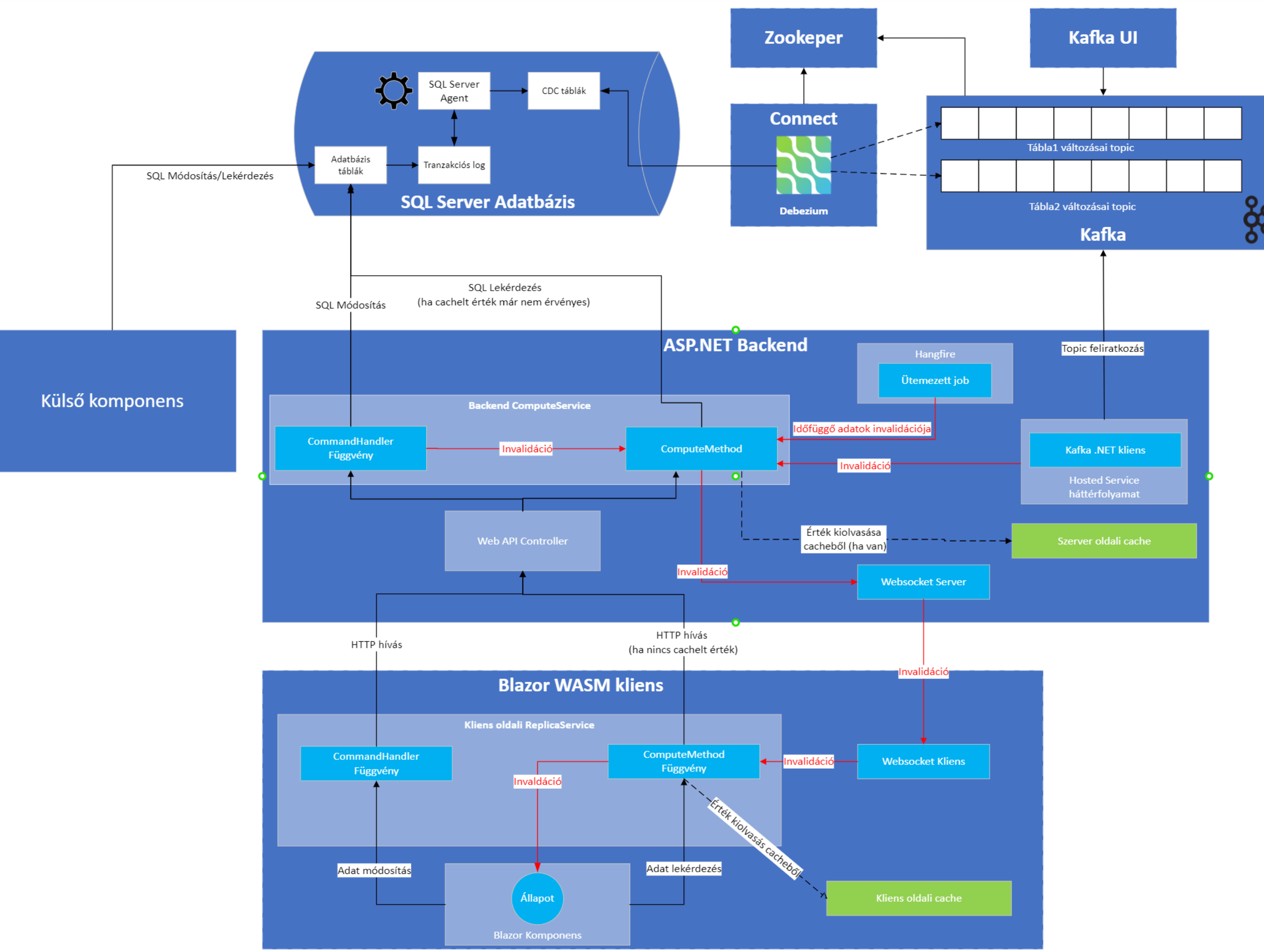
Az egyedüli olyan elem, ami talán nem olyan gyakori vagy szabványos az a Swagger leíróból generált kliens. Azonban ezt csak a fejlesztés megkönnyítése végett használtam fel, az architektúra szempontjából nincs nagy szerepe.



12. ábra Tradicionális implementáció, architektúra nézet

Fontos megjegyezni, hogy ezt még kiegészíthettem volna különböző reaktív működést segítő elemekkel, mint például a SignalR, ami egy gyakran használt könyvtár a .NET világban. Azonban klasszikus adatkezelő alkalmazásokban a SignalR-t ritkán alkalmazzák, inkább csak olyankor, amikor a push jellegű kommunikáció a funkciók alapján alapvető elvárás, például egy chat alkalmazás esetén.

Ezzel szemben a reaktív alkalmazás felépítése a következő ábrát követi:



14. ábra Reaktív implementáció, architektúra nézet

Láthatóan ez az architektúra tartalmazza ugyanazokat az építőelemeket, mint amelyek a tradicionális alkalmazásban is megtalálhatóak. Azonban itt ezen felül még szükségünk van egy Apache Kafka infrastruktúra kiépítésére, ez konkrétan a következő négy komponens használatát jelenti.

- **Zookeper:** Apache infrastruktúra elemek közti kommunikációt és szinkronizálást biztosítja és különböző konfigurációs információkat tárol és tesz elérhetővé számukra.
- **Kafka:** Maga az üzenetbusz, ahova az események kerülnek, amire a backend alkalmazás feliratkozik
- **Connect:** Lehetővé teszi azt, hogy **connector** komponenseket használjunk, amelyek képesek más rendszerekből adatokat kinyerni és azt a Kafka eseménybuszra továbbitani. Itt fut ténylegesen a Debezium, ami szintén egy ilyen connector.
- **Kafka UI:** Vizuális felület az események és topic-ok kezelésére, leginkább a működés ellenőrzésére, illetve az egyes problémák felderítésére hasznos.

Ezen felül itt az egyes komponenseken belül bejelöltem a korábban ismertetett Fusion komponenseket és azok jellemző interakcióit, külön kitérve a lehetséges invalidációs útvonalakra.

4.2 Projektek és futtatási környezet bemutatása

Két alap komponens (backend, frontend) futtatási módjában igazából semmi szokatlan nincs, a backend alkalmazás a REST API mellett szintén hosztolja a frontend alkalmazás fájljait, ahogy azt korábban a 3.1.6 fejezetben említettem. Az Apache infrastruktúrát Docker konténerek használatával állítottam fel. Ezeket a konténereket egy docker-compose.yml fájlban soroltam fel és konfiguráltam őket, hogy tudjanak kommunikálni egymással. Ez a fájl a dolgozat függelék részében (6.1) látható. Az egyes rendszerkép fájlokat a Debezium docker registry tárhelyéről hivatkoztam be, az itt található Connect komponensre már eleve telepítve van a Debezium connector. Ezen felül az adatbázist is docker konténerben futtatam, hogy egyszerűbben megoldható legyen a kommunikáció a Connect komponenssel.

Ezután még szükség volt arra, hogy jelezzem a Debezium számára, hogy milyen típusú adatbázishoz kell kapcsolódnia, illetve, hogy ezt hol találja meg. Az ilyen konfigurációs beállításokhoz használható a Connect komponens REST API-ja. A backend alkalmazás automatikusan minden induláskor ide küld egy kérést a következőhöz hasonló JSON üzenettel.

```json
{
  "name": "adventureworks-connector",
  "config": {
    "connector.class": "io.debezium.connector.sqlserver.SqlServerConnector",
    "tasks.max": "1",
    "database.server.name": "adventureworks",
    "database.hostname": "sqlserver",
    "database.port": "1433",
    "database.user": "sa",
    "database.password": "Password123!",
    "database.dbname": "AdventureWorks2019",
    "database.history.kafka.bootstrap.servers": "kafka:29092",
    "database.history.kafka.topic": "schema-changes.adventureworks"
  }
}
```

4.3 Backend funkciók implementálása

Ebben a fejezetben különböző üzleti funkciók backend oldali megvalósítását mutatom be, elsősorban arra koncentrálva, hogy milyen esetekben kellett más szemléletet követnem a reaktív alkalmazás implementációjakor és hogy ez milyen többlet kóddal vagy esetleg komplexitással járt.

4.3.1 Termék alapdadatok lekérdezése és szerkesztése

Elsőnek egy nagyon egyszerű példán, egy adott termék alapadatainak lekérdezésén és szerkesztésén keresztül szemléletem a különböző koncepciók használatát. Az itt bemutatott forráskód (4.3.2-ig bezárólag) bár az én alkalmazásomból származik, de nagy mértékben hasonlít a Fusion könyvtár egyik oktatóanyagához [12]. Azonban ennek ismerete szükséges lehet a későbbi komplexebb funkciók megértéséhez, ezért itt is bemutatom ugyanazokat az elveket, amelyekre ez az oktatóanyag is kitér.

```csharp
public class ProductService :
 DbServiceBase<FusionDbContext>, IProductService
  {
    private readonly IMapper _mapper;
    private readonly IDbEntityResolver<int, Product> _productResolver;

    [ComputeMethod]
    public virtual async Task<ProductDto> GetProduct(
      int productId, CancellationToken cancellationToken)
    {
      var product = await _productResolver.Get(productId, cancellationToken);
      if (product is Product)
      {
         return _mapper.Map<ProductDto>(product);
      }
    }
```

```
// ComputeService regisztrálása
var fusion = services.AddFusion();
fusion.AddComputeService<IProductService, ProductService>();
```

Egy ilyen egyszerű *ComputeService* esetén mindössze annyi a dolgunk, hogy le származunk a *DbServiceBase* osztályból és a lekérdező függvényeket virtuálissá tesszük és ellátjuk a *ComputeMethod* attribútummal. Ezután regisztrálnunk kell az osztályunkat a fent látható módon, ezzel elérhetővé tesszük azt a Dependency Injection számára, illetve bekapcsoljuk a proxy osztályok generálását (lásd **Error! Reference source not found.**). Egy apró különbség, hogy az adatbázis lekérdezést egy *EntityResolver* objektummal kell végeznünk. Ennek akkor lesz igazából jelentősége amikor egyszerre több elemet kérdezünk le adatbázisból, erre még a több termék egyszerre való lekérdezésénél visszatérek.

```
[CommandHandler]
public virtual async Task Edit(UpdateProductCommand command)
{
  if (Computed.IsInvalidating())
  {
    _ = GetProduct(command.ProductId, cancellationToken);
    return;
  }
  using var dbContext = await CreateCommandDbContext();
  var product = await dbContext.Products.AsQueryable()
        .SingleOrDefaultAsync(p => p.ProductId == command.ProductId);
  if (product is Product)
  {
    // product entitás módosítása a command értékeivel
    _mapper.Map(command, product);
    await dbContext.SaveChangesAsync(cancellationToken);
  }
```

A módosító műveleteket a *CommandHandler* attribútummal láttam el, illetve hozzáadtam egy invalidációs blokkot, jelen esetben ez a *Computed.IsInvalidating()* feltételt tartalmazó elágazás. A működés alapötlete, hogy a módosító függvény kétszer fog lefutni a háttérben, az első futás során *Computed.IsInvalidating()* értéke hamis lesz, így az invalidációs blokk kimarad és a „rendes" műveletek futnak le, jelen esetben egy termék adatainak módosítása adatbázisban. A második lefutás annyiban lesz más, hogy egyrészt ez a feltétel igaz lesz így bejutunk az invalidációs blokkba, illetve bármilyen *ComputeMethod* függvény értéke, amelyet ezen lefutás során meghívunk **invalidálásra kerül**. Jelen esetben ez láthatóan a *GetProduct* függvény lesz.

4.3.2 Több termék alapadatainak listázása

Több termék listázásához fel tudjuk használni az előbbi *GetProduct* függvényünket, ami egy termék adatait adja vissza Id alapján.

```
[ComputeMethod]
public virtual async Task<List<ProductDto>> GetManyProducts(
  List<int> productIdList)
{
  await using var dbContext = CreateDbContext();
  var products = await Task.WhenAll(
    productIdList.Select(async productId =>
    {
      return await GetAllProducts (productId);
    })
  );
  return products.ToList();
}
```

Itt az egyes Id alapú lekérdezéseket a *Task.WhenAll* utasítás segítségével **párhuzamosítva** futtattam, ahogy arra a Fusion dokumentáció is javaslatot tesz [31]. Erre az a motiváció, hogy „reménykedünk" benne, hogy a legtöbb termék már benne lesz a cache-ben, így a *GetProduct* függvényhívás nem fog ténylegesen adatbázishoz nyúlni, cache memóriából felolvasni pedig könnyedén tudunk párhuzamosítva. Előfordulhat viszont olyan eset, hogy a cache üres, vagy hogy sok elem nem található meg benne amit, épp el szeretnénk érni. Itt jön képbe az *EntityResolver* objektum, amit a *GetProduct* függvény használ adatbázis elérésre, ugyanis ennek az osztálynak a *Get* függvénye a háttérben csoportosítja az azonos entitásokhoz tartozó Id alapú lekérdezéseket és egy adatbázis kérésben hajtja őket végre. Így nem fogunk feleslegesen sok külön kérést küldeni az adatbázis felé, viszont mivel technikailag meghívtuk ezeket az Id alapú *ComputeMethod* lekérdezéseket maguk a függőségek kiépültek a *GetAllProducts* és a *GetAllProducts* függvények között. Amire még figyelnünk kell ilyen listázó függvények esetén, hogy nem csak az egyes elemek (jelen esetben termék) módosításakor, hanem törlés vagy hozzáadás esetén is invalidálnunk kell.

```
[CommandHandler]

public virtual async Task Add(AddProductCommand command,
CancellationToken cancellationToken)
{
  if (Computed.IsInvalidating())
  {
    _ = GetAllProducts();
```

```
    return;
  }
  //...
```

4.3.3 Komplex lekérdezések megvalósítása - termékek közti keresés

Legtöbb webshop alkalmazásban központi funkció a termékek közti keresés megvalósítása. Fontos különbség az előző listázó funkcióhoz képest, hogy itt a termék keresési eredmények listában nem csak a termékek alapadatai látszanak, hanem a hozzájuk tartozó értékelés, raktárkészlet és akció entitásokból aggregált adatok is. Ezeket az adatokat a tradicionális alkalmazás esetén egy C# LINQ kifejezés segítségével állítottam elő, amiből a háttérben egyetlen, sok JOIN utasításból álló SQL lekérdezés keletkezik majd. Ez kód szinten a következőhöz hasonlóan nézett ki.

```
var products = await dbContext.Products.AsQueryable()
  .Select(p => new ListProductDto()
  {
    // Értékelés entitásokból származó adatok
    AverageRating = p.ProductReviews.Average(pr => pr.Rating),
    NumberOfRatings = p.ProductReviews.Count(),

    // Raktárkészlet entitásokból származó adatok
    CurrentStock = p.ProductInventories.Sum(pi => pi.Quantity),

    // Termék kép entitásokból származó adatok
    ThumbnailImage = p.ProductProductPhotos
      .First(ppp => ppp.Primary).ProductPhoto.ThumbNailPhoto,

    // Kosár termékekből entitásokból származó adatok
    IsInShoppingCart = p.ShoppingCartItems.Any(sci =>
      userCartItemIds.Contains(sci.ShoppingCartItemId)),

    ReservedQuantityInShoppingCarts = p.ShoppingCartItems
      .Sum(sci => sci.ReservedQuantity),

    // Akció entitásokból számazó adatok
    BestAvailableSpecialOffer = p.SpecialOfferProducts.Where(p => //... )
  })
```

Elméletben a reaktív alkalmazásban is készíthettem volna egy ugyanilyen függvényt, és elláthattam volna a ComputeMethod attribútummal, azonban ez több szempontból is problémás lett volna.

- Először is ez a lekérdezés a termékeken felül még 5 más entitás adatait használja fel, ezek bármelyikének módosítása esetén invalidálnunk kellene ezt a függvényt. Mivel többféle módosító művelet is tartozik az egyes entitásokhoz, ez jelen esetben nagyságrendileg 15-20 esetet jelent, amikor invalidálnunk kell. Könnyen belátható tehát hogy ez a megközelítés sok plusz kódot eredményezne, illetve sok potenciális hibaforrást adna az alkalmazáshoz, hiszen egy ilyen invalidációról könnyen megfeledkezhetünk.
- Másrészt pedig, ha ezt az invalidációt sikeresen el is végezzük az összes helyen, akkor sem lenne arra módunk, hogy csak a ténylegesen megváltozott részt számoljuk újra, így minden invalidáció esetén az egész adatbázis lekérdezést újra végre kellene hajtanunk.

Ezért ehelyett inkább a 2.1 fejezetben ismertetett elveket követve több kisebb függvényre bontottam ennek a funkciónak a megvalósítását. A végleges megoldás vázlatosan a következő lépésekből áll.

1. Először kilistázom azokat a termék azonosítókat melyek megfelelnek a keresési feltételeknek, ezt végzi a `GetProductIdListForFilter` függvény. Ez egy sima adatbázis lekérdezés, ebben még semmi szokatlan nincs.
2. Ezután kilistázom a termékeket a korábban bemutatott `GetManyProducts` függvényhez hasonló módon, ami egy `EntityResolver`-t használ. Fontos különbség, hogy itt szükség lesz a termék entitáshoz kapcsolódó adatokra is, ezért az `EntityResovler`-t úgy konfiguráltam, hogy a termékhez kapcsolódó entitásokat is lekérdezze, ezt egy-egy `Include` hívással tettem meg, amiből DB szinten majd JOIN utasítások lesznek.

```
services.AddDbContextServices<FusionDbContext>(dbContext =>
{
  dbContext.AddEntityResolver<int, Product>(_ => new()
 {
   QueryTransformer = products => products
     .Include(p => p.ProductSubcategory)
     .Include(p => p.ProductReviews)
     .Include(p => p.SpecialOfferProducts)
     .Include(p => p.ShoppingCartItems)
     .Include(p => p.ProductModel)
     .Include(p => p.ProductProductPhotos)
       .ThenInclude(pp => pp.ProductPhoto)
 });
```

Ezzel a lekérdezéssel igazából már minden szükséges adat rendelkezésünkre fog állni, viszont fontos, hogy a függőségi gráfot is kiépítsük, ezért az itt lekérdezett kapcsolódó entitásoknak csak az ID mezőit fogom felhasználni, amivel további függvény hívásokat végzek.

3. Tehát ezután ezekkel az ID listákkal meghívtam a kapcsolódó entitásokhoz tartozó GetManyXXX függvényeket, melyek mind a *`GetManyProducts`* függvényhez hasonlóan működnek és ugyanúgy saját EntityResolvert használnak a háttérben.
4. Ezután rendelkezésünkre fognak állni a termékek a hozzájuk kapcsolódó entitások, ezeken még néhány aggregációs műveletet elvégezve megkapjuk a kívánt végeredményt.

A konkrét kód ezek alapján így nézett ki:

```
[ComputeMethod]
public virtual async Task<PageResponse<ListProductDto>>
  Search(ProductSearchDto searchDto)
{
  var result = new List<ListProductDto>();
  var productIdList = await GetProductIdListForFilter(
    searchDto.Filter, searchDto.PageRequest);
  // Termék alapadatok
  var products = await _productService.GetManyProducts(productIdList);

  // Értékelés entitásokból származó adatok
  var reviews = await _productReviewService.GetManyReviews(
    products.SelectMany(p => p.ProductReviewIds).ToList());

  // Raktárkészlet entitásokból származó adatok
  var inventories = await _productInventoryService.GetManyInventories(
    products.Select(p => p.ProductId).ToList());

  // További kapcsolódó entitás lekérdezések...

  // További aggregációs műveletek a kapott adatokból, értékelés átlag
  // számítás, raktárkészlet összegzés, legjobb akció kiválasztás stb.
  // ...

   return new PageResponse<ListProductDto>(result,
     productIdList.CurrentPage, productIdList.TotalCount);
 }
```

Ezekkel a lépésekkel elértük, hogy az összes termékhez és a hozzájuk kapcsolódó entitáshoz meghívásra kerüljön egy ID alapján lekérdező *ComputeMethod* amelyet könnyű invalidálni és jól cachelhető. Az *EntityResolver* segítségével annyit nyertünk, hogy ezek a függvényhívások ne legyenek külön adatbázis lekérdezések, azonban még így is az ID listával meghívott *GetManyXXX* függvények külön fognak futni, tehát jelen esetben 6 féle entitás esetén ugyanennyi külön adatbázis kérésünk lesz, szemben a tradicionális alkalmazás 1 kérésével. Fontos azonban, hogy csak üres cache esetén lesz ilyen nagy a különbség, az első lefutás után vagy nem is kell adatbázishoz nyúlnunk mert cache-ben tárolva lesz az eredmény, vagy ha adatbázishoz is kell nyúlnunk akkor is csak a megváltozott adatokat kérdezzük le újra.

Érezhető, hogy ugyanazt a funkcionalitást több kóddal tudtuk megoldani, bár a reaktív megközelítés rendszerszintű követésével lesznek bizonyos előnyeink (lásd 5.1) itt ez egy nyilvánvaló hátrány. Fontos azt is meggondolni, hogy ez a plusz komplexitás igazából nem magukból a tervezési mintákból adódik, hanem a relációs adatbázisokból származó adatok reaktív kezelésének nehézségeiből.

4.3.4 Időalapú invalidáció – akciók kezelése

Az elkészült alkalmazás a követelményeknek megfelelően kezel akciókat is az egyes termékekre. A *GetValidSpecialOffersIdsForProduct* függvény visszaadja azoknak az akcióknak az azonosítóját melyek a paraméterként kapott termékre vonatkoznak és érvényesek, az akciók ugyanis szokás szerint rendelkeznek egy érvényességi intervallummal. Az eddigiekben láthattuk, hogy hogyan tudjuk invalidálni a függvényeket a módosító műveletek hatására, itt azonban arra van szükség, hogy adott idő eltelte után invalidáljunk adatokat, mintha maga az idő lenne a külső függőségünk.
Ezt úgy oldottam meg hogy definiáltam egy *TimeService* nevű *ComputeService* osztályt, aminek egy *ComputeMethod* függvénye van a *GetCurrentDate*. Akárhányszor valahol az üzleti logikában szükség volt az aktuális dátumra azt innen kérdeztem le, ezzel függőségeket kiépítve.

```
public class DateService : IDateService
{
  [ComputeMethod]
  public virtual Task<DateTime> GetCurrentDate()
            => Task.FromResult(DateTime.Now.Date);
}
```

```
[ComputeMethod]
public virtual async Task<List<int>> GetValidSpecialOffersIdsForProduct(
  int productId)
{
  var currentDate = await _dateService.GetCurrentDate();

  await using var dbContext = CreateDbContext();

  return await dbContext.SpecialOfferProducts.AsQueryable()
    .Where(s => s.ProductId == productId)
    .Where(s => s.SpecialOffer.StartDate.Date <= currentDate.Date
             && s.SpecialOffer.EndDate.Date >= currentDate.Date)
    .Select(s => s.SpecialOfferId)
    .ToListAsync();
}
```

Innentől kezdve már csak arról kellett gondoskodnom, hogy a *GetCurrentDate* függvény értékét a dátum változásakor invalidáljam, hogy az dátumot felhasználó lekérdezések mint például a *GetValidSpecialOffersIdsForProduct* szintén invalidálódjanak. Egy ilyen *TimeService* osztály bevezetésére a Fusion dokumentáció is ad ajánlást [32], azonban ott az invalidációt fix időközönként végezték el a keretrendszer egy másik beépített megoldásával, nekem viszont erre csak minden nap éjfélkor volt szükségem. Ezért végül ezt az invalidációt egy ütemezett **Hangfire job (3.1.2)** segítségével oldottam meg, melyet a backend alkalmazás minden nap éjfélkor futtatott.

4.3.5 Adatbázis változás külső forrásból - termék kezelő CRM

A webshop termék, illetve raktárkészlet kezelő funkcióit úgy kezeltem mintha az egy külön komponens lenne, ami nem az általam készített REST API-n keresztül végez módosításokat, hanem adatbázis szinten. Ennek megfelelően az eddig látott invalidációs megoldások itt nem használhatóak, itt lesz szükség a Debezium által készített Kafka események feldolgozására. Ezt gyakorlatban egy **ASP.NET Hosted Service (3.1.1)** háttérfolyamattal valósítottam meg. Az egyes táblák módosításainak kezelésére külön háttérfolyamatokat definiáltam, példaképpen a termék módosításokat feldolgozó *ProductInventoryKafkaConsumerService* osztályt ismertetem röviden.

```
public class ProductInventoryKafkaConsumerService : BackgroundService
{
  private string topic = "adventureworks.Production.Product";
  private readonly IProductService _productService;
  private ConsumerConfig _kafkaConfig = new ConsumerConfig
```

```
  {
    GroupId = "backend",
    BootstrapServers = "localhost:9092",
    AutoOffsetReset = AutoOffsetReset.Earliest
  };
```

A háttérfolyamatként való futtatás miatt le kell származnunk a *BackgroundService* osztályból. Az invalidációs műveletekhez szükségünk lesz egy *IProductService* példányra, ezt szokásos módon DI segítségével kértem el. Itt láthatóak még a Kafka konfigurációs beállítások, a topic amire feliratkozunk, ami igazából az adatbázis tábla nevéből származik, illetve a Kafka üzenetbusz címe.

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
  using (var consumerBuilder = new
  ConsumerBuilder<Ignore,string>(_kafkaConfig).Build())
  {
    consumerBuilder.Subscribe(topic);
    var cancelToken = new CancellationTokenSource();
    try
    {
      // Esemény feldolgozó ciklus
      // ...
    }
    catch (OperationCanceledException) {}
}
```

A háttérfolyamat végrehajtásakor az *ExecuteAsync* függvény kerül majd futtatásra, ahol a tényleges invalidáció is történni fog. Itt felépítünk egy Kafka Consumer objektumot átadva a konfigurációt, majd feliratkozunk az adott tábla változásainak eseményeire a *Subscribe* függvény segítségével. Maga a feldolgozó ciklus a következőképpen néz ki:

```
while (true)
{
  var consumer = consumerBuilder.Consume(cancelToken.Token);
  var productEvent = JsonSerializer.Deserialize
    <ProductInventoryChangeEvent>(consumer.Message.Value);

  switch (productEvent.Payload.Op)
  {
    // Termék törlése esemény
    case "d":
      using (Computed.Invalidate())
      {
        _ = _productService.TryGetProduct
            (productEvent.Payload.Before.ProductID);
        _ = _productService.GetAll();
```

```
        }
        break;

    // További Update és Insert események hasonló feldolgozása...

}
```

A meghívott *Consume* függvény várakozik a bejövő eseményekre, ezeket Json formátumban kapjuk meg, ezeknek a deszerializálására ChangeEvent osztályokat hoztam létre, mint a példában a *ProductChangeEvent*. A módosított entitások a *Payload* mezőben találhatóak, ezen belül az *After* és *Before* mezők adják meg a módosítás előtti és utáni állapotot. Az *Op* mező alapján el tudjuk dönteni, hogy milyen jellegű módosító művelet történt (Insert/Update/Delete). Ezután történik a tényleges invalidáció a *using (Computed.Invalidate())* blokkon belül, az itt meghívott *ComputeMethod* függvények automatikusan invalidálódnak.

4.3.6 Jogosultágtól függő adatok kezelése – saját megrendeléseim lekérdezése

Sok esetben előfordul, hogy egy adott lekérdezés visszatérési értéke függ attól, hogy milyen felhasználóként vagyunk bejelentkezve, illetve hogy milyen jogosultságok vannak a fiókunkhoz társítva. Ez befolyásolja az ismertetett cachelési mechanizmust is, hiszen egyes függvények értékét felhasználónkként külön cache bejegyzésben kell tárolnunk. Emiatt a Fusion keretrendszer saját autentikációs mechanizmussal is rendelkezik [33], itt azt mutatom be, hogy ezt hogyan használtam fel egy konkrét funkció, a felhasználó saját megrendeléseinek listázásakor, illetve hogy hogyan kapcsoltam ezt össze az ASP.NET autentikációval, illetve az Identity keretrendszerrel.

A Fusion rendelkezik egy saját ***Session***-nel, ez igazából egy egyedi szöveges érték, amit a kliens az egyes kérések mellé felküld és kapcsolódik egy adatbázis bejegyzéshez, ahol többek között azt tároljuk, hogy ez a *Session* mely felhasználói fiókhoz tartozik (nem bejelentkezett felhasználó esetén egyikhez sem). Ezt a *Session*-t a *UseFusionSession* middleware hozza létre egy HTTP only cookie formájában. Beépített szolgáltatások segítségével, el tudom kérni ezt a Session-t az alkalmazás tetszőleges pontján és lekérdezni az aktuálisan hozzá társított felhasználót. Az ajánlott megoldás az, hogy a *ComputeMethod* függvényeink paraméterként kapják ezt a Session-t, így eltérő értékekre más cache bejegyzések készülnek majd és más függőségek épülnek

ki. A következő kódrészletben is ez történik, ahol elkérem az aktuális felhasználót és a hozzátartozó azonosító segítségével kérdezem le a megrendeléseket adatbázisból.

```
[ComputeMethod]
public virtual async Task<List<OrderHeaderDto>> GetMyOrders(Session session)
{
  var sessionInfo = await _authService.GetSessionInfo(session);
  if(sessionInfo.IsAuthenticated())
  {
    var user = await _authService.GetUser(session);
    if(user.Claims.ContainsKey("read_orders"))
    {
      var userId = long.Parse(user.Id.Value);
      // Lekérdezés adatbázisból user id felhasználásával...
```

Felmerülhet bennünk, hogy használhatnánk a felhasználó ID-ját is mint paraméter, hogy külön cache bejegyzéseket alakítsunk ki. A Session használatának egyrészt az az előnye, hogy a Fusion sok beépített segédfüggvényt ad, mint például az `IAuthBackend` osztály bejelentkezés, kijelentkezés és jogosultság módosítás függvényeit melyek a Session-t és a hozzá társított felhasználót módosítják és automatikusan elvégzik az invalidációt. Az előző példában látható `_authService.GetUser()` függvény is egy ilyen ComputeMethod. A Session használatának másik indoka pedig az, hogy a ComputeService szolgáltatásoknak sokszor van kliens oldali ReplicaService másolata, melynek az interfésze megegyezik. A kliens alkalmazásra viszont nem bízhatjuk rá, hogy tetszőleges felhasználói ID-t átadva végezzen műveleteket, mert ez nem lenne biztonságos.

Session használatával tehát láthatóan felhasználófüggővé tehetjük az egyes ComputeMethod függvényeket, az itt bemutatott osztályok azonban tényleges autentikációt nem végeznek, csak az ezzel kapcsolatos információk lekérdezésére és ezek egyszerű invalidálására szolgálnak. Tényleges autentikációra az ASP.NET keretrendszer megoldásait használtam. Ebben a keretrendszerben az adott beérkező kéréshez tartozó autentikációs információk egy **`HttpContext`** objektumban érhetőek el, alapeseteben ezt szoktuk arra használni, hogy elkérjük az aktuális felhasználó azonosítóját vagy jogosultságait. Ahhoz, hogy ezt a két megoldást összekapcsoljam az itt tárolt információkat kellett átvezetnem a Fusion által használ Session-be, ezt konkrétan a `_Host.cshtml` fájlban valósítottam meg, mert az itt megadott kód minden oldalbetöltéskor lefut (lásd 3.1.6 Szerver oldali előrenderelés). Ennek a szinkronizációnak a legfontosabb részei látszanak a következő kódrészletben. Az `UpdateAuthState` beépített Fusion

függvény az aktuális HttpContext autentikációs adatait szinkronizálja azáltal, hogy meghívja a be és kijelentkezést elvégző függvényeket, amelyek az adott Session adatait módosítják. Ezután előrendereljük az alkalmazásunkat az <app> tagen belül és átadjuk neki az aktuális Session értékét.

```
@page "/"
@inject ServerAuthHelper _serverAuthHelper
@inject BlazorCircuitContext _blazorCircuitContext
@{
    await _serverAuthHelper.UpdateAuthState(HttpContext);
    var sessionId = _serverAuthHelper.Session.Id.Value;
}

<app id="app">
        @{
            using var prerendering = _blazorCircuitContext.Prerendering();
            var prerenderedApp = await Html.RenderComponentAsync<App>(
                RenderMode.WebAssemblyPrerendered,
                new { SessionId = sessionId });
        }
        @(prerenderedApp)
 </app>
```

A tényleges bejelentkezést, aminek hatására bekerülnek az adott felhasználó adatai a HttpContext objektumba, illetve a felhasználókezelést is az ASP.NET Identity keretrendszer segítségével valósítottam meg.

4.4 Frontend funkciók implementálása

4.4.1 Websocket invalidációs mechanizmus bekapcsolása

Ahhoz, hogy bekapcsoljuk a korábban ismertetett Websocket invalidációs értesítést a kliens számára mindössze annyi a dolgunk, hogy az egyes adatlekérdező Controller végpontjainkat ellátjuk a Publish attribútummal.

```
[Route("api/[controller]/[action]")]
[ApiController]
public class ProductController : ControllerBase, IProductService
{
  private readonly IProductService _productService;

  public ProductController(IProductService productService)
  {
    _productService = productService;
  }
```

```
  [HttpGet, Publish]
  public Task<ProductDto> GetProduct(int productId)
    => _productService.GetProduct(productId);
```

Mint ahogy azt korábban említettem a kliens a szerver oldallal való kommunikációt ReplicaService osztályok segítségével végzi el. Az ilyen osztályokat a ComputeService osztályokhoz hasonlóan regisztrálnunk kell, a következőhöz hasonló módon.

```
var fusionClient = fusion.AddRestEaseClient(
  options => { options.BaseUri = baseUri; // ... });
fusionClient.AddReplicaService<IProductService, IProductClientDef>();

[BasePath("product")]
public interface IProductClientDef
{
  [Get("getProduct")]
  public Task<ProductDto> TryGetProduct(int productId);
  // ...
```

Konfigurációként először meg kell adnunk a címet, ahol a backend alkalmazás elérhető lesz, majd felsorolhatjuk az egyes Service osztályokat. Az interfész ugyanaz lesz, mint backend oldalon, a konkrét megvalósítás viszont egy `ClientDef` végződésű HTTP kliens osztály. Itt megtévesztő lehet, hogy a megvalósítás is egy interfész, de ez a RestEase könyvtár működése miatt van így, ebből az interfészből ez a könyvtár későbbiekben konkrét implementációt tud majd nekünk generálni. A regisztráció után ezt a `IProductService` osztályt el tudjuk kérni a konkrét komponenseinkben. Példaképpen itt látható egy nagyon egyszerű komponens, ami egy termék alapadatait jeleníti meg.

```
@page "/product/{Id:int}/baseData"
@inject IProductService ProductService
@inherits ComputedStateComponent<ProductDto>

@{
    var state = State.ValueOrDefault ?? new ProductDto();
    var error = State.Error;
}

// Felület HTML template...

@code {
    [Parameter]
    public int Id { get; set; }

    protected override async Task<ProductDto> ComputeState(
      CancellationToken cancellationToken)
```

```
    {
        return await ProductService.TryGetProduct(Id, cancellationToken);
    }
}
```

Az egyes komponenseinket általában a *ComputedStateComponent* osztályból származtattam, ez igazából egy Computed értéket tárol és tart konzisztensen a `State` mezőn belül. A komponens, ha invalidációs eseményről értesül akkor automatikusan újból kiszámítja ezt az értéket a `ComputeState` függvény segítségével.

4.4.2 Adatok megosztása komponensek között - kosárkezelés

Az előző példában láthatóan nem nagyon jár extra komplexitással a reaktív megközelítés, sőt a reaktív alkalmazásom kliens oldali része lényegesen kevesebb kóddal implementálható volt, mint tradicionális esetben. Ennek az volt az oka, hogy tradicionális megközelítés esetén amikor valamilyen adatmódosító műveletet kezdeményeztem kliens oldalon akkor annak eredményét több helyre át kell vezetnünk a felületen. Erre jó példa a kosár kezelés, ugyanis egy adott termék listanézetét adó `ProductListItem` komponensben megjelenik, hogy az adott termék kosárban van-e már, illetve az aktuális kosár tartalma állandóan látható a kliens alkalmazásban az oldalsávot alkotó `Leftbar` komponensben. A tradicionális alkalmazásban ennek kezelésére be kellett vezetnem egy `AppStateContainer` osztályt, ahol a kosár tartalmát tároltam kliens oldalon és itt különböző módosító függvényeket és eseménykezelőket kellett definiálnom. Ezután pedig az összes komponensben, ahol a kosár adataira szükség volt fel és le kellett iratkoznom ezekre az eseményekre és kezelnem a különböző változásokat. Ezen osztály forráskódjának egy részlete látható itt.

```
public class AppStateContainer
{
  public List<ShoppingCartItemDetailsDto> ShoppingCartItems { get; set; }
  public event Action OnShoppingCartChange;

  public void AddShoppingCartItem(ShoppingCartItemDetailsDto newItem)
  {
    ShoppingCartItems.Add(newItem);
    NotifyShoppingCartStateChanged();
  }

  public void RemoveShoppingCartItem(int shoppingCartItemId)
  {
    var itemToRemove = ShoppingCartItems
      .FirstOrDefault(sci => sci.ShoppingCartItemId == shoppingCartItemId);
```

```
    if(itemToRemove != null)
    {
      ShoppingCartItems.Remove(itemToRemove);
      NotifyShoppingCartStateChanged();
    }
  }

 public void UpdateShoppingCartItem(int shoppingCartItemId,
   UpdateShoppingCartItem updateData)
 {
   var itemToUpdate = ShoppingCartItems
     .FirstOrDefault(sci => sci.ShoppingCartItemId == shoppingCartItemId);
   if (itemToUpdate != null)
   {
     // updateData adatainak beállítása
     NotifyShoppingCartStateChanged();
   }
 }

 private void NotifyShoppingCartStateChanged()
   => OnShoppingCartChange?.Invoke();
```

Itt konkrétan a *ShoppingCartItems* mezőben tároltam a kosár tartalmát, a különböző függvények ezt módosítják. Van egy `OnShoppingCartChange` nevű esemény melyet minden ilyen módosítás után kiváltottam, az egyes Blazor komponensek erre iratkoznak fel, hogy értesülni tudjanak a változásról. Itt egy ilyen komponens a *ProductListItem* kódjának egy részlete látható.

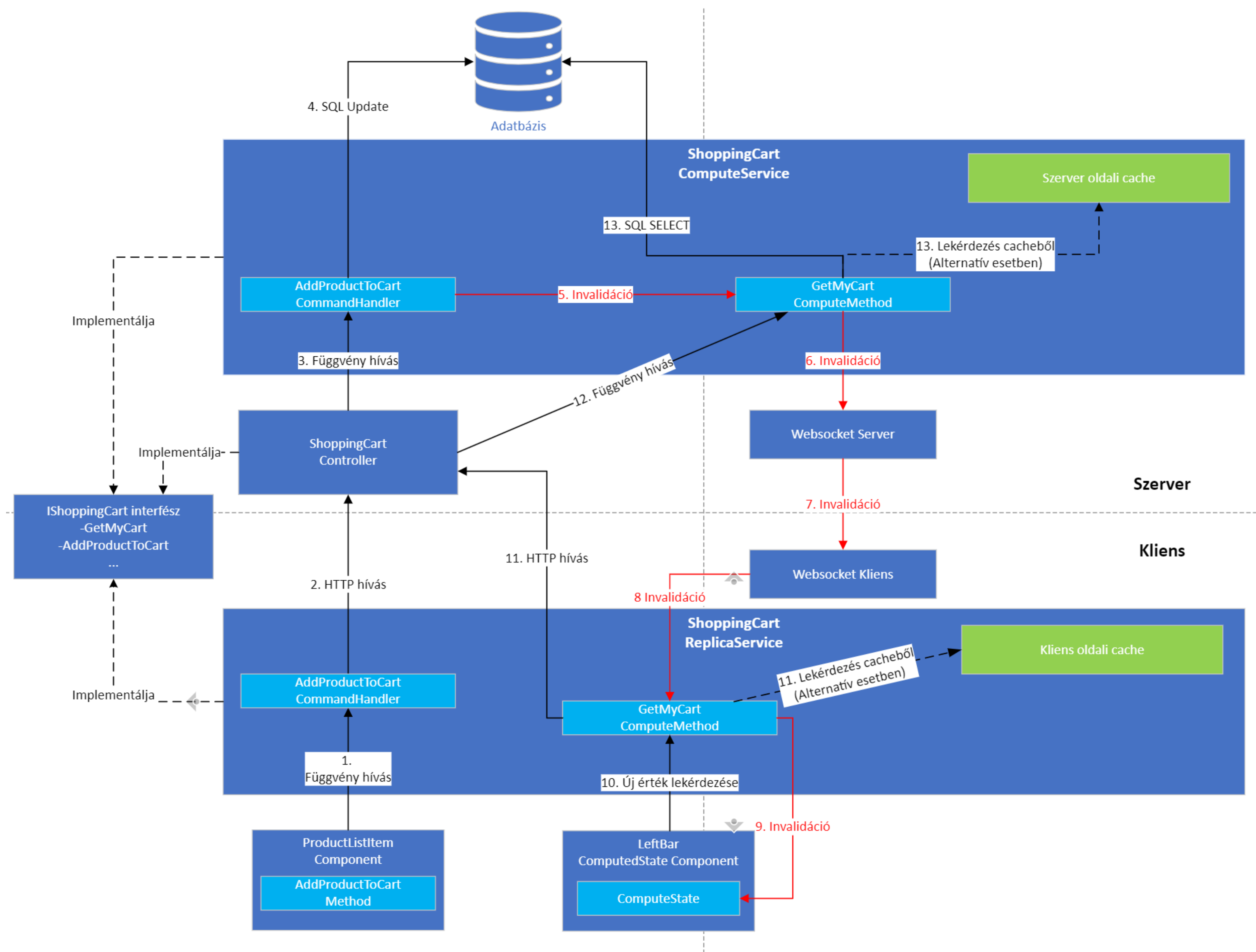
```
@inject AppStateContainer AppState

@if(state.ShoppingCartItems.Any())
{
  @foreach (var cartItem in state.ShoppingCartItems)
  {
    // Kosár elem megjelenítését végző template...
  }
}

protected override void OnInitialized()
{
  AppState.OnShoppingCartChange += StateHasChanged;
  // További inicializációs logika...
}

public void Dispose()
{
```

```
  AppState.OnShoppingCartChange -= StateHasChanged;
}
```

A feliratkozás konkrétan az `OnInitialized` életciklus függvényben történik, itt az esemény bekövetkezésekor meghívtam a *`StateHasChanged`* függvényt, amivel minden Blazor komponens rendelkezik (ComponentBase ősosztályból származik) és arra szolgál, hogy a felület újra renderelését kezdeményezzük. A komponens megsemmisülésekor meg fog hívódni a *`Dispose`* függvény, ahol pedig leiratkoztam erről az eseményről. Az egyes műveletek elvégzésekor az egyes API hívások eredményét átvezettem ebbe a kliens oldali állapotot tároló `AppStateContainer` osztályba.

```
public async Task AddToShoppingCart()
{
  var createdCartItem = await ShoppingCartDetailsClient
    .AddProductToCartAsync(
      new AddShoppingCartItemCommand
      {
        ProductId = Product.ProductId,
        Quantity = 1
      });
    AppState.AddShoppingCartItem(createdCartItem)
  }
```

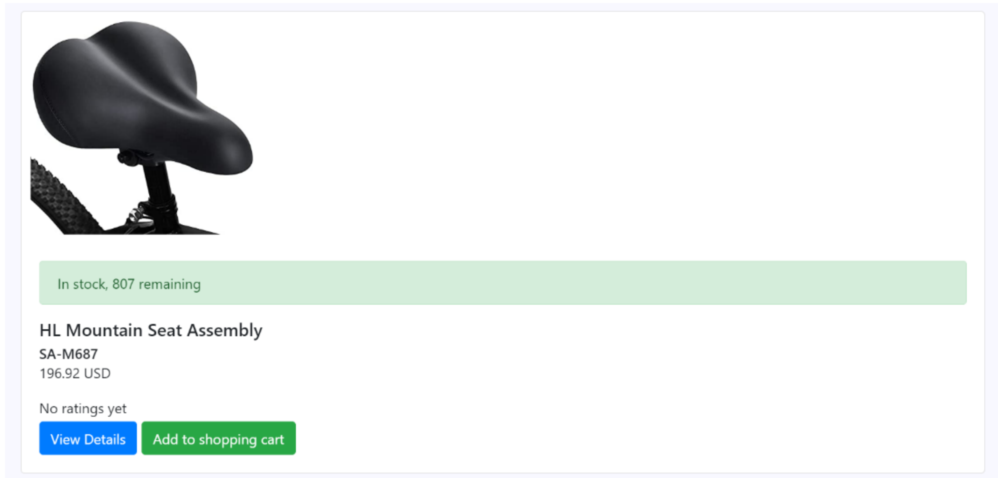
Ilyen típusú szinkronizációra a kliens és szerver oldali állapot között szükség lehet minden olyan esetben, ahol olyan adattal dolgozunk melyre több komponensnek is szüksége van. Ez itt még egy elég egyszerű példa, de komplexebb esetben sokszor szükség lehet igen bonyolult állapotkezelési minták használatára, mint például a Flux [34] vagy a Redux [35]. Viszont erre a reaktív példa alkalmazásomban nincs szükség, hiszen minden komponens fel van iratkozva az adatok megváltozására és frissíti magát, ilyen módon a közös adatokon dolgozó komponensek konzisztensen tartása **nem jár semmilyen plusz kóddal**. Ennek pontos működését szintén a kosárkezelés példáján próbálom bemutatni a következő ábrán. A konkrét forgatókönyv az, hogy a *`ProductListItem`* komponensen az aktuális terméket kosárba helyezzük és ez invalidálja a kosár tartalmát a *`Leftbar`* komponensen.



15. ábra Komponensek közti invalidáció

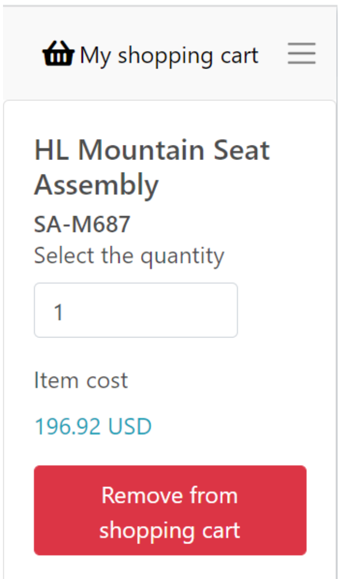
A folyamat lépéseit számozással láttam el, a könnyebb követhetőség érdekében.

- A 1-3. lépésekben először kliens oldalon a felhasználó megnyomja a kosárba helyezés gombot a `ProductListItem` komponensen, ennek hatására lefut a ReplicaService HTTP kérése, ami egy Controlleren át beérkezik az `AddProductToCart` CommandHandler függvényhez.

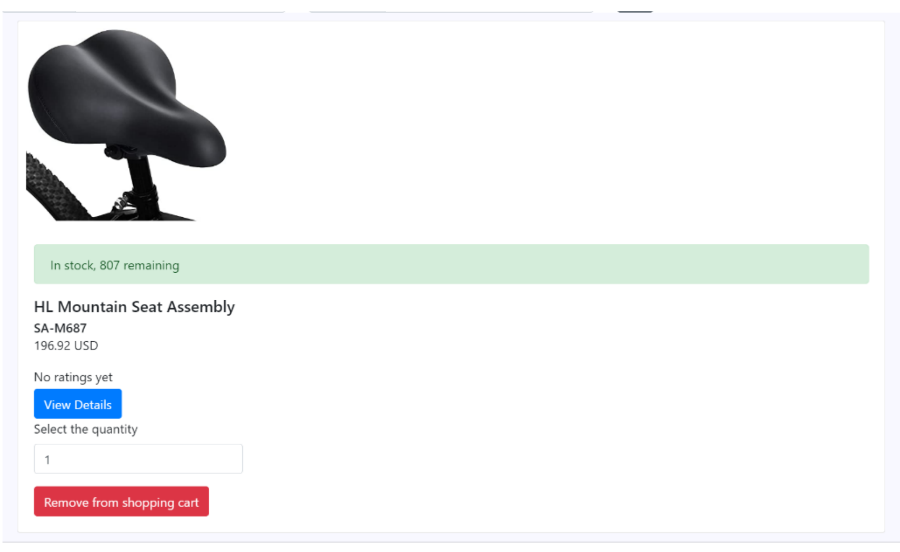


16. ábra ProductListItem komponens, kosárba helyezés előtt

- 4-5. lépésekben megtörténik a kosár módosítása adatbázisban és emiatt a kosár tartalmát lekérdező függvények invalidálásra is kerülnek.
- 6-9. lépésekben ez az invalidációs üzenet többek között egy websocket üzenet segítségével eljut a `Leftbar` komponensbe. Itt tudja meg a komponens, hogy általa használt adatok már nem tükrözik a legfrissebb állapotot, az ő döntése azonban, hogy lekérdezi-e az új állapotot.
- Itt úgy döntünk, hogy igen, mert üzletileg fontos, hogy folyamatosan a kosár legfrissebb állapotát lássuk, ezért meghívjuk a `ReplicaService GetMyCart` függvényét, ami az előzőhöz hasonló módon megint beérkezik a szerver oldali `ComputeService` osztályunkba a 10-12. lépések során. A 11-es lépésnél fontosnak tartottam megemlíteni, hogy az ilyen függvény hívásokból nem mindig lesz http hívás, előfordulhat ugyanis, hogy mondjuk egy másik komponens már előttünk kiszámította ezt az értéket mert neki is szüksége volt rá, ebben az esetben ezt egyszerűen ki tudjuk olvasni a kliens oldali cache tárolóból.
- A 13-as lépésben pedig szerver oldalon lekérdezzük adatbázisból a kosár új tartalmát, amit az előbb elmondtam az itt is igaz, hogy ha valaki ezt már kiszámolta akkor szintén fel tudjuk használni a cachelt értéket.
- Ezután a 13-as pontban kiszámolt új érték cachelésre kerül és a HTTP kérésre adott válaszban visszajut a kliensre. A visszafelé történő adatáramlást az ábrán már nem szemléltettem az átláthatóság miatt.



17. ábra LeftBar komponens, kosárba helyezés után



18. ábra ProductListItem komponens, kosárba helyezés után

4.4.3 Felület állapota mint lokális függőség – termék keresés

Az előző példában láthattuk, hogy egy komponensben hogyan tudunk értesülni a különböző adatmódosító műveletek okozta invalidációról. Sokszor előfordul azonban. hogy felületen lévő inputok aktuális állapotától is függ, hogy milyen adatokat jelenítünk meg. A termékek közötti keresés megvalósításakor is ez volt a helyzet, ahol a felületen megadott szűrési és rendezési feltételek változása esetén frissítenem kellett a megjelenített termékeket, Erre a Fusion könyvtár által nyújtott `MixedStateComponent` osztályt használtam, aminek megadtam a helyi és a távoli állapot típusát paraméterként.

```
@page "/products"
@inherits MixedStateComponent<ProductListComputedState,
ProductListLocalModel>
@inject IProductSearchService ProductSearchService
```

A *ProductListLocalModel* tárolta a szűrési és rendezési feltételeket. Azzal, hogy leszármaztunk a MixedStateComponent ősosztályból kaptunk egy **MutableState** mezőt, ami a kliens oldali állapotot tárolja egy Computed példányba csomagolva.

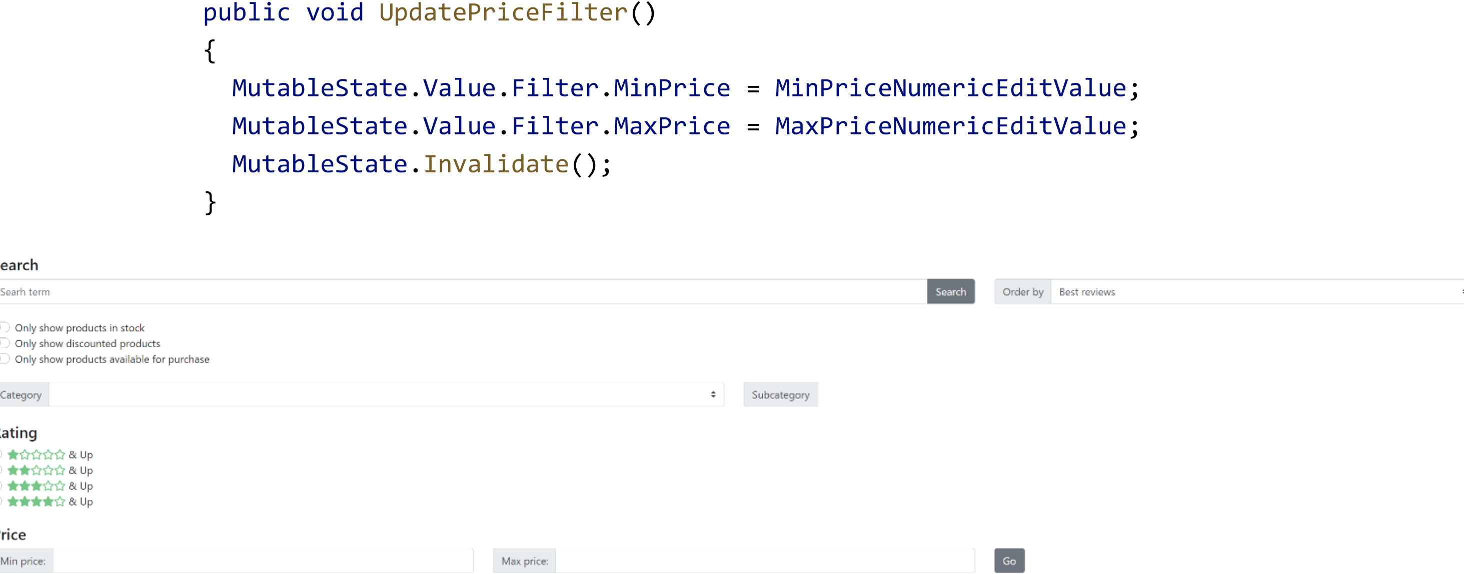
```
@{
    // Szerverről jövő állapot, termék eredménylista,
       ProductListComputedState
    var state = State.ValueOrDefault ?? new ProductListComputedState();
    var error = State.Error;

    // Szűrők állapota, ProductListLocalModel-t tárol
    var localState = MutableState;
}
```

Ezt a helyi állapotot felhasználtam a termék keresés API hívás elvégzésekor, mely a *ComputeState* függvényben történik. Itt azt kellett elérnem, hogy a szűrők változása esetén ez mindig fusson le újra. Mivel ezek a szűrők be vannak csomagolva egy *Computed* példányba ezért megadhatóak függőségként, ezt konkrétan a *Computed.Use()* metódus meghívásával tudtam elérni. Innentől kezdve, ha a *MutableState.Computed* invalidálásra kerül akkor ez a függvény újra le fog futni, ezzel frissítve a keresési eredményeket.

```
protected override async Task<ProductListComputedState> ComputeState()
{
  var locals = await MutableState.Computed.Use();

  var results = await ProductSearchService.Search(
    new ProductSearchDto {
      Filter = locals.Filter,
      PageRequest = locals.PageRequest
    });

   // ...
```

A konkrét invalidációt pedig az `Invalidate` függvény segítségével tettem meg, amit a szűrőfeltételeket módosító függvényekben hívtam meg.

```
public void UpdatePriceFilter()
{
  MutableState.Value.Filter.MinPrice = MinPriceNumericEditValue;
  MutableState.Value.Filter.MaxPrice = MaxPriceNumericEditValue;
  MutableState.Invalidate();
}
```



19. ábra Szűrési feltételek

5 Összefoglalás, kitekintés

Dolgozatomban megmutattam, hogy hogyan lehet összeállítani meglévő komponensekre építve egy alapvetően klasszikus többrétegű, ugyanakkor minden rétegben reaktív elveket követő architektúrát.

5.1 Előnyök, rendszerszintű többletfunkciók

Az előző fejezetben bemutattam, hogy hogyan alkalmazható a reaktív megközelítés konkrét üzleti funkciók implementálásakor, ez bizonyos esetekben plusz komplexitással vagy nem megszokott megoldásokkal járt. Itt röviden összefoglalom, hogy milyen plusz rendszer szintű funkciókat nyertünk ezzel az egésszel és hogy miért érhette meg végső soron ezt a megközelítést követni.

5.1.1 Cachelés szerver és kliens oldalon

A szerver oldali cachelés miatt az API kérések válaszideje sokkal kedvezőbb lehet. Egyrészt azért, mert ha az utoljára kiszámított Computed érték érvényes akkor meg tudjuk előzni azt, hogy adatbázishoz kelljen fordulnunk másrészt pedig, ha a kiszámított érték nem is érvényes, akkor is sok esetben a felépített függőségi gráf miatt rendelkezésre fognak állni különböző részeredményeink és csak a ténylegesen megváltozott adatokat kell adatbázisból lekérdeznünk. Ezen felül pedig a kliens oldali cachelés pedig csökkentheti a kliens és szerver közti hálózati kérések számát.

5.1.2 UI komponensek függetlensége, egyszerűbb API kommunikáció

Frontend komponensek nagyon egyszerűek, a legtöbb komponens pontosan egy adat lekérdező végpontot használ, amit ráadásul adat változás esetén automatikusan újra meghív. Ezzel az állapotkezelés nagy mértékben egyszerűsödik, emiatt reaktív a kliens alkalmazás **kevesebb kóddal megvalósítható** volt, mint a tradicionális.

Ezen felül a frontend alkalmazásunk **sokkal könnyebben bővíthető** lesz azáltal, hogy a komponenseink teljesen függetlenek lesznek egymástól, hiszen az egyes adatváltozásokról mind szerver oldalról értesülnek, nem feltétlenül szükséges egymás között kommunikálniuk. Ez a függetlenség ezen felül lehetőséget adhat például a **Micro frontends** [36] architektúra stílus használatára.

5.1.3 Jobb felhasználói élmény, reszponzivitás

A valós idejű felület egyes esetekben gazdagabb felhasználói élményt eredményez és bizonyos üzleti funkciók használhatóságát nagyban megnöveli. Itt egy-két konkrét példát próbálok erre mutatni.

- Referencia alkalmazás esetén előfordulhat, hogy egy termék adatlapjára navigálva azt látjuk, hogy készleten van, de amikor a kosárba akarjuk tenni akkor derül ki, hogy az oldal betöltése óta már kifogyott. Ez a felhasználó számára
- Webshop alkalmazások esetén sokszor a felhasználók több böngésző fület tartanak nyitva, ahol például az egyes termékek adatlapjait tekintik meg. Ilyenkor az általálam készített tradicionális implementáció és a legtöbb interneten elérhető népszerűbb webshop esetén is igaz, hogy az adott böngésző fülön végzett műveletek nem jelennek meg más füleken. Például, ha az egyik fülön a terméket a kosárba helyezzük, akkor ez a másik fülön nem jelenik meg a kosárban csak frissítés után. Ez esetleg megzavarhatja a felhasználót és előfordulhat, hogy csak később mondjuk a fizetés előtt veszi észre, hogy a kosara nem pont azt tartalmazza, amire számított. A reaktív implementáció esetén ez nem fordulhat elő mert a felület naprakész adatokat fog tartalmazni akkor is, ha az alkalmazást több külön böngésző tabról, vagy akár különböző eszközökről megnyitva használjuk.
- A kosár foglalási rendszer reaktív esetben sokkal jobb élményt nyújt, mert azonnal értesülni tudunk arról, hogy egy termék elérhetővé vált, ez a gyakorlatban azt eredményezheti, hogy a felhasználók bátrabban fognak olyan termékeket a kosarukba helyezni, amelyek egyelőre mások számára foglaltak, hiszen tudják, hogy egyszerűen értesülnek majd azok felszabadulásáról. Ez akár hozzájárulhat a webshop forgalmának növeléséhez

5.1.4 Adatbázis szintű integráció más alkalmazásokkal

Az előbb felsorolt előnyök (cachelés, valós idejű felület) akkor is megmaradnak, ha a reaktív alkalmazásunkat olyan más alkalmazásokkal kapcsoljuk össze melyek nem a mi API felületünkön végeznek adatmódosítást. Ez megkönnyíti az együttműködést abban az esetben, ha több alkalmazásunk használ azonos adatbázist és ezek közül csak néhány esetén tartjuk érdemesnek a reaktív architektúra bevezetését.

5.2 Hátrányok, architektúra korlátai

A konkrét implementáció során már valamennyire látható volt, hogy milyen hátrányokkal és plusz komplexitással jár a kialakított architektúra, itt ezeket próbálom összefoglalni és kicsit értékelni őket.

5.2.1 Nagyobb backend komplexitás

A reaktív implementáció backend részének komplexitása érezhetően nagyobb, mint tradicionális esetben. Ezt főképpen következő okokból adódik:

- **CDC események**: Az érkező adatváltozási események feldolgozása egy olyan plusz feladat, ami tradicionális esetben nincs jelen.
- **Adatok közti függőségek kiépítése**: Ahogy azt a 4.3.3 fejezetben is láthattuk, ahhoz, hogy adatbázisból származó adataink között kiépüljenek függőségek, gyakran máshogy kell a lekérdezéseinket megfogalmazni és ez a megközelítés több lépésből áll, illetve több függvényt igényel, mint tradicionális esetben
- **Invalidációs blokkok**: szintén a függőségek kiépítése szempontjából van rá szükség, azonban ez az adatmódosításoknál jelentkezik, ebből is származik plusz komplexitás.

Ez a plusz komplexitás forráskód szintjén is jelentkezik, itt láthatóak a két alkalmazás backend részeinek forráskód (és egyéb komplexitással kapcsolatos) mérőszámai.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
API (Debug)	71	81	4	160	590	315
Dipterv.Bll (Debug)	76	262	2	148	1 577	699
Dipterv.Dal (Debug)	79	2 061	5	152	5 805	4 394

20. ábra Tradicionális implementáció backend, kód metrikák

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
API (Debug)	72	133	4	255	946	508
Dipterv.Bll (Debug)	79	457	2	189	2 296	1 192
Dipterv.Dal (Debug)	79	2 075	5	157	5 815	4 399

21. ábra Reaktív implementáció backend, kód metrikák

A DAL réteget alkotó projekt szinte megegyezik a két alkalmazásban, ezért az ezzel kapcsolatos metrikák nekünk most nem fontosak. Viszont az API és BLL projektekben a forráskódot (Lines of Source Code) és strukturális komplexitást (Cyclomatic Complexity) jelző mérőszámok mind nagyobbak/rosszabbak voltak reaktív

esetben. Ezeket a méréseket Visual Studio segítségével végeztem, az egyes mérőszámokról itt [37] található további információ.

5.2.2 Nagyobb memória használat

A Fusion könyvtár nyújtotta cachelés miatt a reaktív alkalmazás több memóriát használ. Értelemszerűen szinte minden hasonló megoldás ezzel jár, azonban ebben az esetben szinte minden függvény kimenetét cacheljük, így ez fokozottan igaz. Erre valamilyen szinten megoldást nyújthat a keretrendszer **swap** megoldása [38], ami az adott idő után nem használt cache bejegyzéseket a háttértárra tölti át. Azonban ez ronthatja a teljesítményt, így hogy konkrétan milyen adatok esetén és milyen idő paraméterrel használjuk azt érdemes lehet meggondolni, illetve különböző méréseket végezni ezzel kapcsolatban.

5.2.3 Fusion könyvtár

Az architektúra nagy mértékben épít a Fusion .NET-es osztálykönyvtárra, viszont ezzel kapcsolatban felmerülhetnek bennünk különböző aggályok. Egyrészről a könyvtárat leginkább 1 ember tartja fent, így a hosszú távú támogatás/karbantartás kérdése kétséges lehet. Másrészt pedig még sok új funkció van tervben, és ezek megjelenése általában breaking change jellegű változtatásokkal jár, ami problémákat okozhat számunkra.

Az is probléma lehet még hogy a könyvtárhoz kliens oldali megvalósítás csak .NET-ben készült, viszont az ilyen jellegű webalkalmazásokat még mindig nagyrészt Javascript keretrendszerek segítségével szokták fejleszteni, az ehhez értő fejlesztőket valószínűleg nehéz lesz rávenni a .NET-es kliens használatára. Az is elmondható, hogy a könyvtár Github oldalán lévő példákon túl kevés a könyvtárat használó alkalmazást, illetve segédanyagot találni. A könyvtár autentikáció részhez például konkrétan semmilyen segédanyagot nem találtam, ehhez végül részben én írtam meg a dokumentációt [33]. Az autentikáció megvalósítása pedig egyébként is szokatlan lehet, hiszen nem teszi lehetővé az ASP.NET keretrendszer attribútum alapú authorizációs megoldásait.

5.2.4 Kafka infrastruktúra felállítása nagy plusz üzemeltetési teher

Sok komponens, ha mi futtatjuk dedikált hardveren az nyilván plusz erőforrás igény, illetve ennek a konfigurációjához és az üzemeltetéshez plusz kompetenciák és fejlesztői

erőforrások szükségesek. Vannak menedzselt Kafka futtatási környezetek a különböző felhő platformokon, amelyek ezt egyszerűsítik (pl. Azure HDInsight [39]), de nyilván ezek is valamilyen plusz költséggel járnak.

5.3 Milyen esetekben ajánlanám ezt az architektúrát

- Leginkább akkor, ha a valós idejű felület szükségszerű, vagy nagy mértékben javítja a felhasználói élményt. Fontos megjegyezni, hogy nem feltétlenül szükséges a teljes alkalmazásunkban az itt felvázolt megközelítést követnünk ahhoz, hogy az működőképes legyen, dönthetünk úgy, hogy hogy csak egy adott funkció esetén használjuk azt fel. Ilyenkor meg tudjuk tenni, hogy csak az adott funkcióhoz szükséges adatokra használunk `ComputeMehod` és `CommandHandler` függvényeket, illetve csak ezen adatokat tartalmazó táblákra kapcsoljuk be a CDC-t.
- Komplex felületi logikával rendelkező kliens alkalmazások esetén, ahol a komponensek által felhasznált adatokban sok az átfedés. Ilyen esetekben ugyanis sokat tudunk spórolni azzal, hogy a komponensek közötti kommunikáció és adat szinkronizáció egyszerűbbé válik a szerver oldalról érkező invalidáció miatt. Figyelembe kell vennünk viszont, hogy ezt a komplexitást ilyenkor részben backend oldalra visszük át, ez előny és hátrány is lehet attól függően, hogy milyen fejlesztői kapacitással rendelkezünk backend és frontend oldalon.

5.4 Kitekintés, továbbfejlesztési lehetőségek

Ahogy ezt korábban az implementációs részletek során említettem a reaktív megközelítés minden rétegben való megvalósításának legnagyobb korlátozó tényezője a relációs adatbázisok használata. Valamilyen NoSQL adatbáziskezelő segítségével elméletben lehetséges lenne olyan reaktív adatkezelési megoldás elérése backend és adatbázis között, ami nem jár ennyi plusz komplexitással backend oldalon. Ha mondjuk a MongoDB change stream megoldását használnánk akkor egyáltalán nem lenne szükség az egyes adatbázis lekérdezések manuális invalidációjára.

Egy ilyen megoldás megvalósítható lenne például az Event Sourcing [40] és CQRS [41] mintákra támaszkodva. Ez a gyakorlatban mondjuk úgy nézhetne ki, hogy az elsődleges adattárunk (source of thruth) egy event stream, amiből aggregálunk egy olvasásra használt MongoDB adatbázist, melynek a change stream szolgáltatását

használva értesülünk az adatváltozásokról. Ezután, hogy backend szinten tudunk az adatváltozásról értesülni használhatjuk akár a dolgozatban ismertetett Websocket technológiát arra, hogy ezt eljuttassuk a kliens alkalmazásba.

6 Irodalomjegyzék

[1] "Reactive programming," 13 10 2022. [Online]. Available: https://en.wikipedia.org/wiki/Reactive_programming.

[2] D. Pine és B. Wagner, „Observer Design Pattern," 13 10 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern.

[3] J. Shvarts, „Understanding Marble Diagrams for Reactive Streams," Medium, 29 december 2017. [Online]. Available: https://medium.com/@jshvarts/read-marble-diagrams-like-a-pro-3d72934d3ef5.

[4] „Elm - Documentation," [Online]. Available: https://elm-lang.org/docs. [Hozzáférés dátuma: 13 10 2022].

[5] „Documentation," [Online]. Available: https://svelte.dev/docs. [Hozzáférés dátuma: 13 10 2022].

[6] „Reactive X," [Online]. Available: https://reactivex.io/intro.html. [Hozzáférés dátuma: 27 10 2022].

[7] „Reactive X Operators," [Online]. Available: https://reactivex.io/documentation/operators.html. [Hozzáférés dátuma: 27 10 2022].

[8] Wikipedia, „Pure function," 22 szeptember 2022. [Online]. Available: https://en.wikipedia.org/wiki/Pure_function.

[9] A. Yakunin, „Fusion: the "real-time on!" switch that actually exists," [Online]. Available: https://github.com/servicetitan/Stl.Fusion. [Hozzáférés dátuma: 13 10 2022].

[10] A. Yakunin és A. Filatov, „Part 1: Compute Services," 30 szeptember 2022. [Online]. Available:

https://github.com/servicetitan/Stl.Fusion.Samples/blob/master/docs/tutorial/Part01.md.

[11] A. Yakunin and F. Alexey , "Part 2: Computed Values and Computed<T>," 30 szeptember 2022. [Online]. Available: https://github.com/servicetitan/Stl.Fusion.Samples/blob/master/docs/tutorial/Part02.md. [Accessed 27 október 2022].

[12] A. Yakunin, „QuickStart: Learn 80% of Fusion by walking through HelloCart sample,” 8 július 2022. [Online]. Available: https://github.com/servicetitan/Stl.Fusion.Samples/blob/master/docs/tutorial/QuickStart.md.

[13] J. Archibald, „HTTP/2 push is tougher than I thought,” 30 május 2017. [Online]. Available: https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/.

[14] WHATWG, "Wikipedia," WHATWG, 22 október 2022. [Online]. Available: https://websockets.spec.whatwg.org/.

[15] T. Duarte, „Using WebSockets With Cookie-Based Authentication,” Coletiv, 8 május 2020. [Online]. Available: https://www.coletiv.com/blog/using-websockets-with-cookie-based-authentication/.

[16] A. Yakunin és P. Dräxler, „Part 4: Replica Services,” 30 szeptember 2022. [Online]. Available: https://github.com/servicetitan/Stl.Fusion.Samples/blob/master/docs/tutorial/Part04.md.

[17] Wikipedia, „Wikipedia,” Wikipedia, 19 június 2022. [Online]. Available: https://en.wikipedia.org/wiki/Server-sent_events.

[18] gRPC, „gRPC,” 2 augusztus 2022. [Online]. Available: https://grpc.io/docs/what-is-grpc/core-concepts/#server-streaming-rpc.

[19] Microsoft, „Microsoft,” Microsoft, 21 szeptember 2022. [Online]. Available: https://learn.microsoft.com/en-

us/aspnet/core/signalr/introduction?WT.mc_id=dotnet-35129-website&view=aspnetcore-6.0.

[20] „What is change data capture (CDC)?," Microsoft, 9 augusztus 2022. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/track-changes/about-change-data-capture-sql-server?view=sql-server-ver16.

[21] Debezizum, „Debezium Documentation," Debezizum, 2022. [Online]. Available: https://debezium.io/documentation/reference/2.0/.

[22] Debezium, „Connectors," Debezium, 2022. [Online]. Available: https://debezium.io/documentation/reference/stable/connectors/index.html. [Hozzáférés dátuma: 31 október 2022].

[23] Apache, „DOCUMENTATION," Apache, 2022. [Online]. Available: https://kafka.apache.org/documentation/.

[24] Confluent, „Kafka Clients," Confluent, [Online]. Available: https://docs.confluent.io/platform/current/clients/index.html#other-languages. [Hozzáférés dátuma: 31 október 2022].

[25] MongoDB, Inc., „Change Streams," MongoDB, Inc., 2022. [Online]. Available: https://www.mongodb.com/docs/manual/changeStreams/. [Hozzáférés dátuma: 31 október 2022].

[26] Microsoft, „ASP.NET documentation," Microsoft, [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-6.0. [Hozzáférés dátuma: 27 október 2022].

[27] Microsoft, „Dependency injection in ASP.NET Core," Microsoft, 6 március 2022. [Online]. Available: https://learn.microsoft.com/en-

us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0.

[28] Hangfire, „Documentation," Hangfire, [Online]. Available: https://docs.hangfire.io/en/latest/. [Hozzáférés dátuma: 1 november 2022].

[29] Hangfire, „Performing recurrent tasks¶," Hangfire, 2022. [Online]. Available: https://docs.hangfire.io/en/latest/background-methods/performing-recurrent-tasks.html?highlight=recurring. [Hozzáférés dátuma: 1 november 2022].

[30] Confluent, Inc. 2014- 2022, „Consumer," Confluent, Inc. 2014-2022, 2022. [Online]. Available: https://docs.confluent.io/kafka-clients/dotnet/current/overview.html#consumer. [Hozzáférés dátuma: 1 november 2022].

[31] A. Yakunin, „Fusion Overview," 27 október 2020. [Online]. Available: https://github.com/servicetitan/Stl.Fusion/blob/01d32cc3df803383b428bc05543d0f574e2ffd95/docs/Overview.md. [Hozzáférés dátuma: 1 november 2022].

[32] A. Yakunin, „Fusion Overview," 27 október 2020. [Online]. Available: https://github.com/servicetitan/Stl.Fusion/blob/01d32cc3df803383b428bc05543d0f574e2ffd95/docs/Overview.md. [Hozzáférés dátuma: 1 november 2022].

[33] A. Yaukin és T. Princz, „Part 11: Authentication in Fusion," Github, 11 július 2022. [Online]. Available: https://github.com/servicetitan/Stl.Fusion.Samples/blob/master/docs/tutorial/Part11.md.

[34] Facebook , „In-Depth Overview," Facebook , 3 június 2022. [Online]. Available: https://facebook.github.io/flux/docs/in-depth-overview/.

[35] D. Abramov, „Core Concepts,” 25 június 2021. [Online]. Available: https://redux.js.org/introduction/core-concepts.

[36] C. Jackson, „Micro Frontends,” 19 június 2019. [Online]. Available: https://martinfowler.com/articles/micro-frontends.html.

[37] Microsoft, „Code metrics values,” Microsoft, 21 október 2022. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022. [Hozzáférés dátuma: 1 november 2022].

[38] A. Yakunin, „[Swap] attribute,” 30 szeptember 2022. [Online]. Available: https://github.com/servicetitan/Stl.Fusion.Samples/blob/master/docs/tutorial/Part05.md#swap-attribute. [Hozzáférés dátuma: 1 november 2022].

[39] Microsoft, „https://learn.microsoft.com/en-us/azure/hdinsight/kafka/apache-kafka-introduction,” Microsoft, 30 március 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/hdinsight/kafka/apache-kafka-introduction. [Hozzáférés dátuma: 1 november 2022].

[40] Microsoft, „Event Sourcing pattern,” Microsoft, [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing. [Hozzáférés dátuma: 1 november 2022].

[41] Microsoft, „CQRS pattern,” Microsoft, [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs. [Hozzáférés dátuma: 1 november 2022].

[42] M. Chtioui, „Medium,” 21 augusztus 2019. [Online]. Available: https://medium.com/@mahdichtioui/reactivex-reactive-programming-principles-dbb1bafa8384.

[43] J. Ekanayake, „3 Methods for Implementing Change Data Capture,” Analytics Vidhya, 30 július 2021. [Online]. Available:

https://www.analyticsvidhya.com/blog/2021/07/3-methods-for-implementing-change-data-capture/.

[44] Microsoft, „What is Apache Kafka in Azure HDInsight," Microsoft, 30 március 2022. [Online]. Available: What is Apache Kafka in Azure HDInsight. [Hozzáférés dátuma: 1 november 2022].

[45] A. Lock, „Enabling prerendering for Blazor WebAssembly apps," 19 január 2021. [Online]. Available: https://andrewlock.net/enabling-prerendering-for-blazor-webassembly-apps/. [Hozzáférés dátuma: 1 november 2022].

Függelék

6.1 docker-compose.yaml

```yaml
version: '3.7'

services:
  zookeeper:
    hostname: zookeeper
    container_name: zookeeper
    image: debezium/zookeeper:latest
    ports:
     - 2181:2181

  kafka:
    hostname: kafka
    container_name: kafka
    image: debezium/kafka:latest
    ports:
     - "29092:29092"
     - "9092:9092"
     - "9101:9101"
    links:
     - zookeeper
    environment:
     - ZOOKEEPER_CONNECT=zookeeper:2181
     - KAFKA_LISTENER_SECURITY_PROTOCOL_MAP=
       PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
     - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://kafka:29092,
       PLAINTEXT_HOST://localhost:9092
     - KAFKA_LISTENERS=PLAINTEXT://:29092,PLAINTEXT_HOST://:9092

   connect:
    hostname: connect
    container_name: connect
    image: debezium/connect:latest
    ports:
     - 8083:8083
    links:
     - kafka
     - sqlserver
    environment:
     - BOOTSTRAP_SERVERS=kafka:29092
     - GROUP_ID=1

  kafka-ui:
    image: provectuslabs/kafka-ui
```

```
    container_name: kafka-ui
    ports:
      - "8080:8080"
    links:
     - zookeeper
     - kafka
    restart: always
    environment:
      - KAFKA_CLUSTERS_0_NAME=local
      - KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS=kafka:29092
      - KAFKA_CLUSTERS_0_ZOOKEEPER=zookeeper:2181

  sqlserver:
    hostname: sqlserver
    container_name: sqlserver
    image: mcr.microsoft.com/mssql/server:2019-latest
    ports:
     - 3033:1433
    environment:
     - ACCEPT_EULA=Y
     - MSSQL_PID=Standard
     - MSSQL_SA_PASSWORD=Password123!
     - MSSQL_AGENT_ENABLED=true
    volumes:
      - sqlData:/var/opt/mssql

volumes:
  sqlData:
    external: false
```