Eötvös Loránd Tudományegyetem

Informatikai Kar

Numerikus Analízis

Tanszék

Delayed resource lock service

Szerző:
Fekete Márk
programtervező informatikus BSc

Belső témavezető:
Dr. Csörgő István
egyetemi adjunktus

Külső témavezető:
Repiczki Zoltán
okleveles mérnökinformatikus

Budapest, 2023

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:
Név: Fekete Márk
Neptun kód: E0VHZ5

Képzési adatok:
Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)
Tagozat : Nappali

Külső témavezetővel rendelkezem

Külső témavezető neve: *Repiczki Zoltán*
munkahelyének neve: ***Leviathan Solutions kft.***
munkahelyének címe: ***Budapest, Ábel Jenő u. 23, 1113***
beosztás és iskolai végzettsége: ***ügyvezető, okl. mérnök informatikus MSc***
e-mail címe: ***repiczki.zoltan@leviathan.hu***

Belső konzulens neve: *Csörgő István*
munkahelyének neve, tanszéke: ***ELTE-IK, Numerikus Analízis Tanszék***
munkahelyének címe: ***1117, Budapest, Pázmány Péter sétány 1/C.***
beosztás és iskolai végzettsége: ***egyetemi adjunktus***

A szakdolgozat címe: DelayedResourceLockService

A szakdolgozat témája:
(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

Szakdolgozatomban egy injektálható singleton szolgáltatást fogok megvalósítani, amely egy komoly vállalati környezetben erőforrások dinamikus lefoglalását teszi lehetővé, végfelhasználók számára kényelmesen használható módon. A projektet C++ nyelven Qt keretrendszerben fogom elkészíteni, valamint felhasználom a céges belső keretrendszerünk nyújtotta kódbázist.
A szolgáltatás képes lesz az erőforrások lefoglalására, a foglalt erőforrásokon a főszáltól elkülönítve végezhetünk írási, olvasási valamint adatfeldolgozó műveleteket, egy zár lehet írási, vagy olvasási ezzel minimalizálva a konkurens helyzetek kialakulását, a megfelelő zárak applikálása után úgynevezett aszinkron műveletek hajtódnak végre majd az előzőleg lefoglalt erőforrások fel is szabadulnak rendre.
Amennyiben a lefoglalandó erőforrások egy részhalmazát már birtokolja valamilyen a szoftverben futó másik folyamat, úgy a lefoglalandó erőforrások, valamint a sikeres lefoglalásukkor futtatandó aszinkron folyamat eltárolódik, egy kimondottan erre a feladatra kialakított struktúra elemtípusú, várakoztatási sorba.
Az erőforrások felszabadulásakor elindul egy mechanizmus amely a sorban várakozó aszinkron folyamatok, és a hozzájuk tartozó erőforrásokat figyeli. Ha a folyamathoz tartozó összes erőforrás elérhető, a folyamat lefut, felszabadulnak a lefoglalt erőforrások, a várakozási sorból töröljük az érintett elemet.

Budapest, 2023. 02. 28.

Tartalomjegyzék

1. fejezet

Bevezetés

A szakdolgozatom témáján sokat gondolkoztam. Nem szerettem volna valami sablonos, haszontalan dolgot csinálni, ezért a munkahelyen, ahol dolgoztam ebben az időben beszélgettem a szakdolgozat témaválasztásának nehézségeiről az egyik kiemelkedően jó fejlesztőnkkel, aki akkoriban felettesem és szakmai mentorom volt. Vele átnéztük az egyik termékünket strukturálisan, kielemeztük, hogy milyen hiányosságai vannak, amikre lehetne kész komponens szinten megoldást kialakítani a jelenleg létező, de kiforratlan kezdetleges eszközök, vagy hiányosságok helyett. A projekt struktúrája tartalmazott egy Object Relation Model-t megvalósító belső keretrendszert, ennek segítségével az adatbázisaink rekordjait, tábláit és relációit képeztük le a C++[1] nyelvnek létező nyelvi elemeire, ami ezekben az esetekben osztályok objektumpéldányai voltak.

A projekt strukturális felépítéséhez továbbá hozzá tartozott egy aszinkronitást támogató többszálúsítást lehetővé tevő eszköz, amivel taszkokat hozhattunk létre és azokra callback-ekkel iratkozhattunk fel. Az Object Relation Model segítségével létrejövő úgynevezett entitások módosítása innentől kezdve a többszálúsítás hatására veszélyessé vált, ami miatt szükség volt egy eszközre, amin keresztül ezen műveletek szinkronizálása, kölcsönös kizárása megvalósítható. Erre a problémára keretrendszer szinten működő megoldás egy erőforrás zároló rendszer, ami lehetővé teszi ezek biztonságos elvégzését. Ez egy létező eszköz volt a projektben ResourceLockService néven, mely kényelmes és biztonságos módja volt az adatbázisentitásaink kezelésének, viszont bizonyos feladatok elvégzésére nem volt képes.

Ennek a problémakörnek a körbejárása során bukkantunk egy korábban már felmerülő, viszont elhanyagolt igényre, ami azt fogalmazta meg, hogy: Legyen lehe-

tőség erőforrások zárolására aszinkron módon, késleltetett végrehajtással ellátva. Az eszköz kapjon egy halmaznyi erőforrást, amit zárol és egy feladatot, amit elvégez. Amennyiben az erőforrások elérhetőek minden gond nélkül lefoglalja őket majd teljesíti a feladatot, amit rá bíztunk. Ha az erőforrások nem elérhetőek akkor a feladatot és az erőforrások halmazát tárolja el, figyelje a felszabaduló erőforrásokat, majd, ha a feladat elvégzéséhez szükséges entitások felszabadulnak, már nem tart fent rájuk semmilyen másik komponens zárat, akkor foglalja le azokat, és végezze el a korábban kapott elvégzendő feladatot. Később felmerülő koncepcionális bővítés keretein belül újabb funkcionalitást is kapott, amely egy időtúllépési limit plusz paramétert jelentett, illetve egy időtúllépés esetén végrehajtandó feladatot, amely szintén plusz paraméterként jelentkezett az eszközben, ezzel növelve annak komplexitását, használhatóságát és értékét a projekt számára. Az elkészítése sok időt energiát és gondolkodást vett igénybe. Az eszköz elkészítését többször kellett újrakezdeni bizonyos rajtam kívül eső okokból, de a kitartás és a szakmai mentorom segítsége a kardinális kérdések eldöntésében meghozta az eredményét és a komponens elkészült.

A munkámra büszke vagyok, és arra is, hogy hasznos részét képezi az egyik igen mély és komplex éles informatikai rendszernek az iparban. Szakdolgozatomban ennek a komponensnek a kipróbáló programját valósítom meg, illetve saját implementációt készítek az ORM[1] réteghez mivel az túlságosan nagy komponens lett volna a projektből való átemeléshez. A projekt C++ nyelven annak a 20-as szabványában készítem el, a Qt[2] GUI[2] keretrendszer használatával.

[1]Object Relation Model
[2]Graphical User Interface

2. fejezet

Felhasználói dokumentáció

A felhasználói dokumentáció során először írni fogok a kipróbálóprogram használatáról, melyben röviden bemutatom a funkcionalitását és hogy hogyan működik. Ez után mivel a komponens felhasználói is programozók, látható lesz egy rövid összefoglaló a publikus API[1] -ról és annak helyes használatáról. Ezt szükségesnek érzem mivel egy jó API elkészítése külön kihívást jelentő feladat. Akkor jó egy API, ha azt lehetetlen rosszul használni, tartja a mondás. Ez sajnos az életben nem mindig kivitelezhető, ezért érdemes az API-jainkhoz megfelelő dokumentációt készíteni.

2.1. Teszt program

A tesztprogram működése a lehető legegyszerűbben hivatott szemléltetni az alkalmazás funkcionalitását, ebből kifolyólag a minimalista dizájn mellett döntöttem és egy számomra nagyon emlékezetes adatbázis sémán dolgozok in memory[2] módon, ez pedig nem más, mint az Adatbázisok 1 során megismert szeret tábla. A tábla tartalmaz Micimackó szereplőket és gyümölcsöket, a köztük lévő reláció pedig jelképezi, hogy a mesefigura szereti-e az adott gyümölcsöt. Ez az egyszerűsége miatt reményem szerint mindenki számára érthető módon szemlélteti majd a háttérben lezajló komplex folyamatokat.

A tesztprogram felhasználói interfésze a következőképpen épül fel:

- Fő ablak
  - Felhasználók tab

[1] Application Programming Interface
[2] Teljes adatbázis leképezés memóriában

  - Gyümölcsök tab

- Felhasználó – Gyümölcs reláció kezelő dialógus

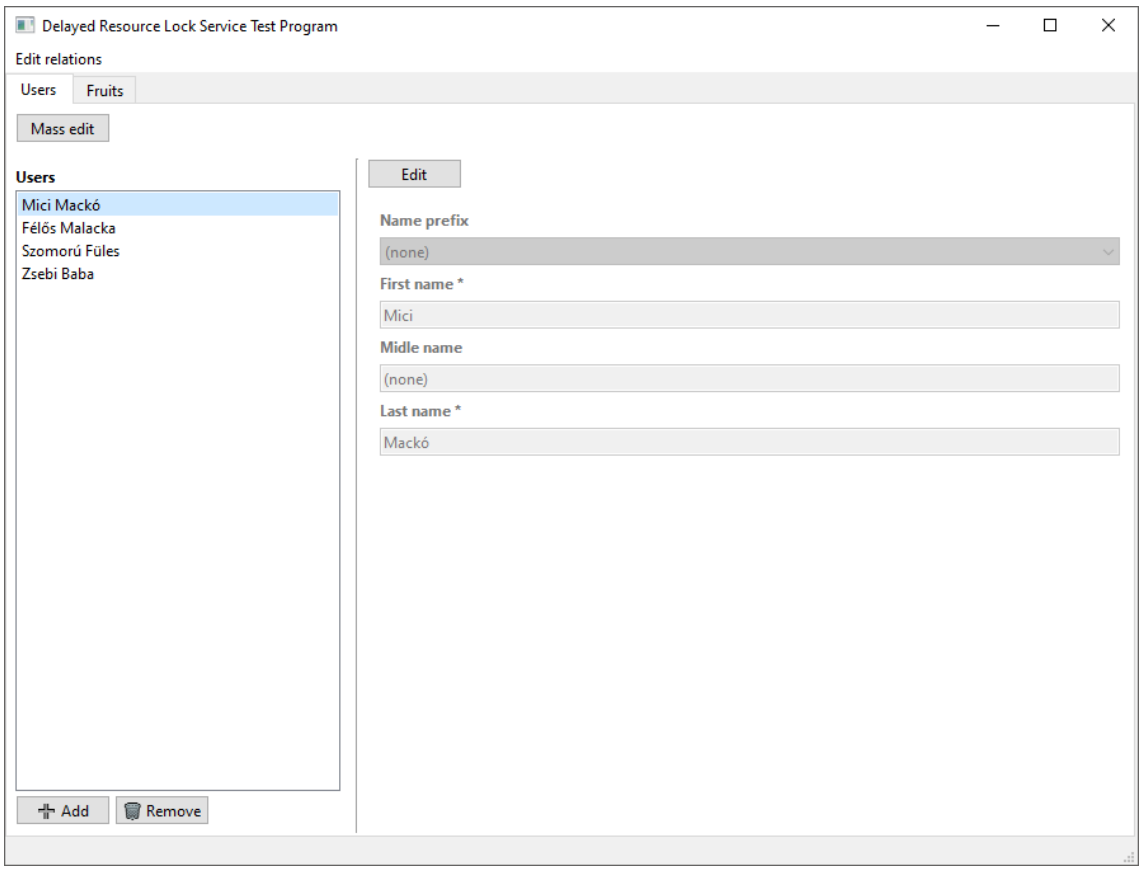
A következő részben az egyes komponensek részletezése következik.

2.1.1. Hordozható alkalmazás

Az alkalmazás portable módon próbálható ki, ami azt jelenti, hogy a futtatható állományt nem szükséges telepíteni a számítógépekre.

2.1.2. Fő ablak

A fő ablak tartalmazza az összes komponensét az alkalmazásunknak, amellyel interakcióba léphetünk. Felül a menüsávban az Edit relations menü elem segítségével szerkeszthetjük meg a kapcsolatainkat. Ez a dialógus résznél tovább lesz részletezve. Lejjebb az alkalmazás 2 fő tabból épül fel, amik között válthatunk megszorítások nélkül balegérgombbal történő kattintás segítségével.

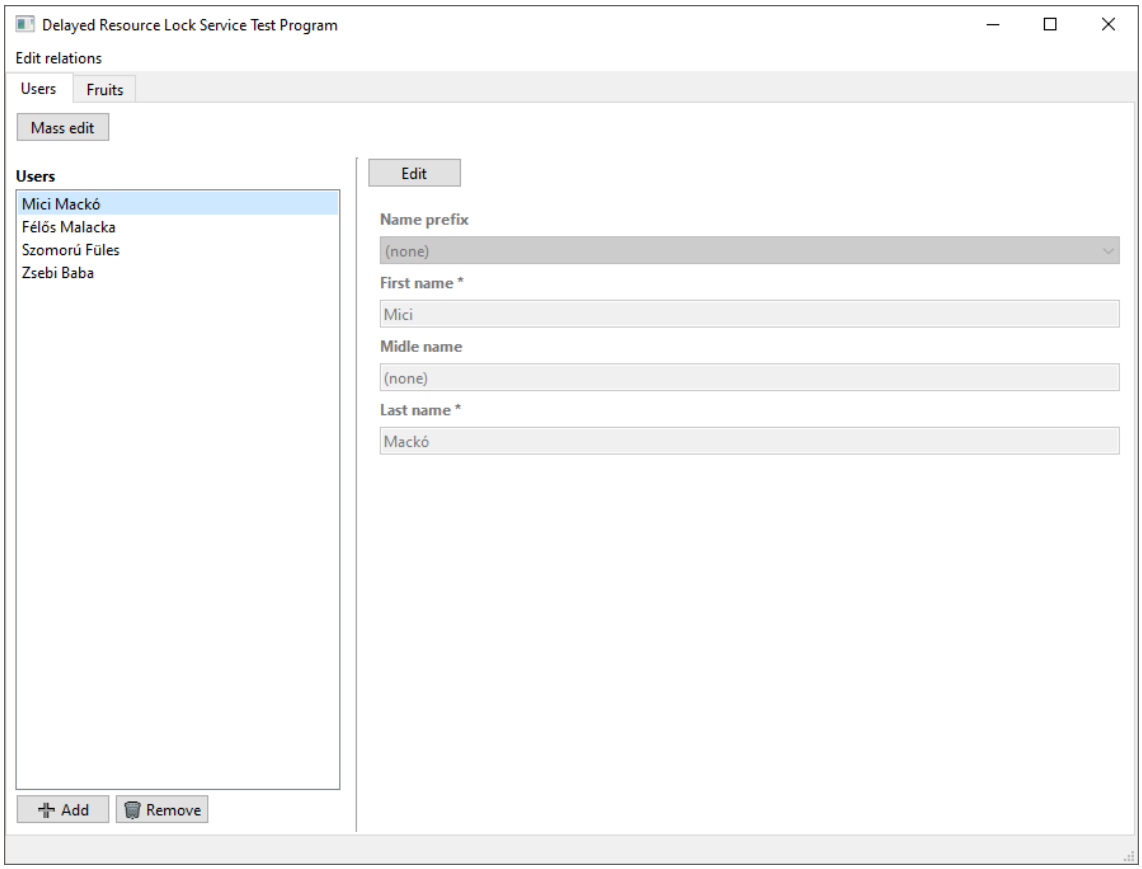


2.1. ábra. Fő ablak

2.1.3. Felhasználók tab

A felhasználók tab egy úgynevezett master detail nézetű szerkesztő felületet valósít meg a felhasználók felett. A tab bal oldalán található egy listás megjelenítés, amiben a felhasználók nevei olvashatóak. A lista alatt található Add gombbal adhatunk hozzá új felhasználót a rendszerhez ez alapértelmezetten "New User" néven jön létre. Ugyancsak a lista alatt látható egy Remove feliratú gomb is, aminek a megnyomása a kiválasztott listaelem és azzal együtt a felhasználó törlését eredményezi.

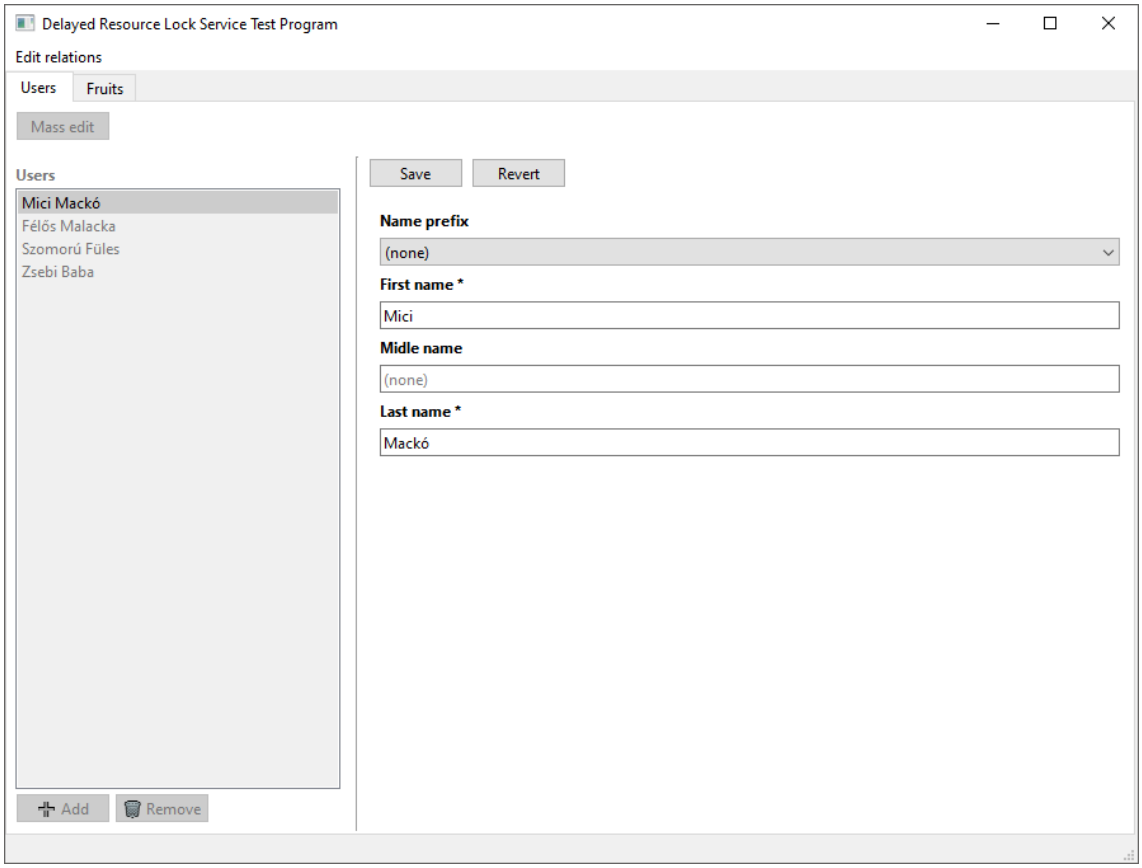
A táblázatban lévő elemek kiválasztásával a details részen látható űrlap komponensei kitöltődnek a felhasználóra jellemző tulajdonságokkal, ez esetünkben a Név előtagja (Name prefix), Keresztnév (First name), Középső név (Middle name) és Vezetéknév (Last name) mezők. Ezek a komponensek alapból letiltott módban vannak, kijelölésük, szerkesztésük nem lehetséges.



2.2. ábra. Felhasználók tab

A szerkesztésre is van természetesen lehetőség, kettő féleképpen is. Az egyik mód a közvetlenül a mezők felett elhelyezett Edit gomb megnyomásával lehetséges. Ez

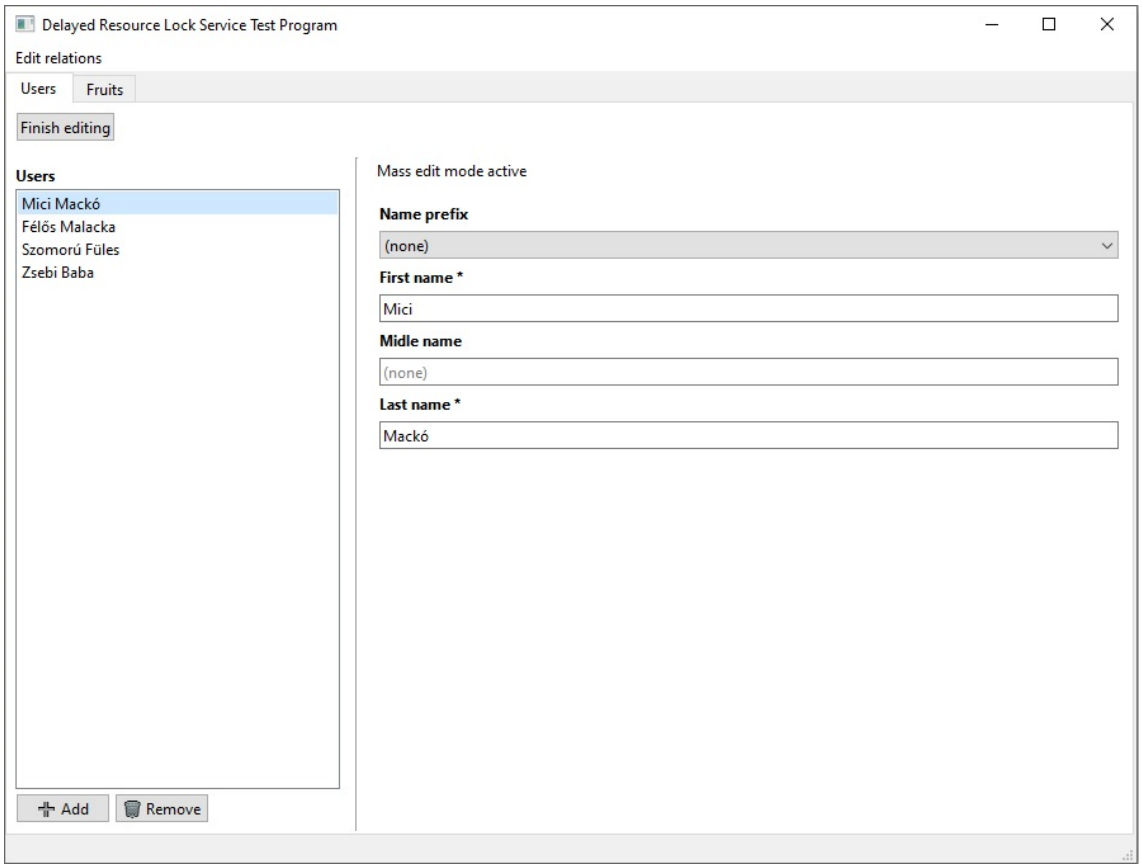
az alkalmazásunkat az úgynevezett Single Edit módba helyezi. Ebben az állapotban lehetőségünk van az általunk kiválasztott elemet szerkeszteni, de csak azt. Az Edit gomb helyett megjelennek az ehhez a módhoz tartozó specifikus gombok a Save és a Revert. A szerkesztés végeztével ezek segítségével menthetjük el vagy állíthatjuk vissza a szerkesztett felhasználót korábbi állapotába, ha nem szeretnénk megtartani a változtatásokat.



2.3. ábra. Felhasználók tab szerkesztés alatt

A másik lehetőség a tab bal felső sarkába elhelyezett Mass edit gomb megnyomásával érhető el. Ennek segítségével az alkalmazásunk a tömeges szerkesztés állapotba kerül. Ebben az állapotban váltogathatunk a felhasználók között, változtathatunk az attribútumokon, hozzáadhatunk és törölhetünk felhasználót. Az előbb említett felhasználói felületen végzett módosulások azonnal fejtik ki hatásukat, nem szükséges a szerkesztési módból kilépnünk. A tömeges szerkesztés befejezéséhez a bal felső Mass edit gomb helyén megjelenő Finish editing gombbal léphetünk ki. A tömeges szerkesztés módban a beviteli mezők felett közvetlenül elhelyezkedő Edit gomb nem jelenik meg, helyette tájékoztató szöveg olvasható arról, hogy miért nem látható a

gomb, ennek az oka természetesen az, hogy egy másik szerkesztési mód jelenleg is aktív.



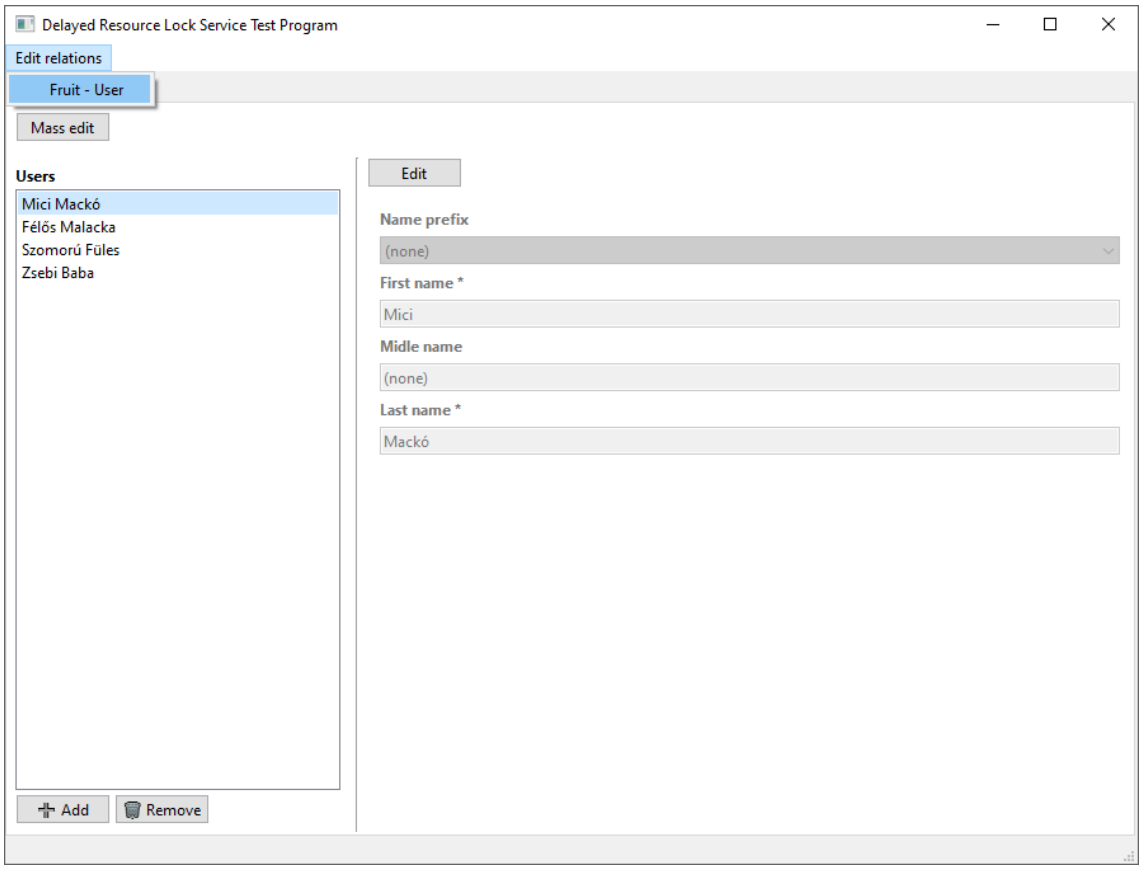
2.4. ábra. Felhasználók tab szerkesztés alatt

2.1.4. Gyümölcsök tab

A gyümölcsök tab a felhasználók mintájára készült el, így a működések teljes mértékben megegyeznek a korábbiakban leírtakkal. Az egyetlen főbb különbség, hogy itt a gyümölcsöknek csupán név tulajdonságuk van így annak a változtatásai érvényesülnek a korábbi 4 megemlített helyett.

2.1.5. Felhasználó - Gyümölcs relációk szerkesztése

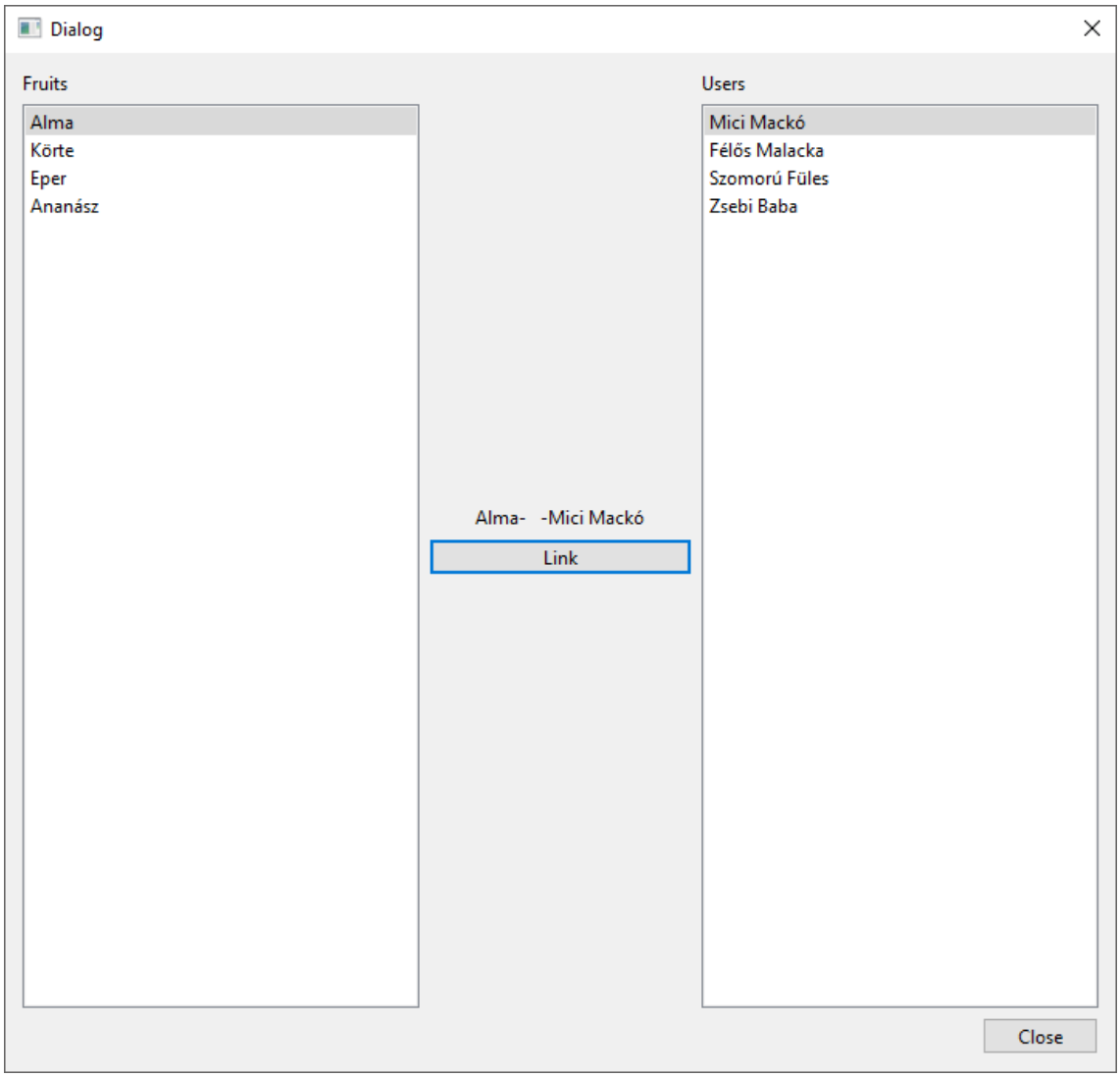
A főoldal bal felső sarkában található Edit relations lenyíló menüben található egyetlen elem a Fruit - User elem.



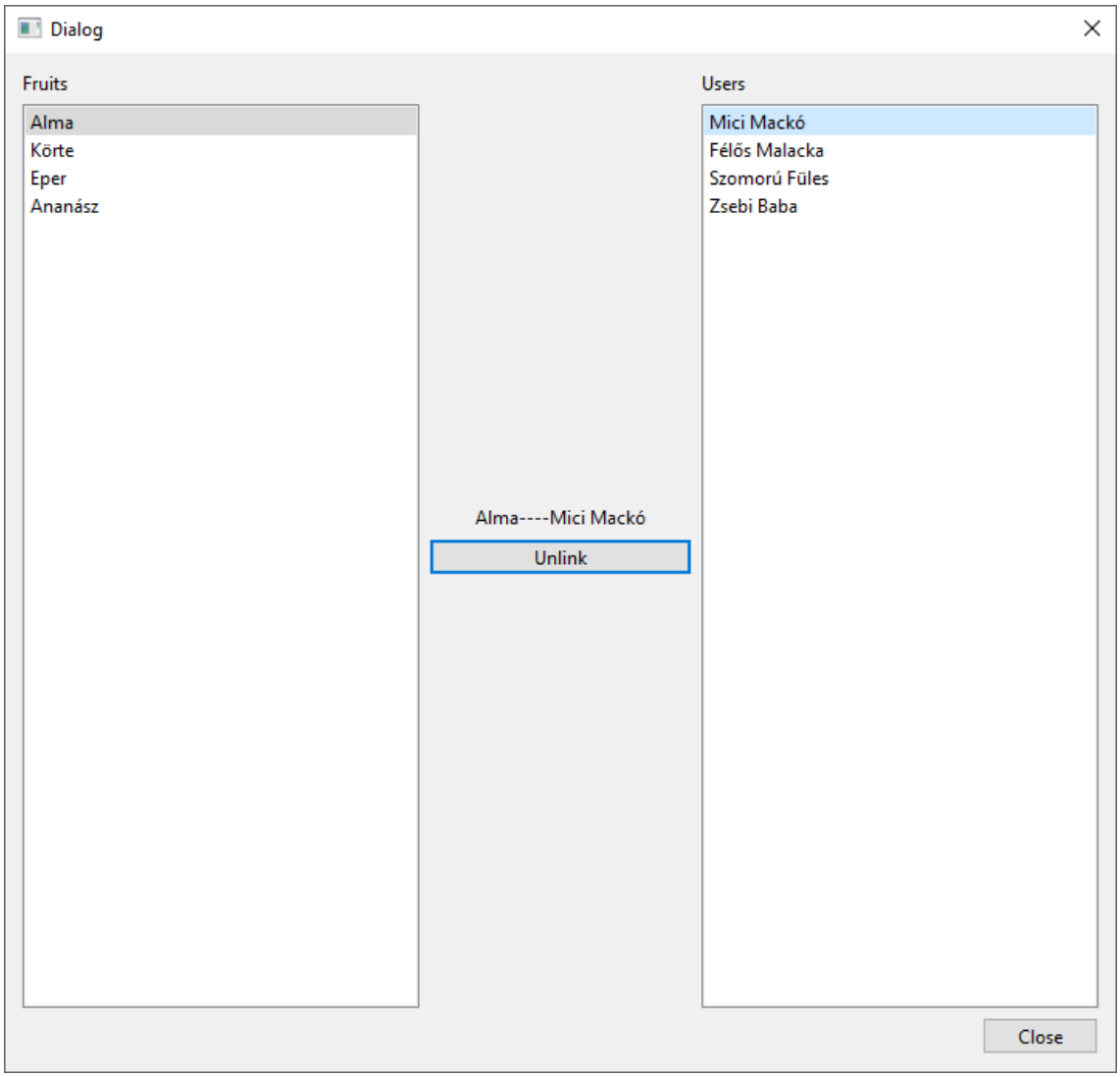
2.5. ábra. Fruit - User menü elem

Ennek megnyomásával lehetőségünk nyílik a gyümölcsök és felhasználók közti relációk szerkesztésére, amit egy dialóguson keresztül tehetünk meg ebben a dialógusban kettő lista található az oldal bal és jobb oldalán. Az egyik a felhasználóinkat tartalmazza nevük alapján beazonosítható módon, míg a másik hasonlóképpen név szerint a gyümölcsöket tartalmazza. A két listából történő egy-egy elem kiválasztása után a két lista között elérhetővé válik a link vagy az unlink opció annak függvényében, hogy a két általunk választott elem, amelyből az egyik a felhasználó, a másik pedig természetesen a gyümölcs. Ez a gomb állapotában leköveti a kiválasztott elemeink státuszát. Amennyiben az elemek között létezik reláció, ami azt reprezentálja, hogy a felhasználó szereti a gyümölcsöt a gomb szövege Unlinkre vált, ami jelöli, hogy a lenyomásával ez a kapcsolat meg fog szűnni. Ennek mintájára, ha a két elem között nem létezik kapcsolat a gomb szövege Link-re vált, amelynek lenyomása kialakítja az elemek közti relációt, szellemesen mondhatjuk, hogy a felhasználóval megszeretteti a gyümölcsöt. A szeretetet reprezentáló reláció meglétére vagy meg nem létére egy másik komponens is felhívja a figyelmünket, ez pedig egy egyszerű szöveges jelölés, ami, a két elem közé egy összeköttetést rak, ha azok relációban állnak egymással,

ha viszont nem akkor ezt az összeköttetést a közepénél egy látványos szakadással látjuk megjelenni.



2.6. ábra. Reláció szerkesztése - Összekapcsolás



2.7. ábra. Reláció szerkesztése - Szétkapcsolás

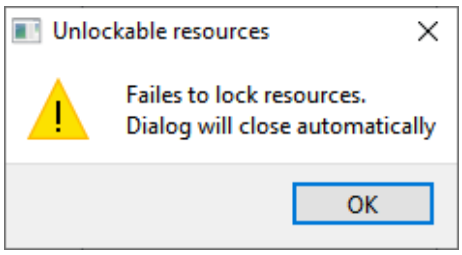
2.2. Egymást kizáró műveletek

A korábbi szekcióban már említés szinten volt szó a szerkesztés módjairól. Ezeknek az elsőre talán feleslegesnek tűnő műveleteknek az indoklása fog a lentebbi részben következni. A háttérben lévő entitások integritása elengedhetetlen az alkalmazás helyes működésének szempontjából. Kimondottan súlyos programozói hiba lenne, ha az entitások különböző helyekről esetleg különböző logikai szálakból szerkeszthetőek lennének. A program működése kiszámíthatatlanná válna, Inkonzisztencia lépne fel, más helyekről mást látnánk ugyan arról az entitásról. Ennek megelőzése érdekében tekintsünk az entitásainkra úgy, mint erőforrásokra. Az erőforrásainkat kettő féleképpen lehet lefoglalni, írási vagy olvasási preferenciával. A konkurens olvasás megengedett, az írás nem megengedett. Az alkalmazásban

explicit módja az entitások erőforrás jellegű lefoglalásának az edit gombok használata. Szerkesztés közben fenntartjuk a zárat, más helyről az azonos entitások nem szerkeszthetőek.

Emellett az alkalmazásban lévő relációk szerkesztése dialógus is, viszont ez implicit módon teszi, zárakat hoz létre az általa szerkesztett entitásokra.

A kipróbáló programunkban lehetőségünk is van ezek megtapasztalására abban az esetben, ha szerkesztés módban nyitjuk meg a relációk szerkesztése dialógusunkat. Ekkor az edit mód által lefoglalt entitásainkat olvasási preferenciájú zárát próbálja meg újra lefoglalni a dialógus, ami nem fog sikerülni mivel ezek nem jöhetnek létre konkurens módon. Az alkalmazásunk ennek sikertelenségéről egy hibaüzenet formájában tájékoztat, amiben leírja, hogy az erőforrások már más birtokában vannak.



2.8. ábra. Zárolási hibaüzenet dialógus megnyitása esetén

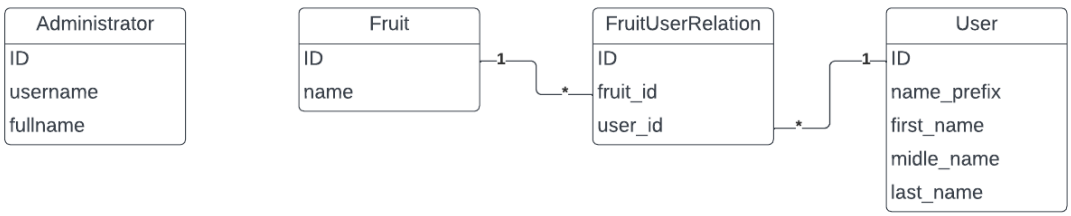
3. fejezet

Fejlesztői dokumentáció

Az általam írt céges kódbázis átemelése során szembesültem vele, hogy a függőségeim nagyon mélyre nyúlnak a céges kódbázisban, amit természetesen nem szerettem volna egy az egyben átemelni. A megoldást abban láttam, hogy a függőségekhez saját implementációkat készítek, amik, ha kevésbé komplex módon is de lemodellezik a működést, ami az általam írt kód számára elengedhetetlen a működéshez. A következőkben ezeket a részeket fogom mélyebben bemutatni aztán rátérek a delayed resource lock service bemutatására.

3.1. ORM által leképezett adatbázis séma

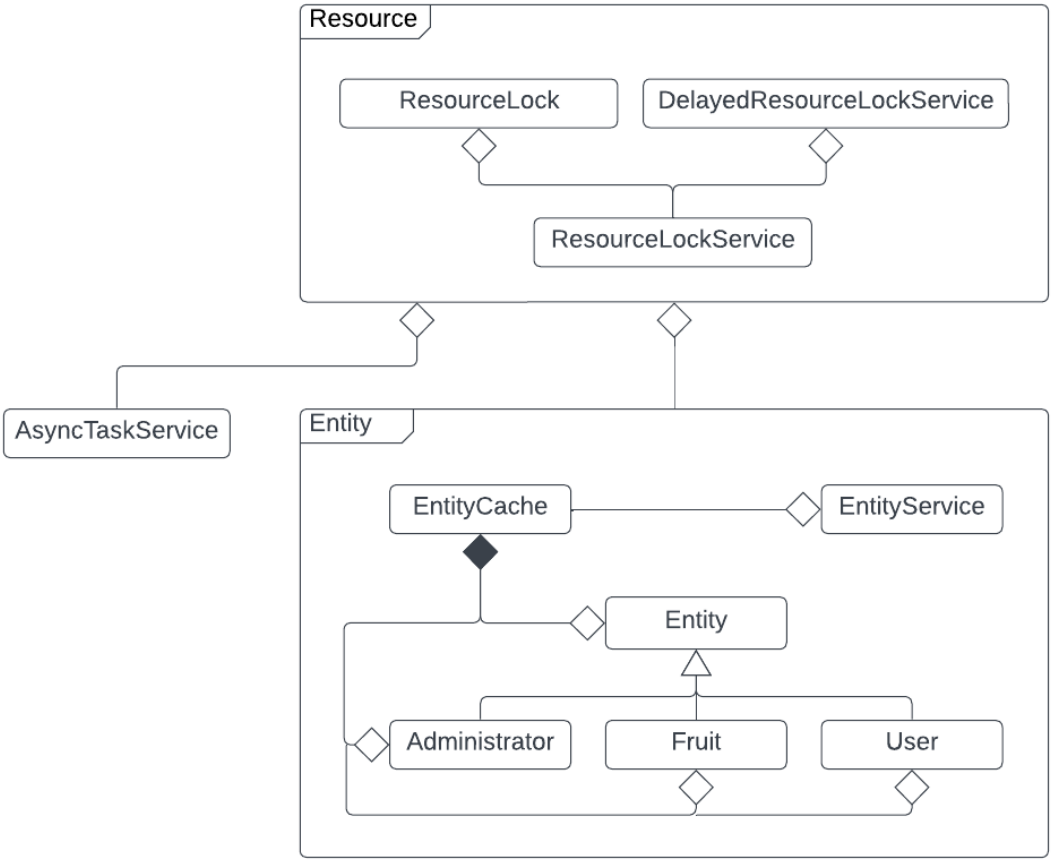
Annak ellenére, hogy valódi adatbázis nincs a projekt mögött, azt ORM réteg által leírt adatbázist érdemes bemutatni. A projekt adatbázissémáját a következő ábra illusztrálja.



3.1. ábra. Adatbázis séma diagram

3.2. Projekt egyszerűsített osztálydiagramja

A projekt megannyi komponensének a függőségek szemléltetése szempontjából érdemes lehet UML-szerűen ábrázolni, a későbbiekben pedig külön-külön lesznek részletezve az egyes komponensek.



3.2. ábra. Egyszerűsített osztálydiagram

3.3. Entitás réteg

Az entitás réteg szerepe, hogy a programban a nyers SQL írás helyett C++ objektumokkal lehessen típusos módon leírni az adatbázisban végzendő műveleteket. A nyers SQL írás során felmerülő hibák nem derülnek ki csak futásiidőben, emiatt az egyik lényegi tulajdonsága nincs kihasználva a nyelvnek, ami a fordítási időben kiderülő hibák felderítésének lehetősége. Az entitásréteg segítségével a futásidejű hibák átkerülnek a fordítási időbe, ezért a termék nem kerülhet a felhasználók elé olyan állapotban, ami nem felel meg egy stabil elvárható sztenderdnek.

3.3.1. Entitások

Az entitások olyan adatbázisrekordot leképező C++ objektumok, amelyek tartalmazzák az adatbázisrekord mezőit erősen típusos módon privát tagváltozóiban, ehhez pedig publikus getter és szetter van definiálva, hogy kényelmesen lehessen használni. Ezeket nevezzük triviális getternek és triviális szetternek.

Ezek mellett vannak esetenként nem triviális getterek. Erre egy jó példa a felhasználók esetében, a teljes név lekérdezésére létrehozott getter. Ez a tulajdonság egy úgynevezett számított mezője az entitásnak. Ez azt jelenti, hogy adatbázisrekord szinten, illetve entitáson belüli privát adattag szinten sem létezik hozzá mező, viszont másik mezők alapján valamilyen módon megalkotható az adat, aminek a lekérdezése a triviális getterekével azonos szintaktikával történik. Ezekhez a nem triviális getterekhez nem létezik szetter mivel ezeknek nem lenne értelme.

További fontos művelete egy entitásnak, hogy törölhető legyen szükség szerint. Erre a feladatra alkalmas egy paraméter nélküli közvetlenül az entitáson hívható remove művelet, ami eltávolítja azt.

Az entitások fontos része még, hogy rajtuk keresztül lehessen kezelni a hozzájuk kötődő más entitásokkal közös relációikat. Természetesen erre is van lehetőség és az erre kialakított eszközök a reláció típusától függ ezzel is szem előtt tartva a fordítási idejű hibák fontosságát. A relációk esetében megkülönböztetünk 3 fajtát abból a szempontból, hogy egy entitás hány másik entitással állhat kapcsolatban. Ezek lehetnek:

- One to One (Egy az Egyhez)
- One to Many (Egy a Többhöz)
- Many to Many (Több a Többhöz)

Mivel az entitáson keresztül lehet módosítani azt, hogy mik a vele kapcsolatban lévő entitások és egyszerre csak egy entitással tudunk dolgozni, ezért szintaktikailag elég kétféleképpen különválasztani az eseteinket, a Many to Many sosem fog megjelenni kód szinten.

A One to One esetben az entitások közti reláció is getter és szetter szinten valósul meg szintaktikailag. Ha egy entitáshoz hozzá akarunk rendelni egy másikat csak odaadjuk a szetterének és az a háttérben felépíti a relációt köztük. A getter visszaadja a relación keresztül az entitást akárcsak egy triviális getter tenné azt egy adattag esetében. Illetve ezek mellett még szükség van az eltávolítás lehetőségére is,

ezt akár egy paraméter nélküli remove függvény készítésével is elérhetjük. Ezzel az implementációval le is fedtük ezt az esetet.

A One To Many esetben a szettert felváltja egy add művelet, amivel a már meglévő relációkon túl egy újabbat hozhatunk létre, nem szüntetve meg és nem módosítva a meglévő relációink állapotát. A getterünk ebben az esetben nem pontosan egy darab entitást ad vissza, hanem egy listányit, mivel a több közül nem tudunk válogatni, nincs egy kimondottan jó elem, ezért az alapelv, hogy adjuk vissza az összeset. A relációk megszüntetésére itt két módszert is lehet használni. Az egyik a már korábban megismert remove függvény paraméterezhető verziója, hogy tudja az entitás, hogy a sok közül pontosan melyik entitáshoz kötődő relációját akarjuk lebontani. A másik módja ennek a clear függvény lesz, aminek segítségével minden relációt fel tudunk számolni paraméter átadás nélkül. Nézzünk meg erre egy példát publikus inteface kialakítás szintjén.

```
class User
    : public Entity
    , public std::enable_shared_from_this<User>
{
    ...

    std::optional<QString> getNamePrefix() const;
    QString getFirstName() const;
    std::optional<QString> getMidleName() const;
    QString getLastName() const;

    // non trivial getters
    QString getFullName() const;

    std::shared_ptr<User> setNamePrefix(const std::optional<QString
        > &namePrefix);
    std::shared_ptr<User> setFirstName(const QString& firstName);
    std::shared_ptr<User> setMidleName(const std::optional<QString>
        &midleName);
    std::shared_ptr<User> setLastName(const QString& lastName);

    void remove() override;

    // relations
    const QList<std::shared_ptr<Fruit>> getFruits() const;
```

```
24
25      std::shared_ptr<User> clearFruits();
26      std::shared_ptr<User> addFruit(std::shared_ptr<db::Fruit>
            fruitToAdd);
27      std::shared_ptr<User> removeFruit(std::shared_ptr<db::Fruit>
            fruitToRemove);
28
29  private:
30      std::optional<QString> namePrefix_;
31      QString firstName_;
32      std::optional<QString> midleName_;
33      QString lastName_;
34
35  private:
36      static int nextId_;
37      static constexpr EntityType entityType_ = EntityType::User;
38  };
```

3.1. forráskód. User Entitás header file

A fentebb lévő kódrészletben jól elkülöníthetőek a korábban említett részek. Az osztálydefiníciót követően a 6. sortól a getterek, a 12. sorban a nem triviális getterek, a 14. sorban a szetterek, a 22. sortól a Many to Many gyümölcsökkel való relációt menedzselő eljárások, végül pedig, bár ez nem a publikus része az osztálynak, a member változóink láthatóak. A következő komponens bemutatása előtt figyeljük meg, hogy az entitások az Entity ősosztályból származnak le, ahogy az a 2. sorban is látszik. Ennek a ténynek az ismerete később még hasznos lesz. Az ok egyértelműen a futásiidejű polimorfizmus, aminek a segítségével nagy mennyiségű kód duplikáció elkerülése vált megvalósíthatóvá, illetve az entitások polimorf módon történő tárolása is így került megvalósításra, parametrikus polimorfizmus[1] vagy type erasure[2] helyett.

3.3.2. EntityService

Az entitások amellett, hogy képesek önállóan megannyi művelet elvégzésére, nem tudnak lefedni minden felhasználói esetet teljes mértékben. Az előbbiekben például szembe tűnhetett, hogy a létrehozásukra nem volt egyszerű konstruktor definiálva.

[1]Programozási technika, amely során egy típusparamétertől tesszük függővé egy eljárás vagy egy osztály működését.

[2]Programozási technika, amely során megszabadulunk egy változó típusától, hogy általános módon lehessen kezelni.

Emellett érződik, hogy nem lenne a legbölcsebb csak hagyni ezeket az entitásokat egymástól teljesen függetlenül lebegni, mivel az, hogy ismerjük ezek létezésének tényét valamilyen fejlesztőbarát könnyen kezelhető módon az elengedhetetlen.

Erre a jogosan megfogalmazódó problémára nyújt megoldást az EntityService nevű komponens.

A létrehozásban segítségünkre lesz a create tagfüggvény, ami klasszikus értelembe véve paraméterek nélküli, viszont templateparaméterben átadható neki egy az Entity ősosztályból származó típus, aminek a megkonstruálása az EntityService feladata lesz majd visszaadja a megfelelő altípusú frissen létrehozott entitást. A visszatérési érték típusának is a templateargumentumot használja.

Ezek után, hogy tudunk entitásokat létrehozni a lekérdezésük válik a következő égető hiányossággá amire megoldásul az EntityService szintű gettereket bizonyultak a legkényelmesebb és legjobban használható megoldásnak. Ezek lettek a:

- getById
- getAll
- getCount
- getAllIds

A getterek neve, habár intuitívan utal arra, hogy pontosan mit csinálnak, nézzük meg, hogy mik a helyes viselkedés, amiket velük szembe elvárunk.

A **getById** pontosan egy paraméteres függvény, ami egy entitás azonosítóját kapja meg paraméterül. Templateargumentumában egy Entitás altípust kap. Elvárt működése a kapott típusnak megfelelő kapott azonosítójú entitás visszaadása.

A **getAll** paraméter nélküli függvény. Templateargumentumában egy Entitás altípust kap. Elvárt működése a kapott típusnak megfelelő összes entitás visszaadása.

A **getCount** paraméter nélküli függvény. Templateargumentumában egy Entitás altípust kap. Elvárt működése a kapott típusnak megfelelő entitások számosságának visszaadása.

A **getAllIds** paraméter nélküli függvény. Templateargumentumában egy Entitás altípust kap. Elvárt működése a kapott típusnak megfelelő összes entitás azonosítójának visszaadása.

Minden különböző entitástípushoz, ami létezik a függvények explicit template specializációval vannak ellátva, hogy a generikus működés jól képeződjön le minden egyes entitás altípusra.

Az osztály a singletinetone[3] tervezési minta szerint készült.

Ezeknek az információknak az ismeretében nézzük meg a publikus inteface szintű implementációt:

```
class EntityService {
public:
    static std::shared_ptr<EntityService> getInstance();

private:
    EntityService(std::shared_ptr<common::EntityCache> entityCache)
        ;

public:
    template<typename Entity_T>
    std::shared_ptr<Entity_T> getById(int id);

    template<typename Entity_T>
    QList<std::shared_ptr<Entity_T>> getAll();

    template<typename Entity_T>
    int getCount();

    template<typename Entity_T>
    QList<int> getAllIds();

    template<typename Entity_T>
    std::shared_ptr<Entity_T> create();

    ...

private:
    static std::shared_ptr<EntityService> instance_;

    ...
};
```

3.2. forráskód. EntityService header file

A fentebb lévő kódrészlet tartalmazza a már írásban taglalt függvénytemplate-ek definícióit, illetve a singletone minta okán privát a konstruktora és publikus statikus

[3]Az osztályból csak 1 példány hozható létre a program teljes futása alatt.

genInstance függvénye van ahogy az a 3. és 6. sorban látható.

3.3.3. EntityCache

A fentebbi részekben már kifejtésre került az entitásainkal végezhető műveletek, és az erre létrehozott osztályokat, service-eket. Ebben a részben az entitásaink tárolásáról tudhatunk meg többet. Mivel a kipróbáló program adatbázis sémával rendelkezik, viszont tényleges adatbázis nincsen a program entitásrétege mögött, valamilyen in memory adatbázisleképezést kellett elkészíteni, hogy ez megfelelően prezentálható legyen, illetve mivel az entitások felépítése alapvetően egy erőforrásigényes folyamat egy gyorsítótár jellegű tároló elkészítése nagyban segíti az alkalmazás reszponzivitását.

Emellet további és talán legfontosabb szerepe az, hogy az alkalmazásunkban ez a gyorsítótár biztosítja az entitásaink pointerintegritását. Ez azt jelenti, hogy ugyan azt az entitást ha lekérdezzük akkor a cache-ben lévő entitást csak egyszer kell megkonstruálni, onnantól kezdve az egész alkalmazásban ugyan azt az objektumot használja minden komponens attól függetlenül, hogy EntityService-en keresztül vagy egy konkrét entitáson keresztül kéri le az adott objektumot.

Ebből is látszik, hogy az EntityCache nem közvetlenül használható a felhasználók, fejlesztők számára, hanem az inkább egy az EntityService, és entitás példányok függőségeként létrjövő, őket közvetlenül kiszolgáló segéd objektum. Ennek a célja, hogy az Entitások tárolása és konténer objektumokból való visszakeresésének feladata ne az entitásokra háruljon, ezt inkább szervezzük ki egy különálló komponensbe, ami dedikáltan csak ezt a feladatot látja el. Nézzük meg az EntityCache-t publikus interface szintjén:

```
1  class EntityCache {
2  public:
3      static std::shared_ptr<EntityCache> getInstance();
4
5  private:
6      EntityCache() = default;
7
8  public:
9      template<typename Entity_T>
10     requires std::is_base_of_v<db::Entity, Entity_T>
11     const QList<std::shared_ptr<Entity_T>> getCached() {
```

```
12          ...
13      }
14
15      template<typename Entity_T, typename Related_T>
16      requires std::is_base_of_v<db::Entity, Entity_T> &&
17               std::is_base_of_v<db::Entity, Related_T>
18      const QList<std::shared_ptr<Related_T>> getRelatedEntitiesOf(
            std::shared_ptr<const Entity_T> entity) {
19          ...
20      }
21
22      template<typename Entity_T>
23      requires std::is_base_of_v<db::Entity, Entity_T>
24      std::shared_ptr<Entity_T> cache(Entity_T* entityRawPtr) {
25          ...
26      }
27
28      template<typename Entity_T>
29      requires std::is_base_of_v<db::Entity, Entity_T>
30      void remove(std::shared_ptr<Entity_T> entity) {
31          ...
32      }
33
34      template<typename Entity_T, typename Related_T>
35      requires std::is_base_of_v<db::Entity, Entity_T> &&
36               std::is_base_of_v<db::Entity, Related_T>
37      void link(std::shared_ptr<Entity_T> a, std::shared_ptr<
            Related_T> b) {
38          ...
39      }
40
41      template<typename Entity_T, typename Related_T>
42      requires std::is_base_of_v<db::Entity, Entity_T> &&
43               std::is_base_of_v<db::Entity, Related_T>
44      void unlink(std::shared_ptr<Entity_T> a, std::shared_ptr<
            Related_T> b) {
45          ...
46      }
47
48      template<typename Entity_T, typename Related_T>
49      requires std::is_base_of_v<db::Entity, Entity_T> &&
```

```
50              std::is_base_of_v<db::Entity, Related_T>
51     void clearLinksOf(std::shared_ptr<Entity_T> entity) {
52         ...
53     }
54
55     ...
56
57 private:
58     static std::shared_ptr<EntityCache> instance_;
59
60 private:
61     QList<std::shared_ptr<db::Administrator>> admins_;
62     QList<std::shared_ptr<db::Fruit>> fruits_;
63     QList<std::shared_ptr<db::User>> users_;
64
65     std::set<std::pair<int, int>> fruitUserRelations_;
66 };
```

3.3. forráskód. EntityCache header file

A fentebbi kódrészletben ismét jól látható módon jelenik meg a singleton tervezési minta. A 3. sorban lévő publikus getInstance felel az objektum lekérdezhetőségéről, és visszaadja az egyetlen objektumra mutató pointert. A 6. sorban lévő privát konstruktor felel az osztály egyetlen példányának megkonstruálásáért. Ez természetesen nem hívható kívülről. Az 58. sorban pedig látható a statikus instance, ami az objektumot tárolja.

Ezek után következnek a service lényegi részei. Mivel az osztály minden típusú entitáshoz generikusan kell, hogy működjön egy igen masszív template definíciókkal ellátott osztály törsz fog következni de minden függvény indokoltan használja ezt a nyelvi elemet.

A **getCached** paraméter nélküli függvény. Templateargumentumában egy Entitás altípust kap. Elvárt működése a kapott típusnak megfelelő összes entitást tartalmazó kollekció érték szerinti visszaadása.

A **getRelatedEntitiesOf** egy paraméterrel rendelkező függvény, ami egy entitás példányt kap. Templateargumentumában kettő Entitás altípust kap, az első megegyezik a paraméterének típusával, a másik egy a paraméterrel relációba állítható Entity altípus. Elvárt működése a kapott entitással relációban lévő összes entitás visszaadása.

A **cache** egy paraméterrel rendelkező függvény ami egy entitás példányt kap viszont raw pointer-en keresztül. Templateargumentumában egy Entitás altípust kap, ez megegyezik a paraméterének típusával. Elvárt működése a kapott entitás raw pointer teljes értékű entitássá alakítása, és ownership kialakítása. Ezt követően pedig az entitás visszaadása.

A **remove** egy paraméterrel rendelkező függvény, ami egy entitás példányt kap. Templateargumentumában egy Entitás altípust kap, ez megegyezik a paraméterének típusával. Elvárt működése a kapott entitás kitörlése az azt tartalmazó kollekcióból, és annak destruálása.

A **link** kettő paraméterrel rendelkező függvény, ami két entitás pédányt kap. Templateargumentumában két Entitás altípust kap, ez megegyezik a paraméterének kapott entitások típusával. Elvárt működése a kapott entitások közötti reláció kialakítása.

A **unlink** kettő paraméterrel rendelkező függvény, ami két entitás pédányt kap. Templateargumentumában két Entitás altípust kap, ezek megegyeznek a paraméterének kapott entitások típusával. Elvárt működése a kapott entitások közötti reláció megszüntetése.

A **clearLinksOf** egy paraméterrel rendelkező függvény, ami egy entitás pédányt kap. Templateargumentumában két Entitás altípust kap, az első megegyezik a paraméterének kapott entitások típusával, a másik szintén egy Entitás altípus. Elvárt működése a kapott entitás összes relációjának megszüntetése a templétargumentumként kapott másik entitástípus összes példányával.

A forráskód privát szekciójában, a 61. sortól kezdve, láthatóak az entitásokat tartalmazó kollekciók, illetve, a 65. sortól, az egyes entitás típusok közötti relációkat tartalmazó konténerek.

3.3.4. Az entitásréteg interakciói a komponensek között

A következő részekben nézzünk meg néhány érdekes interakciót a korábban megismert Entitás réteg komponensei között. Először vessünk egy pillantást az Entitások létrehozásának folyamatára.

```
1 auto aple = entityService->create<db::Fruit>()
2                 ->setName("Alma");
```

3.4. forráskód. Alma nevű gyümölcs entitás létrehozása

A fentebbi mintakódban az látható, ahogy az EntityService segítségével létrehozunk egy Gyümölcs entitást, majd ennek az entitásnak beállítjuk a név tulajdonságát az Alma szöveges literálra, és végül az egész kifejezés kiértékelése után a visszatérési értéket berakjuk egy aple nevű változóba. Nézzük meg, hogy az egyes részek miként működnek és pontosan merre jár a vezérlés ennek az igen kényelmesen használható, intuitív kódrészlet kiértékelése közben. Az első kiértékelésre kerülő kifejezés az entityService objektum példányon meghívott create utasítás lesz.

```
template<>
std::shared_ptr<db::Fruit> common::EntityService::create() {
    return entityCache_->cache(new db::Fruit());
}
```

3.5. forráskód. Create belső működése

A vezérlés az EntityService, Fruit típussal történő explicit templatespecializációjába fog megérkezni, ahol az entityCache objektum példányán meghívjuk a cache függvényt. Ez paraméterül kap egy helyben default konstruktorral, a heap-en létrehozott gyümölcs típusú entitást.

```
Fruit::Fruit()
    : Entity(nextId_++)
{}
```

3.6. forráskód. Fruit belső működése

Itt a konstruktor létrehozza az entitást, és meghívja az ősének az Entity-nek a konstruktorát, aminek továbbadja a nextId_ nevű statikus változót, rögtön ezt követően meg is növelve azt. Ennek segítségével érhető el az entitások Id-jának inkrementális növelése. Ezt követi az Entity konstruktora.

```
Entity::Entity(int id)
    : entityCache_(common::EntityCache::getInstance())
    , id_(id)
{}
```

3.7. forráskód. Entity belső működése

Itt az entitásban lévő EntityCache példány megkapja az EntityCache::getInstance függvénye által visszaadott egyetlen objektumra mutató pointert. Emellett elvégzi a másik nagyon fontos feladatát, ami az Id beállítása. Ezt természetesen a member initializer list-je segítségével teszi meg, mivel az a best

practice, másrészt pedig mivel a változó const megszorítása miatt nem lehetne ezt a konstruktor törzsében megtenni.

Ezt követően a vezérlés visszatér az EntityService-be, ahol végül pedig a kifejezés visszatérési értékével tér vissza ő maga is, de ez előtt a vezérlés tovább halad az EntityCache cache nevű függvényre a frissen megkonstruált pointert átadva annak.

```
template<typename Entity_T>
requires std::is_base_of_v<db::Entity, Entity_T>
std::shared_ptr<Entity_T> cache(Entity_T* entityRawPtr) {
    auto entity = std::shared_ptr<Entity_T>(entityRawPtr);
    getListOfType<Entity_T>().append(entity);
    return entity;
}
```

3.8. forráskód. cache belső működése

Itt pedig megtörténik a már korábban megismert függvény feladatának beteljesítése, ami az entitás feletti ownership kialakítása, és eltárolása. Ez technikailag egy std::shared_ptr használatával teszi meg. Az erről készített másolat bekerül a belső kollekcióba, ami a templateargumentum alapján kerül kiválasztásra a getListOfType privát függvény segítségével. Ezt követően a függvény visszatér az eredeti pointer-rel és bezárul a kör.

A vezérlésünk visszatér a vermünk felszínére, ahol a következő utasítás a már megkonstruált entitás nevének beállítása. Ez itt már természetesen a Fruit entitás egyik szettere lesz.

```
std::shared_ptr<db::Fruit> Fruit::setName(const QString& name) {
    name_ = name;

    return shared_from_this();
}
```

3.9. forráskód. Fruit belső működése

Itt egy triviális szettert láthatunk egy Builder tervezési mintával ellátva, amiben annyi plusz csavar van, hogy a builder a this-t referencia helyett egy std::shared_ptr-be csomagolva adja vissza. Ennek köszönhető, hogy a hívó fél a szetter használata után is értékül adhatja egy változónak az entitását.

Ezt ki is használva a teljes kifejezés visszatérési értékét odaadjuk a változónak és ezzel létre is jött az Alma nevű Gyümölcs típusú entitás.

3.4. Delayed resource lock service

A fentebb leírt eszközök működése mellett felmerülhetnek további olyan problémák, amikre ez az eszköztár nem nyújt teljes körű megoldást. Az egyik ilyen felhasználói eset például, ha egy folyamatot meg akarunk várakoztatni az erőforrások elérhetetlensége miatt, akkor erre nemigazán van jó módszer, illetve az erőforrásoknak nem tájékozódunk a felszabadulásáról ezért az, hogy mennyi időt kell várni azt valamilyen módon a felhasználónak kell kitalálnia. A DelayedResourceLockService ennek a kimondott esetnek a kezelésére lett megalkotva.

Abban az esetben, ha a folyamat jellege miatt nem érdekes, hogy mikor lesz elvégezve csak az a célunk, hogy előbb-utóbb a végrehajtás történjen meg A DelayedResourceLockService lesz a jó választás. A service egy teljesen a felhasználótól függő névtelen függvényt kap paraméterül, és egy halmaznyi lefoglalandó erőforrást, amik a folyamat elvégzéséhez szükségesek. A működése egyszerű, igazából kettő fő esetre lehet szétválasztani.

Az első eset az, amikor a folyamat által igényelt paraméterül átadott erőforrások elérhetőek. Ekkor ezek gond nélkül lefoglalásra kerülnek, a zár birtokbavétele megtörténik. Ezt követően a folyamat lezajlik, ennek végeztével az erőforrások felszabadulnak. Ez az eset ekvivalens a ResourceLockService azon esetével, amikor az erőforrások lefoglalása megtörténik. Ez egyébként a tervezés során elvárt viselkedés is volt.

A másik eset, amikor a szükséges resource-ok lefoglalása valamilyen hibára fut, természetesen ez csak azt jelenti, hogy más éppen dolgozik azokkal. Ekkor a felhasználótól paraméterben kapott névtelen függvény az erőforrásokkal együtt bekerül egy várakoztatási sorba. Új szál nem indul ezekkel egyidejűleg, hogy ez ne csökkentse a thread pool-ban lévő használható szálak mennyiségét. Ebből a várakoztatási sorból az elemek kettő féleképpen kerülhetnek ki.

Az egyik módja, ha a ResourceLockService elemmitál egy szignált, ez jelzést ad arról, hogy az eddigi lefoglalt resource-ok halmaza megváltozott. Erre a jelzésre figyelve a DelayedResourceLockService végig iterál a várakoztatási soron, és megkísérli az újonnan felszabadult erőforrásoknak megfelelő elemeket lefuttatni. Amennyiben ez sikeresen lezajlik, a folyamat befejeződött az elem kikerül a várakoztatási sorból. A bennmaradó elemeken ismét végig iterálunk a jelzések érkezésével, így fokozatosan elfogyasztva a lista teljes tartalmát.

A másik módja az elemek listából való eltávolításának az, ha az elem által igényelt erőforrások nem szabadulnak fel az elemnek megadott időtúllépési limit tartalma alatt. Az elemeknek meghatározható a felhasználó által egy maximális időkorlát amíg bennmaradhat egy elem a várakoztatási sorban. Amint a várakoztatási sorban töltött idő átlépi a maximálisan dedikált időt a feladat elvégzésére, a feladat az időtúllépési státuszba lép. Ekkor a feladat végrehajtása nem valósul meg, a várakoztatási sorból kikerül, végrehajtás nélkül.

Tipikus példája ez a kiéheztetésnek és ez a mechanizmus az örökké várakozó folyamatok beragadását hivatott elkerülni. De mit tudunk tenni abban az esetben, ha a folyamatunkat kiéheztették? Erre a kérdésre is a komponens felhasználója kell, hogy fel legyen készülve hisz a komponens csak a folyamatok megvárakoztatásáért és végrehajtásáért felel. Ha egy folyamat kiéheztetése megtörténik akkor az a hatáskörén kívül eső probléma, ami ennél jóval mélyebben is gyökerezhet. A felhasználó erre gondolva adhat egy opcionális paraméterként egy időtúllépés esetén lefuttatandó folyamatot. Ebben megkísérelhet tenni a kiéheztetést végző folyamatok ellen vagy logot helyezhet el a hibajelenség detektálhatóságának érdekében és későbbi javíthatóságának megkönnyítéséért. Az a program, ami kiéhezteti a saját folyamatait eleve rosszul működik tehát ez a rész igazából egy megoldás a végső esetre, de jobb, hogyha van és sosincs kihasználva, mintha nem lenne és egyszer egy ilyen jellegű hiba felmerülése során egy jóval nagyobb kódbázist kell kielemezni a hiba helyének feltárása érdekében.

Nézzük meg az osztályt kicsit technikaibb szemmel, hogy a működések hátterében levő mechanizmusokról kiderüljenek a részletek.

3.4.1. Függőségek

A komponens explicit függőségei az AsyncTaskService az aszinkron folyamatok kezelésére hivatott komponens, és a ResourceLockService, ami segítségével az erőforrások lefoglalását végzi a komponens. Ezeket konstruktorparaméterben kapja.

3.4.2. Belső típusok

A komponens egy privát belső típussal rendelkezik, név szerint az AsyncLock-kal. Ebben a típusban tudjuk eltárolni a várakozó folyamatokat.

```
1 struct AsyncLock {
```

```
    std::variant<common::CallerContext, QString> contextOrTag_;
    std::map<common::LockableResource, common::ResourceLockType>
        resources_;
    AsyncTaskPtr task_;
    std::unique_ptr<QObject> guard_ = std::make_unique<QObject>();

    ...
};
```

3.10. forráskód. AsyncLock belső típus

Itt látható a felhasználó által megadott task, az erőforrások map-je, és egy guard objektum, ami egy nyers QObject. Erre a timeout task miatt van szükség.

3.4.3. Tagváltozók

A tagváltozóink között az osztályunk két függősége mellett egy lista látható, ami AsyncLock belső típussal rendelkező elemeket tárol. Ez a technikai implementációja a várakoztatási sornak. Közvetlenül alatta látható egy std::mutex, aminek a szerepe a lista szinkronizációja. A lista akár egyszerre több szálból is módosulhat. A lehetséges racecondition-ök elkerülése végett a lista minden művelete lelokkolja a mutexet az implementációban. Alatta található két std::atomic-ba wrapp-elt logikai érték, ezek szerepe a lista feldolgozása során van.

```
std::shared_ptr<common::IResourceLockService> resourceLockService_;
std::shared_ptr<common::AsyncTaskService> asyncTaskService_;

QList<std::shared_ptr<AsyncLock>> asyncLocks_;
std::mutex asyncLocksMutex_;

std::atomic_bool inProgress_      = false;
std::atomic_bool hasMissedSignal_ = false;
```

3.11. forráskód. Tagváltozók

3.4.4. Publikus interface

A publikus interface-en található kettő függvény, ami az adminisztrátor és rendszer szintű folyamat futtatásra lett megalkotva. A paraméterlistájuk nagyrész megegyezik így ezt egyben fogom kifejteni.

Az első paraméter az adminisztrátor szintű hívás esetén egy CalleContext kontextust tartalmazó objektum, ami az adminisztrátor beazonosításában játszik szerepet. A rendszer esetben ez egy egyszerű szöveges változó, ami a rendszer beazonosításában játszik szerepet.

A második paraméter a már korábban megismert LockableResource kulcsokhoz ResourceLockType kulcsaikat rendelő map, ez itt is a lefoglalandó erőforrásokat és azok írási vagy olvasási típusú zárolásának többletadatát hordozza.

A harmadik paraméter természetesen a task, aminek az elvégzéséhez a második paraméterben átadott erőforrások szükségesek.

A negyedik paraméter egy időtúllépési korlát, amit milliszekundumban adhatunk meg. Ezzel megszabhatjuk, hogy a folyamat által igényelt, második paraméterként átadott erőforrások, rendelkezésre nem állásának állapota maximálisan mennyi ideig fogadható el a feladat elvégzésének szempontjából. Fontos, hogy ez az időtúllépés csak a várakozásra vonatkozik, ha a folyamat már elkezdődött akkor, a futásának ideje nem számít bele az időtúllépésnek megszabott felső határba, tartson bármilyen sokáig.

Ezt követően következik az utolsó paraméter, ami egyben az első opcionális paraméter, az időtúllépési task. Ez akkor fut le, ha a kijelölt milliszekundumnyi idő alatt nem sikerül az erőforrások lefoglalása.

```
1  void addAsyncLock(CallerContext context,
2                    std::map<LockableResource, ResourceLockType>
                        resources,
3                    AsyncTaskPtr task,
4                    int timeoutMs,
5                    AsyncTaskPtr timeoutTask = nullptr) override;
6
7  void addAsyncSystemLock(QString tag,
8                          std::map<LockableResource, ResourceLockType
                               > resources,
9                          AsyncTaskPtr task,
10                         int timeoutMs,
11                         AsyncTaskPtr timeoutTask = nullptr)
                               override;
```

3.12. forráskód. Publikus interface

3.5. Tesztelési tervezet

A tesztelés során a DelayedResourceLockService és az AsyncTaskService integrációs tesztjei kerültek megvalósításra. Ennek oka az, hogy az AsyncTaskService szorosan összefügg a működéssel az aszinkron mivoltából kifolyólag és nem lehetne jól mock-olni a működését. A másik függőség, ami a ResourceLockService természetesen jól mock-olható objektum, így az ilyen módon szerepel a tesztelésben. A tesztek során a DelayedResourceLockService teljeskörű tesztje valósul meg törekedve a minél nagyobb kódbázis lefedettségének elérésére és a minél változatosabb végrehajtásokra.

Integrációs tesztek	
Teszteset	*Leírás*
single AsyncLock AvailableResources	A teszt azt vizsgálja, hogy elérhető erőforrások mellett egy egyedüliként indított adminisztrátor jogosultságú folyamat lefut-e.
single AsyncSystemLock AvailableResources	A teszt azt vizsgálja, hogy elérhető erőforrások mellett egy egyedüliként indított rendszer jogosultságú folyamat lefut-e.
multiple AsyncLock AvailableResources	A teszt azt vizsgálja, hogy elérhető erőforrások mellett több egyidejűleg indított adminisztrátor jogosultságú folyamat lefut-e.
multiple AsyncSystemLock AvailableResources	A teszt azt vizsgálja, hogy elérhető erőforrások mellett több egyidejűleg indított rendszer jogosultságú folyamat lefut-e.
single AsyncLock LockedResources	A teszt azt vizsgálja, hogy lefoglalt erőforrások mellett egy egyedüliként indított adminisztrátor jogosultságú folyamat bekerül-e a várakoztatási sorba, majd a megadott idő elteltével kikerül-e belőle az időtúllépés miatt.
single AsyncSystemLock LockedResources	A teszt azt vizsgálja, hogy lefoglalt erőforrások mellett egy egyedüliként indított rendszer jogosultságú folyamat bekerül-e a várakoztatási sorba, majd a megadott idő elteltével kikerül-e belőle az időtúllépés miatt.

Teszteset	*Leírás*
single AsyncLock LockedResources WitchThenFreesUp	A teszt azt vizsgálja, hogy lefoglalt erőforrások mellett egy egyedüliként indított adminisztrátor jogosultságú folyamat bekerül-e a várakoztatási sorba, majd később amikor a megfelelő erőforrások felszabadulnak, a sorban lévő folyamat elindul-e és végrehajtja-e a neki kijelölt feladatot. Illetve, hogy ennek végeztével kikerül-e a sorból.
single AsyncSystemLock LockedResources WitchThenFreesUp	A teszt azt vizsgálja, hogy lefoglalt erőforrások mellett egy egyedüliként indított rendszer jogosultságú folyamat bekerül-e a várakoztatási sorba, majd később amikor a megfelelő erőforrások felszabadulnak, a sorban lévő folyamat elindul-e és végrehajtja-e a neki kijelölt feladatot. Illetve, hogy ennek végeztével kikerül-e a sorból.
two Concurrent AsyncLock	A teszt azt vizsgálja, hogy két egymással az erőforrásokon konkuráló adminisztrátor jogosultságú folyamat gond nélkül le tud-e futni. A sorrendjük természetesen nem determinisztikus tehát a teszt a futásuk sorrendjéről nem feltételez semmit.
two Concurrent AsyncSystemLock	A teszt azt vizsgálja, hogy két egymással az erőforrásokon konkuráló rendszer jogosultságú folyamat gond nélkül le tud-e futni. A sorrendjük természetesen nem determinisztikus tehát a teszt a futásuk sorrendjéről nem feltételez semmit.

3.1. táblázat. Automatizált integrációs tesztek összegzése

4. fejezet

Összegzés

A projekt elkészítése igen hosszú időt vett igénybe, sok új technikát és sok hasznos dolgot volt szerencsém elsajátítani a készítése közben. Sok tapasztalatot szereztem konkurens programozás témában és egyéb C++ tool-okat ismerhettem meg mint például a smart pointer-ek.

A komponens, az elkészülése után, bekerül a cég legnagyobb termékébe, ahol fontos feladatot lát el.

Az elkészítési folyamat izgalmas kihívásai után a szakdolgozat elkészítése is tartogatott további érdekes kihívásokat és néhány újdonságot is mivel itt a céges C++ 17-es szabvány helyett a C++ 20-as szabványát használtam.

Irodalomjegyzék

[1] „C++ Dokumentáció”. 2020. URL: https://en.cppreference.com/w/.

[2] „Qt Dokumentáció”. 2023. URL: https://doc.qt.io/.

Ábrák jegyzéke

Táblázatok jegyzéke

Forráskódjegyzék