Alba Regia Műszaki Kar
ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY
Természettudományi és Szoftvertechnológiai Intézet

SZAKDOLGOZAT

Energiahatékony feladatszámítás a szoftverfejlesztés területén

OE-AMK

2024

Hallgató neve: Gorácz Roland

Hallgató törzskönyvi száma: T001022/FI12904/A-M

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Óbudai Egyetem
Alba Regia Műszaki Kar
Természettudományi és Szoftvertechnológiai Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: Gorácz Roland
Szakdolgozat száma: SZD2301230928341247[redacted]
Törzskönyvi száma: T001022/FI12904/A-M **Neptun kódja:** [redacted]

Szak: mérnökinformatikus
Specializáció: Felhő szolgáltatási technológiák és IT biztonság specializáció

A dolgozat címe: Energiahatékony feladatszámítás a szoftverfejlesztés területén
A dolgozat címe angolul: Energy Efficient Computing in the Area of Software Engineering
A feladat részletezése: A hallgató vizsgálja meg, hogy milyen módszerek segítségével fejleszthetők az általánosnál energiahatékonyabb szoftverek. A hallgató elemezze, illetve tegyen javaslatot olyan metodologikák alkalmazására, melyek mentén futási időben hatékonyabb szoftver állítható elő. A hallgató válasszon ki alkalmas fejlesztési módszertanokat a globális felmelegedés problémájának a hatékonyabb szoftverek fejlesztésén keresztüli mérséklésére. Részletezze a példaalkalmazás specifikációját, megtervezését, fejlesztését, a bevezetést beleértve. A hallgató értékelje az elkészült koncepciót, és mutassa be a továbbfejlesztési lehetőségeket, illetve mutassa be az energiahatékonyság becslése során kapott eredményeket, majd értékelje azokat.

Intézményi konzulens neve: Módné Takács Judit

A kiadott téma elévülési határideje:2026. 02. 10.
Beadási határidő: 2023. 12. 15.

A szakdolgozat: Nem titkos.

Kiadva: Székesfehérvár, 2023. 10. 27. P.H.

ÓBUDAI EGYETEM Alba Regia Műszaki Kar Természettudományi és Szoftvertechnológiai Intézet Székesfehérvár

.....
Intézetigazgató

A dolgozatot beadásra alkalmasnak találom. 2023. 12. 15.

.....
belső konzulens

.....
külső konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Alba Regia Műszaki Kar
Geoinformatikai/Mérnöki/
Természettudományi és Szoftvertechnológiai Intézet

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a dolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült dolgozatban található eredményeket az Óbudai Egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja, a titkosításra vonatkozó esetleges megkötések mellett.

Kelt: Budapest *(hely)*, 2024.05.05 *(dátum)*

Gordea Dávid
.....
hallgató aláírása

Tartalomjegyzék

1. BEVEZETÉS

Manapság egyre nagyobb figyelem irányul a globális piacon az energiahordozók effektív felhasználására, az energiatakarékosságra, a globális felmelegedés különböző hatásaira, és nem mellesleg, általánosságban véve a környezetünk kímélésére.

Mind szoftverfejlesztőként, mind Z generációs növendékként feladatomnak tekintem a környezetkímélő életvitelt, hiszen a mindennapokban már én is egyértelműen tapasztalom a fiatal koromhoz képest bekövetkezett negatív változásokat.

Személyes meggyőződésem, hogy minden fejlesztési műveleteket és döntéseket végző személy aktívan hozzá tud járulni ahhoz, hogy csökkentse a világon működő szoftverek, és az azok mögött elhelyezkedő infrastruktúra ökológiai lábnyomát azzal, hogy megfelelő architektúrájú, hatékony és nem energiapazarló szoftvereket fejleszt, és hogy a mögöttes infrastruktúrát nem pazarolja, megfelelően skálázza, legyen szó akár horizontális, akár vertikális skálázásról.

Előzőekben leírtak alapján adott tehát tézis, mi szerint amennyiben a szoftvermérnökök kiemelt figyelmet szentelnek alkalmazásaik párhuzamosítására, a legtöbb számolt érték gyorsítótárazására, és általánosságban véve a zöld programozás ajánlásai mentén történő optimalizációra, mindezzel olyan, jelentős energiamegtakarítás érhető el, mely jelentős erőforrás megtakarítással jár a Földre nézve.

Célom tehát bíztatni a hardver- és szoftverfejlesztőket – nem mellesleg a professzionális hierarchia szempontjából felettük álló döntéshozókat – arra, hogy kiemelt figyelmet szenteljenek az energiahatékony platformok kifejlesztésére és a zöld programozás irányelvei alapján történő szoftverfejlesztésre és üzemeltetésre, ugyanis az előzőekben említettekből következésképp és ez alapján ezzel aktívan hozzájárulhatunk környezetünk megóvásához. Mindezt szeretném tézisem igazolásával alátámasztani, melynek keretében összehasonlítom dolgozatomban egy komplex szoftver ezen irányelvek alapján történő fejlesztését és üzemeltetését, és szintén ugyanezen szoftver az irányelveket nem követő változatát energiafelhasználás, hardverkihasználás, és egyéb szempontokból.

Célom továbbá olyan fejlesztési minták felfedezése, megfogalmazása, melyek segítségével és követésével kevéssé energiaigényes szoftverek fejleszthetők az

átlagszoftverekhez képest, melyek végül aztán hozzájárulhatnak a környezet kisebb mértékű szennyezéséhez, és a globális felmelegedés mértékének csökkentéséhez.

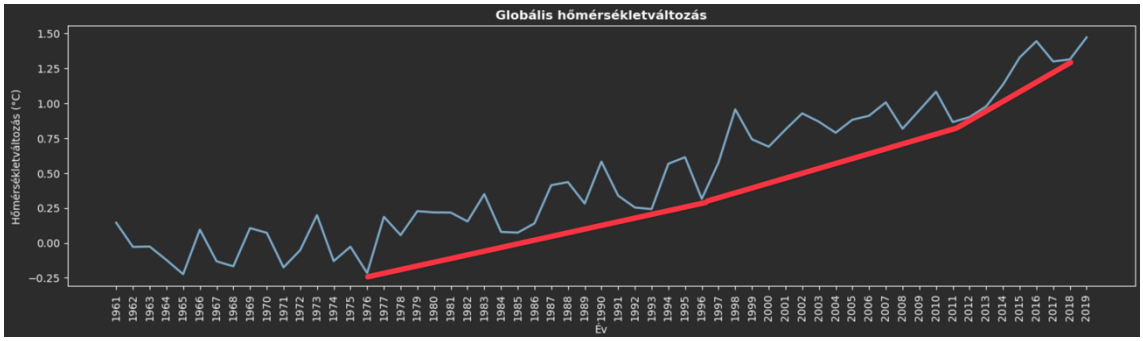
A következő, ennek dedikált fejezetben részletesen taglalom a motivációm mögöttes okait.

2. MOTIVÁCIÓ

Dolgozatom témájának megválasztásában két fő tényező játszott szerepet: a globális felmelegedés (avagy hogyan járulhatna hozzá a szoftver ennek csökkentéséhez), illetve az energiahordozók felértékelődése (avagy hogyan lehetne hatékonyabban felhasználni a szoftver futása során egységnyi energiát).

2.1 A globális felmelegedés ijesztő mértéke

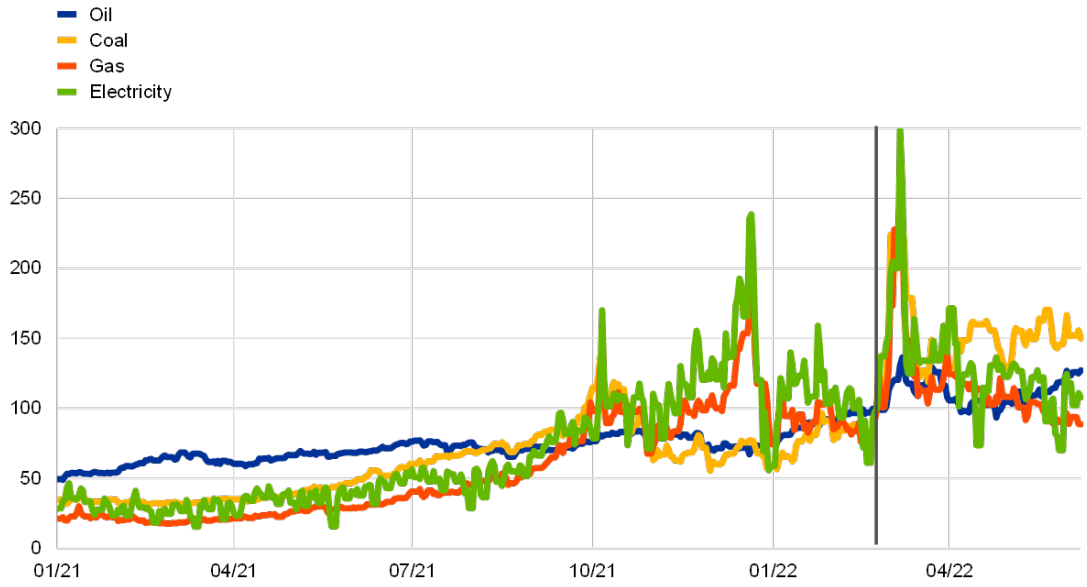
Ahogyan az a 2.1. ábráról leolvasható és ismert, egyértelműen növekvő tendenciát mutat a Föld globális hőmérséklete egészen a 19. századi iparosodás óta, azonban ez a trend mind a 20., mind a 21. században folytatódott. Az ábráról egyértelműen leolvasható, hogy mindennek hatásai körülbelül 1976-tól voltak jelentősebb mértékben megfigyelhetők, amikor is bekövetkezett az úgynevezett Nagy Csendes-óceáni Éghajlatváltozás. A növekvő trend azóta sem csökkent, sőt 2011 óta a korábbinál is gyorsabb növekedés figyelhető meg.



2.1 ábra Globális hőmérsékletváltozás 1961 és 2019 között [saját szerkesztés] [1]

Mint ismert, ennek fő okozója az üvegházhatást fokozó gázok, különösképp a szén-dioxid kibocsátás egyre nagyobb mennyiségben történő kibocsátása, mely az informatika területén csökkenthető lenne azáltal, hogy energiahatékonyabb hardvereket, illetve optimálisabb szoftvereket használunk, állítunk elő. Amennyiben élünk azzal a feltételezéssel, hogy az előzőekben taglaltak szerint járunk el, ez esetben jelentősen kevesebb erőforrást kellene igénybe vennünk a különböző hardver elemek megfelelő hűtésére, mely szintén szén-dioxid kibocsátással jár, kiemelve a szerverparkok kellő mértékben történő hűtését.

2.2 Az energiahordozók felértékelődése



2.2 ábra Energiaárak alakulása az orosz-ukrán háború 2022-es erőnyerésének hatására [2]

Ahogyan az a 2.2 ábrán látható, az elmúlt időszakban számottevő tényező lett az energiával történő logisztikai gazdálkodás. Az Európai Központi Bank által készített kutatás [2] alapján megfigyelhető ugyanis, hogy az orosz-ukrán háború új erőre kapása révén az energiaárak szignifikáns növekedésnek indultak.

Hozzájárul az energiaárak növekedéséhez a nem megújuló energiaforrások számának folyamatos csökkenése is, hiszen előbb-utóbb csak megújuló energiaforrásokból lesz majd előállítható újabb energia – ezt szemlélteti a 2.2 ábrán a szén árának jelentős felértékelődése.

A 2.2. ábrán látható, hogy az előzőekben említett okokból kifolyólag nemcsak a szén, hanem az olaj ára is növekőben van a koronavírus pandémia óta, így továbbra is fontos szempontként említhető az energiával történő effektív gazdálkodás.

Hatékony hardver-szoftver párosítással azonban növelhető a nagy számítási kapacitásokat igénylő folyamatok hatékonysága, mellyel egységnyi számítás kevesebb energiával is elvégezhető – ez kiemelten fontos az úgynevezett High-performance Computing (HPC) és a mesterséges intelligencia mögöttes infrastruktúrája esetében.

A mesterséges intelligencia exponenciális terjedése

Az OpenAI vállalat által fejlesztett GPT-3-mas mesterséges intelligencia modell 2022-es megjelenését követően mértéktelen léptékkel kezdett el terjedni a hasonló modellekre épülő, mesterséges intelligenciára épülő megoldások fejlesztése és alkalmazása.

A mesterséges intelligencia széleskörű elterjedésének és alkalmazásának azonban van egy óriási hátránya a szociológiai és pszichológiai hatásokat leszámítva, melyekről mindenki igyekszik hallgatni: ez pedig a technológia energiafogyasztása.

Az MTI által publikált, „Great Power, Great Responsibility: Recommendations for Reducing Energy for Training Language Models" című cikk [3] ugyanis a terület körüljárását követően egyértelműen leszögezi: a mesterséges intelligencia modellek tanítása igenis nagy energiafogyasztással jár. Azonban mindennek van egy olyan aspektusa, mely még energiaigényesebb, és emellett gyakrabban kerül előtérbe: ez magának a már betanított modellnek a használata.

Az említett tényezőkből kifolyólag fontos rávilágítani annak fontosságára, hogy a hardvergyártók, kiemelve a videokártya gyártókat törekedjenek a minél energiahatékonyabb hardver megtervezésére, a szoftverfejlesztők pedig szintén összpontosítsanak az energiahatékonyságot előtérbe helyező szoftver megtervezésére.

A társadalmi viszonyok átrendeződése

A társadalmi viszonyok átrendeződése, a Z generáció és az alfa generáció által széleskörűen használt számítástechnikai eszközök energiafelhasználása összességében mérve is hatással vannak a globális energiafogyasztásra. Ráadásul jellemző, hogy a szegényebb rétegek régebbi, x86-os architektúrájú, energiapazarló számítógépeket használnak, melyek szintén negatívan érintik a vizsgált területünket.

Szintén fontos megemlíteni, hogy bizonyos társadalmi átrendeződések miatt egyre többen folytatnak nagy energiaigényű tevékenységeket a számítógépeken, például számítógépes játékok futtatását, ezek élő közvetítését.

Az előzőekben leírtak alapján fontos lenne realizálnia a számítógépes játékiparnak, és az ehhez hardvert gyártóknak, hogy érdemes lenne elhagyni az x86-os és x64-es

platformokra történő játékfejlesztést, ugyanis az arm, nevezetesen az egyre szélesebb körben elterjedt arm64-es platform jóval energiahatékonyabb számításokkal kecsegtet.

3. A MEGOLDANDÓ PROBLÉMA MEGFOGALMAZÁSA

A bevezetésben említettek alapján tehát a célom olyan fejlesztési minták felfedezése, megfogalmazása, melyek segítségével és követésével kevéssé energiaigényes szoftverek fejleszthetők az átlagszoftverekhez képest, melyek végül aztán hozzájárulhatnak a környezet kisebb mértékű szennyezéséhez, és a globális felmelegedés mértékének csökkentéséhez.

Fontos azonban megjegyezni, hogy a legkevésbé hatékony szoftverek általában úgynevezett legacy rendszerek, melyek bizonyos nagy vállalatok birtokában állnak, és ezen cégek nem feltétlenül lesznek motiváltak abban (például pénzbőség, emberi erőforrás hiánya, vagy egyéb okokból kifolyólag), hogy ezen szoftverek modernizálás alá essenek. Nem mellesleg ebben a kontextusban az sem ritka, hogy bizonyos speciális szoftverekhez speciális hardver párosul, tehát nem lehet egyszerűen kicserélni a mögöttes infrastruktúrát egy hatékonyabbra, és ezzel viszonylag egyszerűen orvosolni a problémát.

A szoftverek nem energiahatékony működése több aspektusból tevődik össze, melyeket bizonyos, különböző technikákkal hatékonyan ki lehet küszöbölni. Az alábbiakban ezek kerülnek szemrevételezésre.

Felesleges újraszámítások

A legfőbb ok, amiért a legtöbb szoftver nem energiahatékony, az abból az egyszerű tényből ered, hogy a szoftverfejlesztők többsége nem épít gyorsítótárazást szoftverébe. Ennek eredményeképpen jelentősen sokszor újraszámításra kerülnek olyan értékek, melyek egyébként huzamosabb ideig nem is változnak.

Elavult hardver

Egy másik oka az energiapazarlásnak az elavult szerverparkok üzemeltetése, konzumer szinten pedig elavult számítógép alkatrészek használata. Az 5-10 évvel ezelőtt megjelent x86-x64 architektúrájú processzorok ugyanis mai szemmel nézve, a mai processzorokhoz hasonlítva ineffektív működésűek, ráadásul ezen processzorok hőkibocsátása jelentős, nem mellesleg hűtésük szintén jelentős energiaigénnyel rendelkezik. [4] [5]

Elavult technológiai stack használata

A régi technológiák, futtatókörnyezetek jelentős mértékben több memóriát foglalnak le az operatív tárban, mint modernebb társaik, sőt számítási teljesítményük is alulmarad, de azt is meg kell említeni, hogy többségük még csak nem is képes teljes mértékben kihasználni a számára rendelkezésre álló erőforrásokat. Sok üzemeltető ezért úgy dönt, mikor azt tapasztalja, hogy megnövekedett alkalmazásának használata, hogy vertikálisan skálázza infrastruktúráját, mely minden esetben erőforráspazarláshoz vezet – ugyanis egyik esetben több erőforrást fog használni a futtatókörnyezet, mint amennyit egy modernebb futtatókörnyezet ugyanakkora terhelés mellett fogyasztana, másik esetben pedig kihasználatlanul marad a futtatókörnyezet számára rendelkezésre bocsátott erőforrások egy része, így effektíven kárba vesznek ezek.

Rossz szoftver vagy infrastruktúra architektúra

Az előző ponthoz kötődve a rossz IT architektúra is problémás pont az energiahatékonyság tekintetében. Ugyanis nagyon könnyen előfordulhat, hogy amennyiben monolitikus szoftver architektúrával rendelkezik szoftverünk, vagy egyszerűen vertikálisan skálázzuk a mögöttes infrastruktúrát, kihasználatlan erőforrások maradnak hátra.

4. A SPECIFIKÁCIÓ KIDOLGOZÁSA

A problémákra való tekintettel megfogalmazható egy sor olyan szoftver réteget érintő specifikáció, mely segít csökkenteni egy szoftver létrehozásának és üzemeltetésének energialábnyomát.

Ezen pontok a dolgozat későbbi szakaszában kerülnek részletesebben taglalásra.

Mindenképpen érdemes gyorsítótárazást alkalmazni, hiszen adott időtartományban történő lekérések többségében ugyanaz az adat utazik. Ennek értelmében az adatokat létrehozásukkor gyorsítótárban tároljuk el, mely csak akkor kerül invalidálásra, amennyiben az ténylegesen megváltozik.

A valós idejű adatfeldolgozás és adatáramlás lehetővé teszi, hogy a gyorsítótár valóban csak és csak akkor legyen invalidálva, amint adat ténylegesen megváltozik. Emellett biztosítja felhasználó számára a kiemelt felhasználói élményt, hiszen azonnal hozzájuthat minden újonnan keletkező adathoz valós időben. [6]

Az alkalmazás párhuzamosítása segítséget nyújt a hardver effektív kihasználásában, mely gyorsabb számításokkal és kisebb energiafelhasználással párosul, így nélkülözhetetlenné válik. [7]

A feladatok párhuzamosítása során kiemelt fontossággal bír ezen feladatok ütemezésekor az ütemezés megfelelő algoritmusának kiválasztása attól függően, hogy milyen jellegű műveletet hajtunk végre az utasításokban (például I/O műveletet) [7]. A modern technológiai eszközök – ideértve programozási nyelveket, futtatókörnyezeteket, adatbázisokat, és egyéb komponenseket – segítenek a hardver minél jobb kihasználtságának elérésében, nem mellesleg általánosságban véve kevesebb erőforrást is igényelnek a rendszertől, így mindenképpen érdemes egy modern technológiai stack-et alkalmazni a megvalósítás során.

5. LEHETSÉGES MEGKÖZELÍTÉSI MÓDOK ÉS MEGOLDÁSOK ÁTTEKINTÉSE ÉS ELEMZÉSE

Alapvetően két lehetséges megoldási mód áll rendelkezésre az előzőekben felvázolt problémahalmazra. A probléma ugyanis megközelíthető hardver, illetve szoftver oldalról egyaránt.

Hardveres megvalósítás

A problémák orvoslása hardver oldalról is megközelíthető.

Ütemező algoritmusok hardverben történő optimalizálása

Lehetőség nyílik a feladatok ütemezésének további optimalizálására mikrokód szinten a feladatok bizonyos algoritmus alapján történő okos szétosztásával a processzor magjai között, mely módon algoritmustól függően csökkenthető a CPU-n belül elvesző energia, vagy az általa termelt hő mennyisége. [7]

Hatékonyabb utasításkészletű processzor használata

További lehetőség újabb utasításkészletek kidolgozása, illetve processzorok gyártása, vagy egyszerűen modernebb utasításkészletű processzorok használata az x86/64-es architektúránál, mint amilyen az ARM platform a maga, úgynevezett Reduced Instruction Set Computer, azaz RISC architektúrájával. Ez nagyobb energiahatékonyságot eredményez utasításainak rövidebb hossza mivoltából, mint a Complex Instruction Set Computer architektúrával rendelkező x86/64-es platformok.

A processzormagok tervezésének okos megválasztása

Elsősorban az Apple, majd az Intel által követett trendszerű praktika, hogy ezen cégek processzoraikat felszerelik mind teljesítményre optimalizált, mind energiahatékonyságra optimalizált magokkal, és a rendszerre nehezedő terhelés, illetve a sorban álló feladatok prioritása alapján a mikrokód és az operációs rendszer együttesen a lehető leghatékonyabban próbálja meg a feladatok ütemezését végrehajtani.

A processzor lapkáján található 1-es, 2-es és 3-mas szintű gyorsítótár méretének megnövelése

Utolsó lehetőségként felmerül a gyorsítótárazás hatékonyságának javítása a processzor lapkáján található, különböző elérési idővel és mérettel rendelkező tárak méretének növelésével. Ez esetben a processzornak kevesebb FETCH utasítással kell az operatív memóriához fordulnia, mely gyorsabb és hatékonyabb utasításvégrehajtást eredményez.

5.1 Szoftveres megvalósítás

A szoftveres megvalósításnak is többféle aspektusa jelenik meg.

Mikroszerviz architektúra használata

A mikroszerviz architektúra lehetővé teszi, hogy üzleti logika szerint kis egységekbe szervezzük ki az applikációnk különböző részeit.

Ezt azért előnyös megtenni, mert konténerizált környezetben hatékonyan lehet horizontálisan skálázni kizárólagosan azokat a szolgáltatásokat, melyekre kiemelt terhelés összpontosul, ezzel elkerülve azt, hogy feleslegesen legyenek rendszererőforrások lefoglalva.

Konténerizált környezet kialakítása

A konténerizált környezet lehetővé teszi, hogy adott infrastruktúra szolgáltató esetében ne kelljen ügyfelenként külön-külön, saját kernellel rendelkező, nagy, úgynevezett overhead-del rendelkező virtuális gépeket létrehozni.

A konténerek ugyanis a hoszt operációs rendszer kernelét használják, így minimális, abszolút elhanyagolható overhead-del rendelkezik minden egyes újonnan indított konténer, ezzel szintén értékes erőforrásokat megtakarítva (rámutatva a virtuális gépek kerneljeinek operatív memóriában lefoglalt helyére és a rendszerhívások megspórolására).

Számolt adatok eltárolása az operatív tárban

Kiemelten fontos az összes (és nem feltétlenül csak a gyakran használt) adat operatív tárban történő eltárolása, főleg komplex számítások esetében, hiszen e módon kifejezetten

értékes CPU idő takarítható meg már rövidtávon is – mely eredményezi mind az adat gyorsabb elérési idejét, mind a rendszer jobb energiahatékonyságát.

Okos ütemezés applikáció szinten

A programozási nyelvek, illetve az elkészült programokat futtató futtatókörnyezetek lehetővé teszik bizonyos, leginkább aszinkron feladatok tetszőleges ütemezését.

Hasonlóan a hardver szintű feladatütemezés optimalizáláshoz, mindez megtörténhet operációs rendszer szinten is, mellyel hasonló pozitív hatások érhetők el a már leírtakhoz.

Event sourcing vagy Message Broker használata

Úgynevezett event sourcing-gal vagy message broker használatával elérhetjük, hogy az applikációnk csak abban az esetben írjon a gyorsítótárba, amikor valamilyen adat frissült, vagy újonnan létrejött.

Ez azért előnyös, mivel így nem kell fix TTL-lel automatikusan lejáratni a gyorsítótárban lévő adatokat, hogy aztán a gyorsítótárból kiürülve első alkalommal ismételten újra kelljen ezeket számolni, hanem kielégítő, ha a rendszer értesül azon adat módosulásáról, mely befolyásolja az eredetileg gyorsítótárazott adatot, és már előre elvégezni a számítást a háttérben, és végül így csak az adat módosulása esetén ténylegesen újraszámítani azt – mindezt megspékelve akár feladatütemezés-optimalizálással is, hogy egy elkerülhetetlenül nagy erőforrásigényes feladat lefuttatása ne terhelje le a rendszer egészét nem várt módon.

Korszerű futtatókörnyezet használata

Az előzőekben említettek alapján ismételten megemlítendő, hogy érdemes olyan programozási nyelvekben szoftvert fejleszteni, mely programok fordított változatát modern futtatókörnyezet hajtja végre – mely effektíven végzi a feladatütemezést és a szálkezelést, illetve megelégszik kevés operatív tárral is.

6. A MEGOLDÁSI MÓDSZER KIVÁLASZTÁSA, A VÁLASZTÁS INDOKLÁSA

A választott megoldási módszer a szoftveres úton történő optimalizáció. Meglátásom szerint több szempontból is előnyt jelent a szoftveres megvalósítás a hardveresnél.

Komplexitás

Elengedhetetlen megemlíteni, hogy a hardveres megvalósítás, illetve optimalizálás borzasztóan komplex feladat, már-már szinte lehetetlennek tűnő vállalkozás, melyek a processzor- illetve alaplapgyártók kezébe centralizálódnak.

Ezzel szemben a szoftveres optimalizációnak semmi sem képezi akadályát, csupán egy jó architektúra tervre, illetve szakértelemre van szükség.

Flexibilitás

Ahogyan általánosságban igaznak bizonyul az a kijelentés, hogy szoftver rétegben egyszerűbb optimalizálni, mint hardver rétegben, ez jelen esetben sincs másképpen.

Tekintve ugyanis, hogy hardveres optimalizáció esetén új processzorokba kellene beruházni a szerverparkok üzemeltetőinek, illetve a konzumer felhasználóknak, ez olyan jelentős költségekkel járna számukra, melyek eltaszítanák őket ezen vállalástól. Megemlítendő, hogy ezen processzorok előállítása, illetve maga a kutatás-fejlesztés is rengeteg közvetlen és közvetett energiabefektetéssel jár, mely jó esetben is csak hosszú távon lenne kifizetődő.

Emellett azt is ki kell emelni, hogy egy esetleges új utasításkészlet megjelenése esetén nem lehetne használni a már megírt programokat, hanem át kellene portolni azokat, vagy teljesen újakat kellene fejleszteni, mely szintén felbecsülhetetlen közvetlen és közvetett energiaköltséggel járna.

Mindezzel ellentétben azonban a szoftveres optimalizációk nem járnak semmiféle negatív mellékhatással, a végfelhasználó számára jó eséllyel észrevétlen is maradhat ezen optimalizációk véghezvitele – bár az előzőekben leírtak alapján, jó esetben még inkább pozitív vonatkozásai lennének mindennek.

Mérhetőség

Az optimalizáció mértékének mérését, kimutatását egyik esetben sem egyszerű feladat véghez vinni, azonban szoftver optimalizációk tekintetében profilozók nyomán erre még mindig nagyobb valószínűség adatik, mint hardveres optimalizációk esetében.

7. A RÉSZLETES SPECIFIKÁCIÓ LEÍRÁSA

A megvalósítás során egy olyan példaapplikáción keresztül kívánom bemutatni az előzőekben taglalt módszereket, azok összevetését a módszerek alkalmazása nélküli eredménnyel, melyből adódóan két megvalósítás is készül.

Az alapötlet egy olyan, televízió-vezérlő alkalmazás készítése, mely segítségével korlátok nélkül irányítani lehet LG gyártmányú okostévéket olyan extra funkcionalitás mellett, mint például statisztikák készítése a tévénézési szokásokról, automatizációs szabályok létrehozásának a lehetősége, és az elektronikus műsorújság közvetlen megtekintése a felhasználó szolgáltatója által nyújtott tévécsatornáknak megfelelően.

A specifikációt logikai szemszögből tekintve két részre osztom, amelyekben először az applikáció funkcionalitásának specifikálása kerül terítékre, míg az azt követő részében annak technikai értelemben vett specifikálása történik bemutatásra.

7.1 Funkcionalitás

Funkcionalitást tekintve az applikáció alapvető célja az, hogy minden olyan szolgáltatást biztosítson, mely egyébként elérhető az LG okostévékhez készített mobilapplikáción keresztül, viszont annál stabilabb módon.

Ezen alapvető funkciók körébe tartozik a csatornaváltás, hangerőmódosítás, a felhasználói menüben történő navigáció, illetve a készülék ki- és bekapcsolása.

Mindezen túlmenően extra funkciók is kívánatosak az applikációban.

Elsőként hasznos olyan statisztikai nyilvántartásokat vezetni, melyeket később a felhasználó érdeklődő szemmel értékelhet ki, és tanulságokat vonhat le belőle, hiszen ilyen okoskészülékek használata során rengeteg értékes adat keletkezik, melyet kár lenne vészni hagyni bármely nemű, jóhíszemű felhasználás nélkül.

Ilyen annak nyomon követhetősége, hogy a televízió mikor volt bekapcsolt állapotban, hogy a felhasználó milyen csatornákat mikor és meddig nézett, illetve, hogy ezen csatornák milyen kategóriákba tartoznak, azaz, hogy a felhasználó milyen témakörben mikor és meddig nézett különböző műsorokat.

Másodkörben egy egyébként szintén kívánatos, azonban az eredeti applikációból hiányzó funkció, az elektronikus műsorújság megtekintésének a biztosítása is egy fontos szempont, így a felhasználó rögtön, személyre szabottan, kifejezetten és csak a számára elérhető csatornákhoz azonnali betekintést nyerhetne az applikáción belül ehhez.

Harmadsorban egy egzotikus funkció, bizonyos feladatok automatikus végrehajthatóságának lehetősége is megemlítésre kerül, melynek segítségével automatikusan, akár dátum-idő párral, akár úgynevezett „cron" kifejezés segítségével válhatna lehetővé például bizonyos csatornák közti váltás automatizálása, vagy a készülék ki-be kapcsolása.

Utolsóként megemlítve nem elhanyagolható szempont a valósidejűség, vagy annak látszatának fenntartása, hiszen adott gyártói applikáció esetén, habár leginkább az első másodpercekben jellemzően, viszont késleltetéssel kerülnek reflektálásra a televízió távirányítójával véghez vitt állapotmódosítások, mint például a hangerő módosítása.

7.2 Technikai jellemzők

Néhány alapvető technikai követelményt is meghatározok, melyeknek történő megfelelés elengedhetetlen az applikáció energiahatékony működéséhez. Ezen pontok az alkalmazás architektúra dizájnjára, illetve annak komponenseire vonatkoznak.

Eseményvezéreltség

Az egyik legfőbb szempont, amely a technikai jellemzők terítékére kerülhet, az az eseményvezérelt programozás.

Az eseményvezérelt programozás lehetővé teszi, hogy bizonyos számítások akkor és csak akkor kerüljenek végrehajtásra, amennyiben egy adott esemény bekövetkezik, ezzel drasztikusan csökkenthetők a felesleges kalkulációk. Emellett, a következő pont predesztinációjaként, a gyorsítótárazást is megkönnyíti, mivel ahelyett, hogy tradicionális módon, bizonyos időközönként ürítenénk azt (például óránként), és utána rákényszerülnénk újbóli számítások lefuttatására még akkor is, ha a mögöttes adat egyébként ugyanaz, bizonyos esemény bekövetkeztekor tudhatjuk, hogy az adat biztosan változott, ekkor pedig befrissíthetjük azt a gyorsítótárban, így elérhetjük mind a felesleges számításokat és gyorsítótárba írásokat, mind pedig orvosoljuk a gyorsítótárazás elterjedt nehézségeit, név szerint az invalidálás problémáját.

Gyorsítótárazás

A gyorsítótárazás a másik legfőbb szempont. A gyorsítótárazás segítségével ugyanis szintén nagyon sok energia takarítható meg, adott egyrészt, hogy a felesleges újraszámítások megelőzhetők, mindemellett pedig rövidebb idő alatt kiszolgálható a kérés, mely szintén kevesebb energiafelhasználást eredményez. Emellett a memória modulok energiafelvétele szinte már-már elhanyagolható a processzor által felvett energiához képest, kifejezetten a komplex számítások esetén.

Többrétegű architektúra

A többrétegű architektúra lehetővé teszi a főbb koncepciók szétválasztását, ezzel átláthatóbbá és egyszerűbbé téve az alkalmazás működését.

Jelen alkalmazásra vonatkoztatva három fő réteg kerül meghatározásra: az első a kommunikációs réteg, mely a televízióval történő kommunikációért felelős, a második az optimalizációs réteg, melyben a hatékony feladatszámítás és a gyorsítótárazás történik, végül a harmadik réteg a prezentációs réteg, amely bizonyos extra optimalizációk mellett kontrollt nyújt a felhasználó számára.

Az első réteg feladata tehát, hogy felvegye a kapcsolatot a televízióval, kezelje az esetleges hibákat, mely a kapcsolat fenntartásáért felelős, majd pedig amennyiben az sikeres volt, az alkalmazás működése szempontjából az ehhez szükséges eseményekre feliratkozzon, majd ezeket továbbítsa az optimalizációs réteg, a második réteg felé, hogy aztán ott eseményvezérelten megtörténhessen az adatok számítása.

A második réteg feladata, hogy egyrészt amennyiben az alkalmazás működése szempontjából releváns eseményt kap az első rétegtől (az ezekre történő feliratkozást követően), elvégezze a kapcsolódó számításokat, majd ezeket eltárolja elsősorban a gyorsítótárba, másodsorban pedig egy külső adatbázisba is, amennyiben sem a gyorsítótárban nem érhető el adat, sem pedig a televízió nem érhető el, mert például kikapcsolt állapotban van – így ennek köszönhetően sohasem maradhat adat nélkül a rendszer, tehát ilyen szempontból, ebben a helyzetben függetlenné válhat a televízió aktuális állapotától. Mindemellett a második réteg az, mely közvetlenül kommunikál a felhasználóval – illetve a felhasználó számára transzparens módon, de közvetetten a televízióval, védve annak hardverét.

A harmadik réteg feladata, hogy oly módon reprezentálja a rendszer jelenlegi állapotát, hogy az valós időben történjen meg, illetve, hogy megakadályozza a komponensek felesleges „újra festését” (angol kifejezéssel élve a re-renderinget) mind pedig a felesleges HTTP-kérések küldését oly módon, hogy az egész rendszerről éppen ismert állapotinformációt eltárolja a memóriában. Szintén fontos, hogy itt is megjelenjen az eseményvezéreltség, hogy elkerülhető legyen az úgynevezett polling alkalmazása, azaz rendszeres, bizonyos rögzített időközönkénti HTTP-kérések küldözgetése a szerver felé annak érdekében, hogy vagy rögzítetten mindig befrissítsük az adatot, vagy ami még ennél is több számítással jár, hogy minden esetben összehasonlítsuk a jelenlegi állapottal, és különbözőség esetén frissítsük be azt. Ennek érdekében biztosítani kell, hogy e helyett a második réteg mindig értesítse a harmadik réteget (annak az eseményekre történő feliratkozását követően), amennyiben valamelyik adatban változás következett be, így a harmadik réteg csak és kizárólag akkor, amikor erre már feltétlenül szükség van, befrissítse az adatot.

Mikroszerviz architektúra

A mikroszerviz architektúra segítségével egy komplex rendszer átláthatóbbá tehető, hiszen még több elemre bonthatók a főbb koncepciók, annak érdekében, hogy azok ne keveredjenek, így tovább könnyítve a rendszer működésének átláthatóságát.

E mellett a szolgáltatások pontosan az azokra helyezkedő terhelés szerint válnak skálázhatóvá, így amennyiben csak adott szolgáltatásra van nagy igény (és annak nincs, vagy csak nagyon kevés függősége van más mikroszervizekre), akkor ez esetben szintén nagyon sok, feleslegesen lefoglalt erőforrás takarítható meg mind CPU magok, mind memória tekintetébe, ezzel csökkentve az applikációm energiafelhasználási lábnyomát.

8. A TERVEZÉS SORÁN VÉGZETT MUNKAFÁZISOK ÉS A TAPASZTALATOK LEÍRÁSA

8.1 Megvalósíthatósági tanulmány

A tervezés során fel kellett mérni, hogy mindenekelőtt megvalósítható-e egyáltalán egy ilyen rendszer, hiszen ehhez hasonló applikációk fejlesztésénél a hardver-szoftver integrációjának lehetősége, távlatai alapvetően meghatározzák, vagy éppen bekorlátolják a fejlesztő lehetőségeit.

Jelen esetben így a megvalósításhoz köthető kutatásom, a megvalósíthatósági tanulmány azzal indult, hogy a saját, LG típusú televíziómhoz kezdtem el keresni szoftverfejlesztői készletet, legyen az bármilyen programozási nyelvhez írt könyvtár. Ezt követően arra a megállapításra jutottam, hogy a lehetőségeim korlátozottak, de a feladat nem kivitelezhetetlen, csupán bizonyos nehézségeken kell túllendülni.

Ugyanis mélyre menőbben a megállapításom az volt, hogy egyrészt a gyártói dokumentáció erősen hiányos volt, sőt termék specifikus volt annak ellenére, hogy minden terméken ugyanazon operációs rendszer fut, másrészt pedig sem hivatalos, sem más olyan, nyílt forráskódú könyvtár nem állt a rendelkezésemre, mely aktívan karban lett volna tartva, így a későbbiekben kénytelen voltam régebbi, nyílt forráskódú projektek forráskódjából kinyert információdarabokra alapozni annak megtervezését, hogy pontosan milyen végpontokból milyen adatokat fogok tudni kinyerni, majd feldolgozni, és hogy hogyan biztosítható az, hogy valós időben tudjon folyni az adat.

Mindezt figyelembe véve elhatároztam, hogy technológiafüggetlen maradok azzal, hogy a televízióval történő kommunikációt külső féltől származó könyvtár nélkül fogom megvalósítani, és hogy inkább rendszerezem a számomra elérhető információkat az ismert végpontokról, majd kiértékelve azokat megtervezem, hogy pontosan melyik végpontból milyen adatokat fogok kinyerni annak érdekében, hogy az alkalmazás az elvártak szerint tudjon működni, így el tudom azt kerülni, hogy amennyiben módosul valamelyik a televízió által nyújtott végpont, és az adott könyvtárt fejlesztő személy félbeszakítja annak további fejlesztését, az alkalmazásom ne működjön többé, hiszen a könyvtárat az azt karbantartó közreműködése nélkül csak használni, szerkeszteni nem tudom, még ha nyílt forráskódú is.

Az előzőhöz kapcsolódva nyílt forráskódú projektekből megismertem, hogy a televízióval WebSocket protokollon keresztül lehet kommunikálni, illetve, hogy ténylegesen van lehetőség akár adatfolyam kinyerésére is, hiszen mind a protokoll, mind a televízió hardvere és szoftvere támogatja ezt a lehetőséget.

Mindezeken felül komoly problémát jelentett az elektronikus műsorújság megvalósíthatósága, ugyanis habár mint az a nyílt forráskódú projektek elemzéséből kiderült, hogy alapvetően alkalmazásprogramozási interfész (API) szinten definiálva van egy bizonyos végpont, amelyre küldött megfelelő kérés esetén a televízió szoftvere visszaadná az általa a tv-szolgáltatótól érkező elektronikus műsorújságot, ez mégsem felel meg a valóságnak, ugyanis a visszaadott tömb üres. Így alternatív módszerek után kellett kutatnom, majd szintén arra a döntésre kellett jutnom, hogy egy egyedi megoldás fejlesztésére lesz szükség ennek a problémának a kikerülése érdekében.

8.2 Elektronikus műsorújság megvalósíthatósága

Az előző pontban említett okokból kifolyólag egy olyan rendszert kellett megterveznem, mely a televíziótól ténylegesen kinyerhető adatok segítségével képes azon adatokat előállítani, melyek aztán ahhoz szükségesek, hogy a felhasználó számára szintén transzparens módon olyan módon legyen megtekinthető a műsorújság, mintha közvetlenül a televízión tekintené meg azokat.

A csatornalista tudatában bizonyos külső forrásokból kinyerhető vagy „letapogatható” az adott csatornához tartozó műsorújság, azzal a kitétellel, hogy ezen külső források másként hivatkoznak ugyanazon csatornákra, így majdan egy összerendelő algoritmus megírására is szükség lesz.

8.3 Tech stack kiválasztása

Elsődleges szempont volt a hatékony programvégrehajtás, tehát hogy a feladatok számítása, elvégzése minél hatékonyabban történjen, ez minél kevesebb energia felhasználásával menjen végbe.

Ezt, a fejlesztés elkezdésekor a birtokomban lévő tapasztalatokat, tudást, illetve a megvalósításhoz szükséges könyvtárak elérhetőségét figyelembe véve arra a következetes döntésre jutottam, hogy az első rétegben TypeScript-et, a második rétegben

Java-t és Rust-ot vegyesen, a harmadik rétegben pedig – elkerülhetetlenül – ismételten TypeScript-et fogok alkalmazni a megvalósításhoz.

A specifikáció alapján felvázolt architektúrából kifolyólag a felhasználói kérések legtöbbje a második rétegig fog csupán eljutni, ahol a kiválasztott programnyelvek, és azok futtatási környezete miatt a leghatékonyabb kiszolgálás történhet, ugyanis mind a Rust, mind a Java az energiafelhasználás szempontjából leghatékonyabb programnyelveknek minősülnek.

Time & Memory	Energy & Time	Energy & Memory	Energy & Time & Memory
C • Pascal • Go	C	C • Pascal	C • Pascal • Go
Rust • C++ • Fortran	Rust	Rust • C++ • Fortran • Go	Rust • C++ • Fortran
Ada	C++	Ada	Ada
Java • Chapel • Lisp • Ocaml	Ada	Java • Chapel • Lisp	Java • Chapel • Lisp • Ocaml
Haskell • C#	Java	OCaml • Swift • Haskell	Swift • Haskell • C#
Swift • PHP	Pascal • Chapel	C# • PHP	Dart • F# • Racket • Hack • PHP
F# • Racket • Hack • Python	Lisp • Ocaml • Go	Dart • F# • Racket • Hack • Python	JavaScript • Ruby • Python
JavaScript • Ruby	Fortran • Haskell • C#	JavaScript • Ruby	TypeScript • Erlang
Dart • TypeScript • Erlang	Swift	TypeScript	Lua • JRuby • Perl
JRuby • Perl	Dart • F#	Erlang • Lua • Perl	
Lua	JavaScript	JRuby	
	Racket		
	TypeScript • Hack		
	PHP		
	Erlang		
	Lua • JRuby		
	Ruby		

8.1 ábra Különböző programozási nyelvek összevetése futási idő, memóriahasználat, illetve becsült energiafelhasználás szempontjából [8]

Ugyanis előzőekben említettekre visszautalva elmondható, hogy adott hardveren, mely adott fogyasztással rendelkezik annak központi feldolgozóegysége és operatív tára nyomán, az a szoftver fog kevesebb energiát felhasználni, mely hamarabb lefut a rendszeren, hiszen ahogyan azt fizikából tanultuk, J = W x S, azaz az energia a teljesítmény és az idő szorzatával megegyező értékű.

Joggal felmerülhet a kérdés azonban, hogy milyen okból kifolyólag döntöttem a Java, mint programozási nyelv használata mellett egy olyan rétegben, mely kifejezetten az optimalizációra helyezi a hangsúlyt, ismervén nagy memóriaigényét. Szerencsére elmondható azonban, hogy a nagyobb memóriaigény ezen, úgynevezett „virtual machine” nyelvek esetén sem korrelál egyenesen arányosan az energiafogyasztással (szem előtt tartva az operatív tár energiahasználatát), sőt közel elhanyagolható, fontosabb az ugyanis, hogy a különböző futtatási környezetek miképpen használják azt. [9]

A 8.1 ábrán látható hatékonysági táblázat alapján megfigyelhető, hogy ami a legalsó réteget illeti, itt a nem túl hatékony, úgynevezett interpreted TypeScript nyelv került kiválasztásra a fejlesztési sebesség növelése érdekében, azonban itt szintén meg kell jegyezni azt, hogy a második réteg a legtöbb kérést elrejti a legalsó réteg alól, így ezen tényező gyakorlatilag ismételten elhanyagolható.

8.4 Architektúra

Az architektúra megtervezése során további konkrét, az implementáció során használt technológiák kerültek kiválasztásra.

Technológiák megválasztása

A második rétegben webes keretrendszerként kiválasztásra került a Spring WebFlux keretrendszer, mely segítségével aszinkron, (az adott szálat) nem blokkoló program valósítható meg, így az összes szál teljes mértékben kihasználhatóvá válik akkor is, ha adott szálon futó folyamat várakozni kényszerül, mely növeli a hatékonyságot és a végrehajtási időt. Ugyanez igaz folyamatszinten: amennyiben egy feladat várakozni kényszerül, vagy az függ egy másik folyamat eredményétől, addig végrehajthatunk olyan feladatot, mely ilyen feladatokat utasításszinten sorrendileg követ, azonban nincsen olyan függő bemenete, amely adott időpillanatban ne állna rendelkezésére.

E mellett úgynevezett „Message Queue” is az architektúra részét képezi, melyen keresztül az első rétegben található interfész értesítheti a második réteget különböző, a rendszer működésének szempontjából érdekes események bekövetkeztéről, így biztosítva a valós idejű feldolgozást és adatfrissességet. Konkrét megvalósításként az Apache nyílt forráskódú Kafka megoldását választottam, mivel olyan népszerű alkalmazások implementációjához is használják, mint például a Spotify, mely szintén bizonyítja azt, hogy ez az egyik, ha nem a legnagyobb üzenet-áteresztőképességgel rendelkező szoftver. Továbbá nem volt elhanyagolható szempont az alacsony késleltetés biztosítása sem, annak érdekében, hogy a szoftver minél reszponzívabb legyen, a Kafka pedig 2 milliszekundumos késleltetést ígér, mely szintén kiemelkedő tulajdonsága.

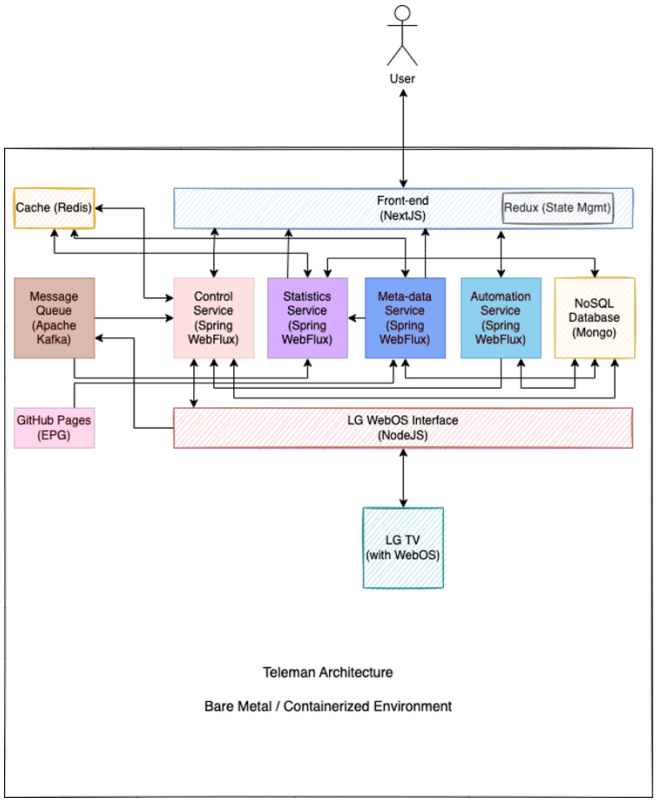
Gyorsítótárként két opció jöhetett szóba: az egyik a Memcached, a másik a Redis. Azonban mára a szakmán belül általános igazságként a Redis sokkal népszerűbb és

kiforrottabb lett vetélytársánál, így egyértelmű volt, hogy Redis-t fogok gyorsítótárként használni.

Adatbázisként relációs adatbázis-kezelő helyett az egyszerűbb és I/O műveletekben gyorsabb, strukturálatlan, úgynevezett NoSQL adatbázis implementációt választottam, mely a jelenleg legnépszerűbb MongoDB-re esett.

Ami érdekesség lehet viszont, az a harmadik réteg technológiai megválasztása: itt úgy döntöttem, hogy egyrészt NextJS-t választok az applikáció felületének megalkotásához, továbbá még úgynevezett state management könyvtár használatát is rendszeresítettem, hogy azon adatokat, melyeket egyszer már elkértem a második rétegtől, ne kelljen oldal vagy lapváltásokkor újra megtennem, még akkor is, ha azok egyébként közvetlenül a gyorsítótárból elérhetőek lennének, ezzel csökkentve a felesleges HTTP-kérések számát illetve a második réteg szolgáltatásaira nehezedő terhelést. Itt nemes egyszerűséggel azért esett a Redux Toolbox könyvtárra a választásom, mert ez az, amellyel már dolgoztam és volt tapasztalatom, illetve mert ez az egyik legszéleskörűbben használt és funkciókkal rendelkező könyvtár.

Komponensek relációja



8.2 ábra A Teleman projekt architektúrája [saját szerkesztés]

A 8.2 ábrán látható, hogy a 3, az előzőekben már említett réteg elkülönül egymástól, ezek egymáson keresztül tudnak kommunikálni.

Az első rétegben csupán az interfész szerepel, mely a televízióval közvetlen formában történő kommunikációért felelős, és TypeScript-ben kerül megvalósításra, futtatókörnyezete NodeJS, azaz a V8-as motor.

A második rétegben mikroszervizek szerepelnek, melyek az alkalmazás funkcióiként különülnek el egymástól, a minél jobb skálázhatóság érdekében (tehát ha csak adott komponensen nagy a terhelés, elég legyen végtére csak ahhoz az alkalmazáshoz összességében több erőforrást rendelni, ne kelljen az egész alkalmazást skálázni, mely összeségében több erőforrást allokálna). Ezek, ha a szükség úgy kívánja, akár egymással, akár az interfészen keresztül a tévével is kommunikálhatnak, illetve adatot szolgáltatnak a prezentációs réteg, azaz a végfelhasználó felé, valós időben, szerver által küldött eseményeken („Server-Sent Events”) keresztül. Célja, hogy az interfésztől a Kafka-n keresztül valós időben kapott adatokat rögtön elmentse a gyorsítótárba, a szükséges

számításokat elvégezze, illetve azok eredményeit szintén elmentse a gyorsítótárba, hogy amennyiben bármelyik másik komponensnek, vagy a prezentációs rétegnek szüksége lenne az adott adatra, akkor egyrészt minél gyorsabban, tulajdonképpen az operatív tárból (a lassú adatbázis helyett) kiszolgálható legyen, másrészt kisebb terhelést helyezzen a többi komponensre, kiemelve magát az interfészt (mely kevéssé hatékony technológiával készült) illetve a televíziót (annak viszonylagosan gyenge hardvere miatt).

A harmadik réteg a már az előzőekben említettek alapján egy NextJS felületből áll, melyhez egy nagyobb Redux tár társul, ahova a szerver által küldött eseményeken keresztül fogadott adatok, illetve az egyébként elkért adatok kerülnek eltárolásra annak érdekében, hogy ezen adatokat soha többé ne kelljen elkérni, hiszen adatmódosulásnál erről valós időben úgyis értesülni fog.

8.5 Komponensek megtervezése

Az applikáció megtervezése, a komponensek megválasztása után az adott komponensek részletesebb megtervezése következett. Megterveztem a szükséges adatmodellek struktúráját, illetve a szükséges interfészeket, melyek absztrakciós szinten definiálják az applikáció működését, és a komponensek együttműködését.

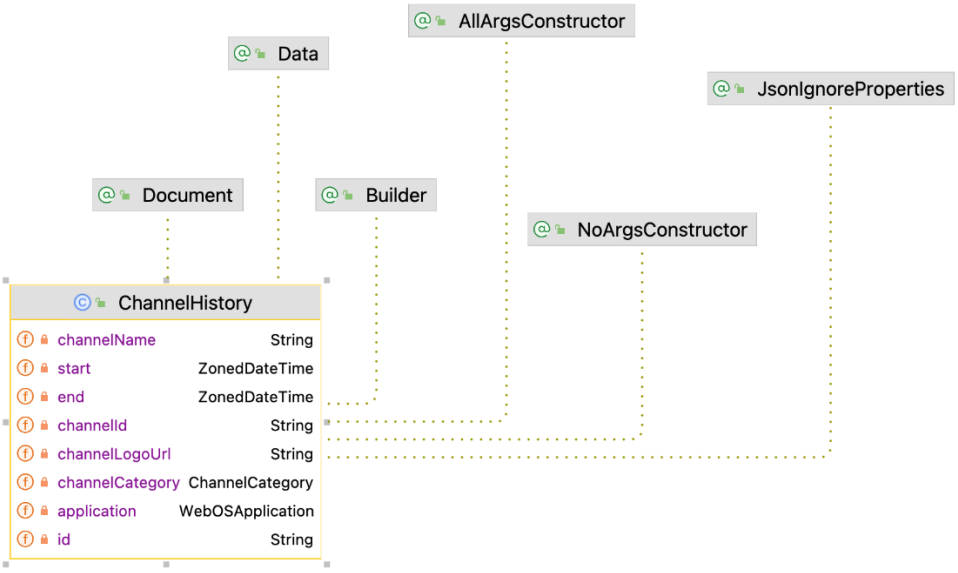
A legfontosabb komponenssel kezdve, a TV-t vezérlő szolgáltatáson belül azok a funkcionalitások csoportosulnak, melyek a tévé általános vezérlését látják el, azaz az olyan funkciók, mint a csatornaléptetés, hangerőállítás, és az applikációk kezelése.

8.3 ábra A vezérlő szolgáltatás tervezett interfészei [saját szerkesztés]

A 8.3 ábrán látható, hogy funkcionalitásonként kerül megtervezésre egy-egy interfész. A „MediaControlService” interfész olyan metódusokat tartalmaz, mint a hangerő fel-le

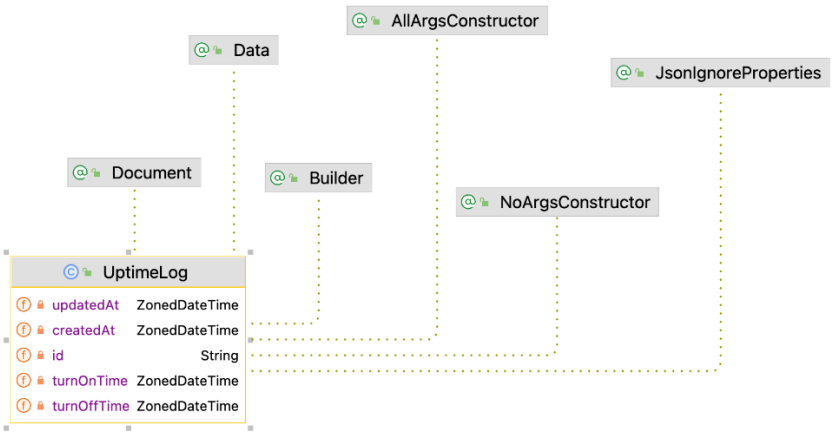
állítása, közvetlen adott értékre állítása, lekérése vagy a hangerő megváltozásának lekezelése (a felhasználói felület értesítése, és befrissítése), míg a „TvControlService” ennek a csatornákra vonatkoztatott megegyezője. A „SystemControlService” tartalmazza a TV ki-be kapcsolásának lehetőségét és jelenlegi állapotának lekérését, illetve a jelenleg a televízión futó szoftver jellemzőinek lekérését. A „MetaService” egy olyan, minden szolgáltatásban megtalálható ugyanazon interfész, mely előír egy olyan metódust, ami bizonyos adatokat, például, hogy hányas verziójú az adott mikroszolgáltatás, elküldi a kérő fél felé. Az „EventSource” interfész kell, hogy felelős legyen az események továbbításáért a prezentációs réteg felé. A „WebApplicationControlService” egy olyan interfész, mely az olyan, alkalmazásokkal kapcsolatos műveleteket foglal magában, mint amilyen az applikációk listájának lekérése, az úgynevezett indítási pontok lekérése, mely olyan információkat foglalnak magukban a telepített alkalmazásokról, melyek alapján meg lehet nyitni azokat, illetve természetesen lehetővé is teszi alkalmazások megnyitását. Végül a „WebChannelMetadataService” egy olyan interfész, mely a metaadatokat szolgáltató mikroszervizhez fordulást határozza meg: ez elsősorban azért szükséges, hogy a csatorna objektumokat fel tudjam tölteni az adott csatorna logójára mutató linkekkel annak érdekében, hogy a prezentációs rétegben a statisztikák mellett meg tudjam jeleníteni azokat.

A második legfontosabb komponens, a statisztikákat számoló és azokat szolgáltató mikroszerviz megtervezése következik.



8.4 ábra A "ChannelHistory" entitás [saját szerkesztés]

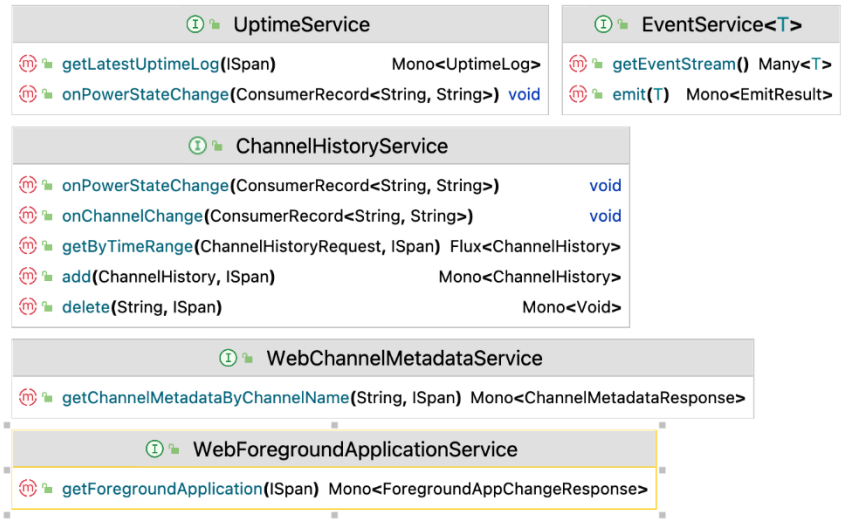
A 8.4 ábrán látható a szolgáltatás egyik entitása, mely magában foglalja, hogy egy adott csatornát meddig néztünk. Tartalmazza a csatorna nevét, azonosítóját, a logójára mutató linket, a csatorna kategóriáját, a kezdő- és végdátumokat, illetve amennyiben applikációról, nem csatornáról beszélünk, akkor a csatorna adatai helyett az applikáció azonosítóját.



8.5 ábra Az "UptimeLog" entitás [saját szerkesztés]

A 8.5 ábrán látható a mikroszerviz másik entitása, mely magában foglalja azokat az információkat, hogy a televízió mikor került be- illetve kikapcsolásra. Ezen adatok

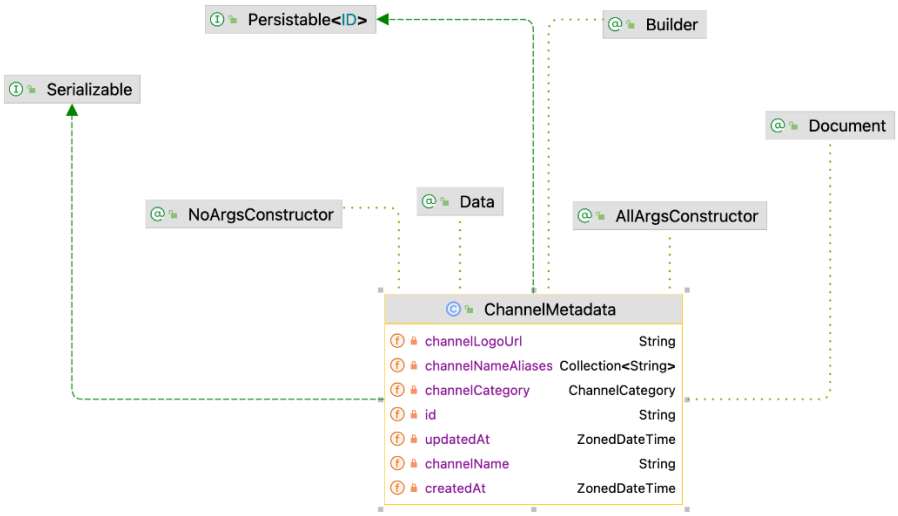
alapján a következőkben bemutatott interfészekben deklarált metódusokkal ki lehet számítani és a prezentációs rétegben ábrázolni lehet, hogy mely időtartományokban és hány percig volt bekapcsolva a televízió.



8.6 ábra A statisztikákat számoló szolgáltatás interfészei [saját szerkesztés]

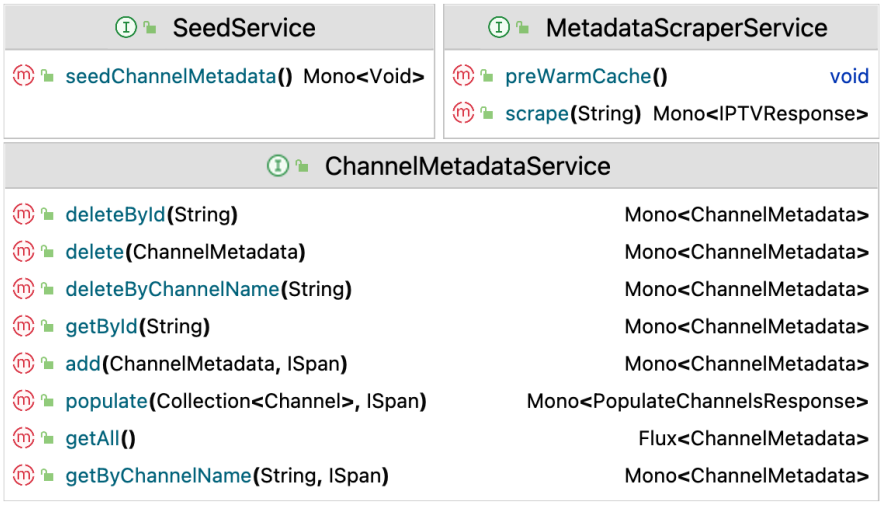
A 8.6 ábrán látható a statisztikákat számoló mikroszolgáltatás 5 interfésze. Az „UptimeService” interfész deklarálja egyrészt a metódust, mely lekezeli a tévé állapotában bekövetkező változásokat (azaz létrehoz egy „UptimeLog” entitást a bekövetkezett esemény alapján, és értesíti erről a prezentációs réteget), illetve azt a metódust, mellyel ezen entitások lekérdezhetők. A „ChannelHistoryService” egy olyan interfész, mely a „ChannelHistory” entitásokkal történő műveleteket foglalja magában: hozzáadás az adatbázishoz, törlés az adatbázisból, olyan entitások keresése, melyek egy bizonyos időszakban kerültek létrehozásra, illetve magában foglalja azt a metódust, mely a csatorna megváltoztatásakor tüzelő eseményt kezeli le, ennek során új „ChannelHistory” objektumot létrehozva. Emellett a „WebForegroundApplicationService” interfész egy olyan metódust foglal magában, melyen keresztül az alkalmazás el tudja kérni a jelenleg előtérbe lévő alkalmazást, amennyiben a „ChannelHistory” alkalmazás mezőjébe valamilyen alkalmazás azonosítót kell írnia az applikációnak, például, ha a kiváltott eseményben, melyet az „onChannelChange” metódus lekezel, az szerepel, hogy alkalmazást nyitottunk meg, nem pedig csatornát váltottunk. A másik két interfész funkcionalitását tekintve ekvivalens az vezérlő szolgáltatásban lévő, azonos nevű interfészekkel.

A harmadik komponens terve, a csatornákról metaadatokat nyújtó szolgáltatás elsődleges feladata különböző olyan HTTP kérések indítása lesz, melyekre adott válaszok alapján, amennyiben a külső szolgáltatás által küldött csatornanév és az ISP által küldött csatornanév megegyezik, akkor visszaadja például az adott csatorna logójára mutató linket, és ezeket az információkat el is tárolja későbbre az adatbázisba.



8.7 ábra A "ChannelMetadata" entitás [saját szerkesztés]

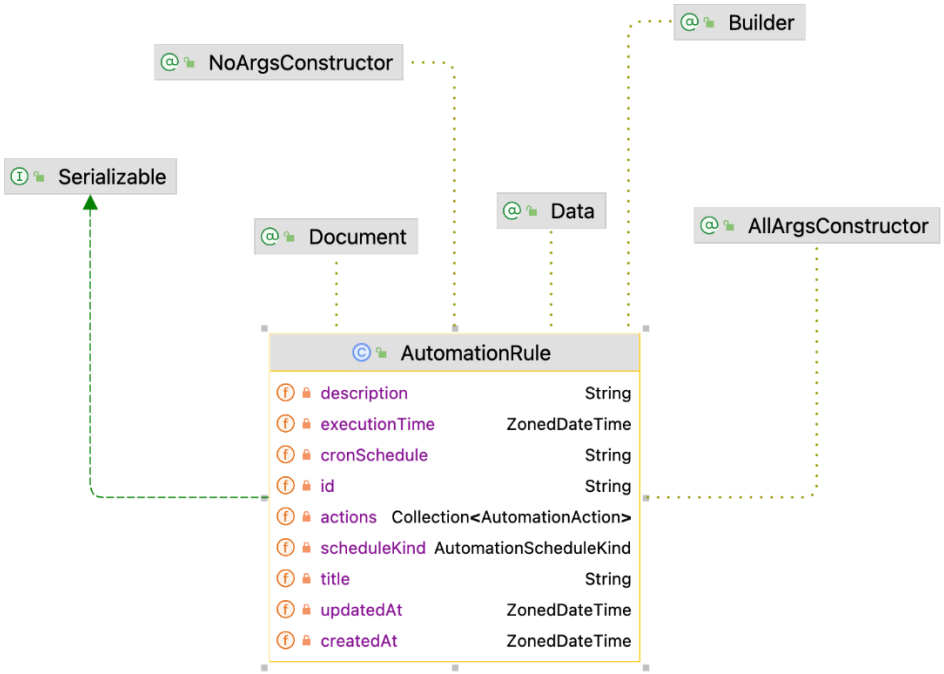
A 8.7 ábrán azon entitás látható, mely magában foglalja az adott csatorna nevét, melyhez a metaadatok tartoznak, a csatorna logójára mutató linket, a csatorna kategóriáját, illetve olyan alternatív megnevezéseit a csatornának, melyek szinonimájaként értelmezhetők az adott csatorna az ISP által elküldött nevének, ezzel megkönnyítve a külső szolgáltatások által küldött csatornanévnek és az ISP által elküldött csatornanévnek az összepárosítását, hiszen a külső szolgáltatások az Ő általuk ismert csatornanévhez kötik a csatornához tartozó metaadatokat.



8.8 ábra A metaadatokat nyújtó mikroszerviz interfészei [saját szerkesztés]

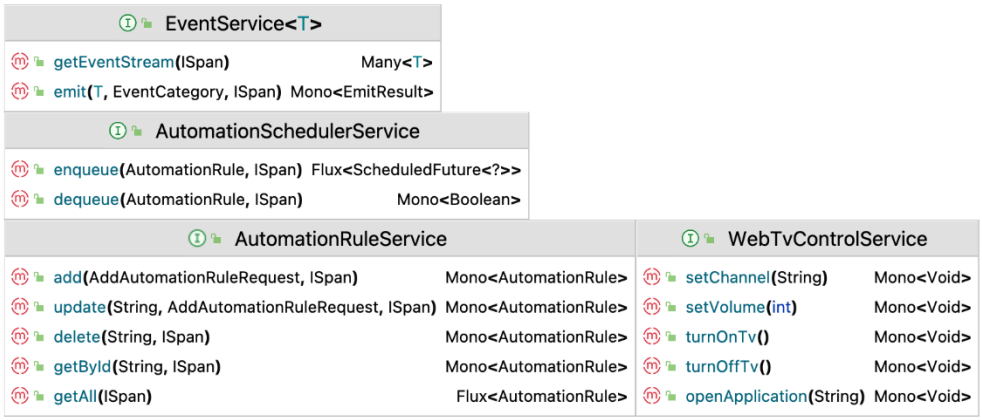
A 8.7 ábrán látható az a 3 interfész, mely ennek a mikroszerviznek az összes tervezett funkcionalitását magába foglalja. Tartalmaz egy „SeedService” interfészt, mely által deklarált metódus azért lesz felelős, hogy az előre (a kódba) felvitt metaadatokat az alkalmazás indulásakor beleírja az adatbázisba. A „MetadataScraperService” egyrészt deklarál egy olyan metódust, mely az összes, adott országkódhoz tartozó metaadatot visszaadja különböző külső szolgáltatásokon keresztül, míg másik metódusa a gyorsítótár feltöltéséért felel. A „ChannelMetadataService” a listázását és törlését teszi lehetővé az adatbázisban eltárolt „ChannelMetadata” objektumoknak, illetve egy olyan metódust is deklarál („populate”), mely a tévé által visszaadott csatornalistát feltölti az ismert metaadatokkal a gyorsítótárból.

A negyedik mikroszolgáltatás tervében szintén szerepel egy entitás, mely az automatizálási szabályokat foglalja magában.



8.9 ábra Az "AutomationRule" entitás [saját szerkesztés]

A 8.9 ábrán látható az az objektum modell, mely egy automatizálási szabály adatait foglalja magában. Magában foglal különböző metaadatokat, mint például cím és leírás, de magában foglalja az ütemezés típusát (hogy úgynevezett „cron” kifejezések alapján történő rendszeres ütemezésről, vagy egyszeri, adott időpillanatban történő végrehajtásról van szó), előbbi esetén a „cron” kifejezést az ütemezéshez, illetve a végrehajtandó műveletek listáját, melyek lehetnek a tévé bekapcsolása, kikapcsolása, a hangerő és a csatornák léptetése, illetve alkalmazás megnyitása.



8.10 ábra Az automatizálási mikroszerviz interfészei [saját szerkesztés]

A 8.10 ábrán látható az előző szolgáltatásokból ismert, funkcionalitását tekintve azokéhoz ekvivalens „EventService” interfész, az „AutomationSchedulerService” interfész, mely a feladatok ütemezését és azok ütemezésének visszavonását tartalmazza, az „AutomationRuleService”, mely az „AutomationRule” entitáshoz tartalmazó listázás, létrehozás, frissítés és törlés műveleteket tartalmazza, illetve a „WebTvControlService”, melyen keresztül a szolgáltatás a vezérlő szolgáltatáshoz fordulhat annak érdekében, hogy az automatizálási szabályokban definiált műveleteket végre tudja hajtani.

9. A MEGVALÓSÍTÁS LEÍRÁSA

A Proof of Concept verzió lefejlesztését követően, az abban a verzióban bevált minták szerint kezdem el fejleszteni a második réteg többi mikroszervizét is, azaz a statisztikai adatokat számoló szervizt, az ennek munkáját asszisztáló metaadat szolgáltató szervizt, illetve egy későbbi ütemben az automatizálási szabályok deklarálását lehetővé tévő szervizt.

A fejlesztés során, a már az előzőekben említettek szerint kialakul egy minta, mely a legtöbb metódusban megfigyelhetővé válik; először is ezt kívánom egy pár bekezdésen keresztül bemutatni az alábbiakban.

A kód követi a Robert C. Martin által lefektetett [10], úgynevezett „Clean Code” elveket. Ily módon közérthetőbb a kívülálló személy számára is, hogy éppen miért felelős az adott kódrészlet, nemmellesleg igény szerint a refaktorálást is egyszerűbbé teszi; ez az egész kódbázist jellemzi. A kód működését tekintve a csatornaváltáskor kiváltódó esemény lekezeléséért két metódus felelős. Az esemény közvetlen lekezelője az „onChannelChange” metódus, mely egyrészt meghívja a „handleChannelChange” privát metódust, másrészt amennyiben annak végrehajtása során hiba történik, lekezeli azt (jelen esetben tovább dob egy egyedileg létrehozott hibát). A „handleChannelChange” metódus dekódolja a Kafka-n keresztül fogadott üzenetet (mely egyébként egy JSON objektum string formátumban), deszerializálja azt egy osztály objektumba, majd a számításhoz szükséges adatot megpróbálja kiolvasni a gyorsítótárból. Amennyiben az ott nem elérhető, az adatbázisra hagyatkozik, és a gyorsítótár helyett onnan olvas. Ezt követően ténylegesen elvégződik a szükséges számítás, az esemény teljes kiértékelése, majd a frissen számított adat kiírásra kerül a gyorsítótárba, illetve a prezentációs rétegben éppen eseményekre figyelő kliensek értesítésre kerülnek a már emlegetett Server-Sent Events segítségével, hogy szinte késleltetés nélkül befrissíthessék a rendszer jelenlegi állapotát.

```
4 usages
private Mono<Sinks.EmitResult> notifyListeners(ChannelHistory channelHistory, ISpan parentSpan) {
    final ISpan span = parentSpan.startChild( operation: "notify-listeners");
    try {
        return Mono.fromCallable(() -> EventMessage.fromChannelHistoryChange(channelHistory)) Mono<EventMessage<...>>
                .flatMap(this.channelHistoryEventService::emit) Mono<EmitResult>
                .publishOn(Schedulers.immediate())
                .doOnTerminate(span::finish);
    } catch (Exception e) {
        span.setStatus(SpanStatus.INTERNAL_ERROR);
        span.setThrowable(e);
        throw e;
    }
}
}
```

9.1 ábra A prezentációs réteg értesítéséért felelős egyik kódrészlet [saját szerkesztés]

Két további fontos tényezőt is érdemes megemlíteni az általános mintákat illetően, melyhez a 9.2 ábrát hívom segítségül: az egyik az ütemezés (illetve jelen esetben annak hiánya), és a nyomkövetés. Az említett ábrán, a mintakód 462-es sorában látható, hogy a feladat végrehajtása azonnali módon kerül kérelmezésre, ami annyit jelent, hogy az egyszerűen egyáltalán nem kerül ütemezésre, hanem ugyanazon szálon fog végrehajtódni, amelyen az ezt a metódust meghívó metódus eddig futott, így minimalizálva a késleltetést, hogy a prezentációs réteg minél hamarabb értesülhessen az esemény bekövetkeztéről, illetve eljusson hozzá az éppen legfrissebb adat. [1]

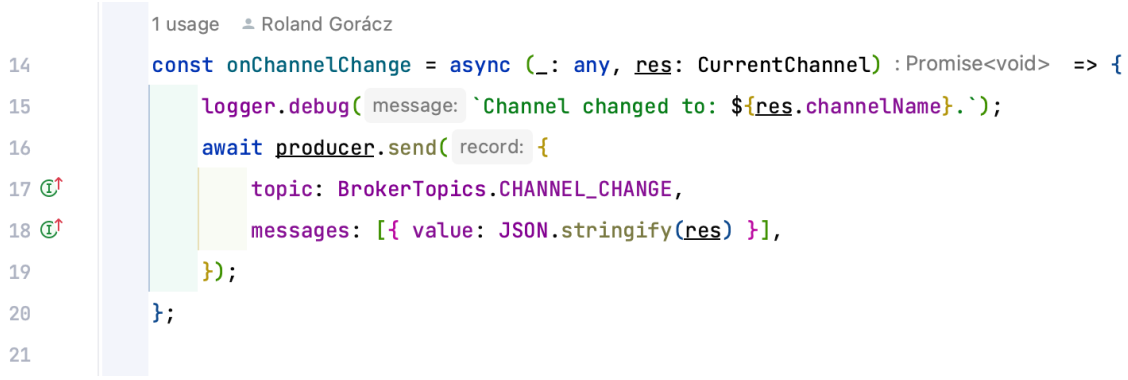
9.1 Interfész megvalósítása

Az interfész megvalósítás viszonylag egyszerű, nem igazán tartalmaz üzleti logikát, tulajdonképpen csupán egy köztes pontként szolgál a második réteg és a televízió hardvere között, továbbítva tévé oldalról az eseményeket a második réteg számára Kafka-n keresztül, míg második rétegtől fogadja a kéréseket és közvetlenül továbbítja azokat a televízió felé bármilyen egyéb feldolgozás jelenléte nélkül (tehát például itt nem történik gyorsítótárazás).

```
import ...

process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0";

const producer : Producer  = initializeKafka();
5+ usages  Roland Gorácz
export let connection : lgtv  = lgtv(config.connectionConfig);
registerEventListeners(connection, producer);

const app: Express = express();
const port: number | string = process.env.PORT ?? 5000;
initializeSentry(app);
registerMiddlewares(app);
registerRouterHandlers(app);
registerSentryErrorHandler(app);

Roland Gorácz *
app.listen(port,  callback: () : void  => {
  logger.info( message: `Listening on :${port}`);
});

2 usages  Roland Gorácz
export const resetConnection = (newConnection: lgtv) : void  => {
  connection = newConnection;
  registerEventListeners(connection, producer);
};
```

9.2 ábra Az interfész applikáció belépési pontja [saját szerkesztés]

A 9.3 ábrán látszik, hogy az applikációban nem történik más, mintsem a Kafka kapcsolat inicializálása, a kapcsolat felépítése a televízióval WebSocket Secure-on keresztül (ez az egy lépés egy könyvtár segítségével történik), a különböző eseményekre történő feliratkozás, a Sentry inicializálása (hibakereséshez és nyomkövetéshez), majd az úgynevezett middleware-ek, API végpontok regisztrálása, a Sentry hibakezelőjének inicializálása, végül pedig a HTTP szerver elindítása.



```
1 usage  Roland Gorácz
const onChannelChange = async (_: any, res: CurrentChannel) : Promise<void>  => {
    logger.debug( message: `Channel changed to: ${res.channelName}.`);
    await producer.send( record: {
        topic: BrokerTopics.CHANNEL_CHANGE,
        messages: [{ value: JSON.stringify(res) }],
    });
};
```

9.3 ábra Csatornaváltás esemény továbbítása Kafka-n keresztül a második réteg felé [saját szerkesztés]

A 9.4 ábrán egy olyan kódrészlet látható, amely akkor kerül végrehajtásra, ha a televízió WebSocket Secure (a továbbiakban WSS) socket-en keresztül arról értesíti az interfészt, hogy csatornaváltás történt. Az üzenetben kapott adatot, azaz annak a csatornának az adatait, melyre váltott a felhasználó, továbbküldi Kafka-n keresztül a második réteg felé. Ez nem történik másképpen az összes többi esemény esetében sem, mint amilyen a hangerő állítás, mely a hangerő le- vagy felfelé állításakor tüzel, a be-kikapcsolási állapot változása, mely bekapcsoláskor, kikapcsoláskor, vagy bizonyos képernyővédő funkciók automatikus aktiválásakor vagy deaktiválásakor tüzel, vagy az éppen előtérben lévő applikáció megváltozása, mely applikációk megnyitásakor, bezárásakor tüzel.

```
55      connection.on( event: "connect",  listener: () : void  => {
56          logger.info( message: "Connected to TV.");
57
58          connection.subscribe(WebOSStreams.CHANNEL_CHANGE_STREAM, onChannelChange);
59          logger.debug( message: `Listening to channel changes...`);
60
61          connection.subscribe(WebOSStreams.VOLUME_CHANGE_STREAM, onVolumeChange);
62          logger.debug( message: `Listening to volume changes...`);
63
64          connection.subscribe(
65              WebOSStreams.POWER_STATE_CHANGE_STREAM,
66              onPowerStateChange
67          );
68          logger.debug( message: `Listening to power state changes...`);
69
70          connection.subscribe(
71              WebOSStreams.FOREGROUND_APP_CHANGE_STREAM,
72              onForegroundAppChange
73          );
74          logger.debug( message: `Listening to foreground application changes...`);
75      });
76  };
```

9.4 ábra Különböző eseményekre történő feliratkozás interfész szinten [saját szerkesztés]

A 9.5 ábrán látható, hogy a televízióhoz történő sikeres csatlakozást követően az ábrán látható helyen történik a közvetlen feliratkozás az említett eseményekre, oly módon, hogy a „connection” objektumon, mely a 9.5. ábrán látható kódrészlet 15. sorából származik, meghívjuk a „subscribe” metódust, melynek első paramétere egy WSS végpont, második paramétere pedig egy metódus, mely az adott esemény lekezeléséért felelős (mely minden esetben csupán az előzőekben említettek szerint az adott esemény továbbítása Kafka-n keresztül).

```
Roland Gorácz
router.get( path: "/", handlers: <T extends ChannelList>(_: Request, res: Response): Response<T> => {
  const channels :T  = sendRequestToTv<T>(connection, WebOSEndpoints.CHANNEL_LIST);
  return res.status( code: 200).json(channels);
});
```

9.5 ábra Egy tipikus API végpont megvalósítása [saját szerkesztés]

Végül a 9.6 ábrán egy tipikus API végpont implementációja látható, ahol szintén nem történik más, mintsem a megfelelő végpontra történő kérés továbbítása a televízió felé, majd a válasz gzip-pel tömörített továbbítása a kérő fél felé.

9.2 A vezérlő szerviz („control service”) megvalósítása

A control service egy Java-s, Spring WebFlux-os mikroszerviz, mely talán a második legkomplexebb komponense az alkalmazásnak, azonban időrendileg ez volt az első komponens a második rétegből, mely elkészült.

Fontos tulajdonsága, hogy nagyban a Project Reactor könyvtárra épül, mely lehetővé teszi a kérések aszinkron, (az adott szálat) nem blokkoló módon történő kiszolgálását, így nagyfokú párhuzamosítást, majd végtére nagyobb hatékonyságot és gyorsabb feldolgozást elérni. A könyvtár továbbá lehetőséget biztosít a feladatok különböző módon történő ütemezésére, hogy azok a lehető leghatékonyabban kerüljenek majd feldolgozásra, az adott feladat jellegétől függően (tehát hogy a feladat I/O, vagy processzor igényes-e inkább).

Feladatkörébe tartozik minden olyan kérés kiszolgálása, mely a televízió irányítására szolgál, ezek körébe bármilyen olyan utasítás beletartozik, mint például a hangerőszabályozás, a csatornák közti váltogatás, alkalmazások megnyitása vagy különböző bemenetek küldése.

Ez az a komponens továbbá, mely a prezentációs réteget a legalapvetőbb funkciókkal és adatokkal ellátja; annak főképernyőjén a legtöbb megjelenő adat innen származik.

A második rétegbeli projektek strukturális felépítését illetően személy szerint már a kezdetektől fogva az osztályok azok funkcióik szerinti csoportosítását részesítem előnyben az adott erőforrások szerinti csoportosításával szemben, így controller, service, egyéb szervizekben repository, config és számos más, úgynevezett „package” -dzsel

találjuk szemben magunkat. Látható továbbá, hogy a legjobb gyakorlatok mentén történő fejlesztésből kifolyólag az üzleti logika rétegében interfészeket valósítanak meg a konkrét, implementációt tartalmazó Java osztályok.

A bemutatást a megvalósítás tetejétől az aljáig kívánom bemutatni, azaz attól a ponttól haladva, mellyel a felhasználó közvetlenül interakcióba lép (HTTP API) az üzleti logika rétegéig átfogóan.

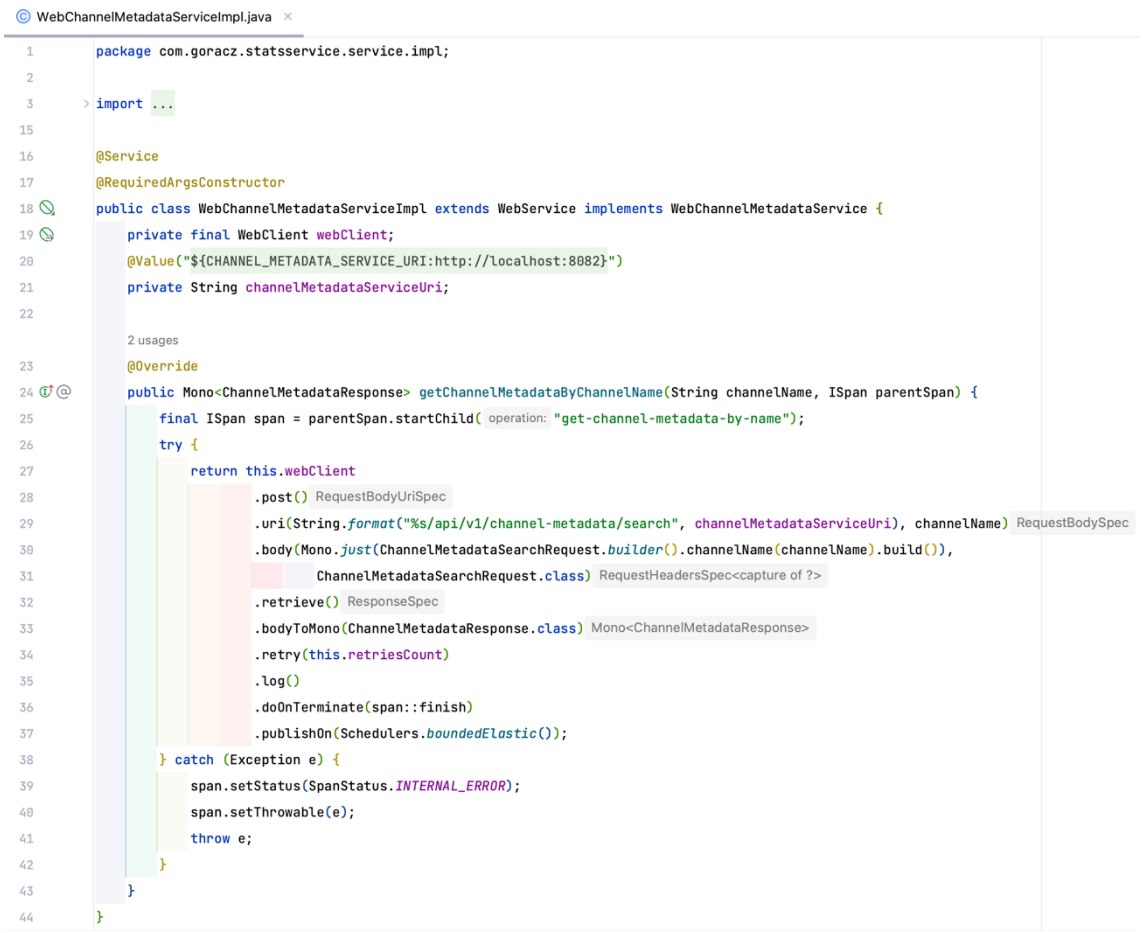
Minden HTTP végpont mögött egy egyszerű osztály, illetve azon belül egyszerű metódusok találhatóak. Minden, az applikáción belül az ezen rétegben található osztály és azok metódusai hasonlóak. Nem állnak másból, mint a metódus elején manuális nyomkövetésből a végrehajtási idő mérhetőségének biztosítása érdekében, illetve azt követően az üzleti logika meghívásából, majd az az által visszaadott válasz elküldéséből.

A nyomkövetés a következőképpen néz ki: a kérés belépési pontján manuálisan készítünk egy új tranzakciót (mely egy különálló „trace” lesz), ezt pedig továbbadjuk annak a metódusnak, melyet a belépési pont lekezelője meghív egy egyel lejjebbi rétegben. Ez mindenhol így történik, tehát egy sokszoros leágazás jön létre, annyi különbséggel, hogy a trace-t követő objektumokat „span” -nek nevezzük. Arra a kérdésre, hogy erre miért ily módon, következetesen manuálisan kell, hogy sor kerüljön, a későbbiekben fogok válaszul szolgálni.

A továbbiakban vegyük példaként a csatornák lekérését célzó kérést. Ez esetben egy olyan metódus fog lefutni, mely aztán meghív egy üzleti logikai rétegben található metódust.

Viszonylag egyszerű metódusokról van szó: először a gyorsítótárhoz fordulunk, és onnan próbáljuk meg beolvasni azt az adatot, mely aztán válaszként fog szolgálni a kérésre. Azonban amennyiben az üres, akkor kénytelenek vagyunk ez esetben a TV-re hagyatkozni, és ahhoz intézünk – az interfészen keresztül – egy kérést, és az eszköz által adott válasz fog végig futni az interfészen keresztül egészen eddig a pontig, aminek értékével aztán visszatér az üzleti logikai réteg, majd a kontroller réteg, viszont ezt megelőzően a kapott érték visszaíródik a gyorsítótárba, méghozzá a „bounded elastic” ütemezővel, mely ideális az I/O műveletek ütemezésére, ugyanis itt korlátozódik a maximálisan használt szálak száma, ebből kifolyólag blokkoló I/O művelet esetén

továbbra is reszponzív marad a rendszer, illetve ezen szálak újra felhasználhatóak is, így csökken a szálak kezelésére fordított processzoridő. A folyamat legvégén lezárul a nyomkövetési funkciókat ellátó tranzakció. A 71-es és a 80-as sorban itt a már az előzőekben említett „immediate” ütemezővel kerülnek ütemezésre a feladatok, hogy azok minél hamarabb visszatérjenek a kívánt értékkel.



9.6 ábra Adott csatorna metaadatait lekérő osztály, mely egy másik mikroszervizhez fordul a kéréssel [saját szerkesztés]

Ami még érdekes lehet ennek a mikroszerviznek az esetében, az az, hogy szükség esetén hogyan kommunikál a többi, az egész applikáció architektúrájának szempontjából ugyanezen rétegben található mikroszervizzel.

Ez sem történik másképp, mint az interfészhez fordulás esetén: a 9.8 ábrán látható, hogy adott egy Java osztály, mely leszármazik a „WebService” osztályból (ezáltal megörökli a „retriesCount” mező alapértelmezett értékét, mely azt deklarálja, hogy hiba esetén hányszor próbálkozzon újra a tényleges megvalósítás) és implementál egy Java interfészt,

majd nemes egyszerűséggel egy HTTP kérést intéz a megfelelő végpontra, melynek értékét aztán adott osztály objektumba szerializálva visszaad a meghívó metódus számára.

9.3 A metaadatokat szolgáltató szerviz megvalósítása

A metaadatokat szolgáltató szerviz egy viszonylag kis mikroszerviz, melynek két feladata van: egyrészt adatokat szolgáltatni bizonyos csatornákról (például amennyiben adott egy csatorna neve, akkor elküldje a hozzá tartozó logó webes linkjét), másrészt az elektronikus műsorújság szolgáltatása. Előbbi sem egyszerű feladat bizonyos diszkrepanciákból adódóan, azonban utóbbi megvalósítása is jóval bonyolultabb a kelleténél bizonyos okokból kifolyólag, melyek később még említésre kerülnek.

```
1 usage  Roland Gorácz
@Override
public Mono<PopulateChannelsResponse> populate(Collection<Channel> channels, ISpan transaction) {
    final ISpan span = transaction.startChild( operation: "populate-channels");
    try {
        return Flux.fromIterable(channels) Flux<Channel>
                .flatMap(channel -> {
                    final ISpan childSpan = span.startChild( operation: "populate-channel");
                    return this.cacheProvider
                            .getChannelMetadataCache() ReactiveValueOperations<String, ChannelMetadata>
                            .get(channel.getChannelName()) Mono<ChannelMetadata>
                            .switchIfEmpty(this.searchInDatabase(channel.getChannelName()))
                            .flatMap(metadata -> writeToCache(metadata, childSpan))
                            .flatMap(channel::populateChannelLogo) Mono<Channel>
                            .subscribeOn(Schedulers.parallel())
                            .doOnTerminate(childSpan::finish);
                }).delayElements(Duration.ofMillis(1))
                .collectList() Mono<List<...>>
                .flatMap(populatedChannels -> Mono.just(PopulateChannelsResponse.builder()
                        .channels(populatedChannels)
                        .build())) Mono<PopulateChannelsResponse>
                .doOnTerminate(span::finish);
    } catch (Exception e) {
        span.setStatus(SpanStatus.INTERNAL_ERROR);
        span.setThrowable(e);
        throw e;
    }
}
```

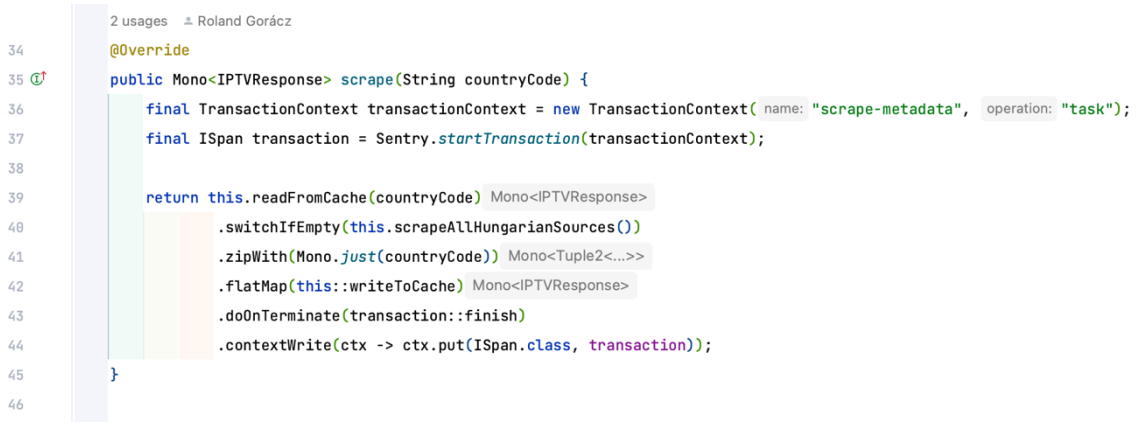
9.7 ábra Metódus, mely a csatorna objektumokat feltölti metaadattal [saját szerkesztés]

Ebben a mikroszervizben már csak a tényleges, üzleti logikai rétegben történő megvalósítást kívánom bemutatni, hiszen az előzőekben említettek szerint az ezen alkalmazásokban használt minták ugyanazok.

A 9.9 ábrán látható, hogy amikor valaki meghívja ezt a metódust, akkor mint adatfolyam végig megyünk a paraméterként kapott csatorna objektumokon, és a gyorsítótárban, vagy az adatbázisban található metaadatokkal kitöltjük azt. Itt a „parallel” ütemezővel történnek ütemezésre a feladatok, hiszen ez egy tipikusan olyan feladat, ahol teljesen párhuzamosan történhet a feladatvégrehajtás.

Felmerülhet azonban a kérdés, hogy végtére hogyan kerülnek az adatbázisba a metaadatok? Nos, erre egy szuboptimális megoldással tudok csak szolgálni: mivel eltérhet a felhasználó ISP-jétől érkező csatornanév, és a különböző külsős adatszolgáltatók által megadott csatornanév az egyébként ugyanazon csatorna esetében, ezért nem hozható létre egyértelmű hozzárendelés az ISP-től érkező, illetve a külső adatszolgáltatóktól érkező csatornaneveket illetően, ami azt eredményezte, hogy kézzel létre kellett hoznom egy olyan adatbázist, mely úgy tartalmazza a csatornaneveket, ahogyan azok érkeznek az ISP-től, és ezen manuálisan felvitt adatok, mint adatbázis rekordok minden indításnál létrejönnek az adatbázisban.

A mikroszerviz másik oldalát az elektronikus műsorújság előállítása képezi. Ezt a funkcionalitást azért nem lehetett olyan egyszerűen implementálni, mint ahogyan az felmerülhet az olvasóban, mert a televízió, habár tartalmaz olyan publikus API végpontot, mely ezt hivatott lenne egy-az-egyben, az ISP-től lekért módon továbbadni kérés esetén, sajnos az minden esetben egy üres tömbbel tér vissza, így nem maradt más lehetőségem, mint hogy külső adatforrásokhoz forduljak.



9.8 ábra Az elektronikus műsorújsághoz szükséges adatok begyűjtéséért felelős metódus [saját szerkesztés]

A 9.10 ábrán látható a metódus, mely a műsorújsághoz szükséges adatokért felelős; ez sem működik másképp, mint a már megismert metódusok, először gyorsítótárból próbál olvasni, majd, ha ez nem sikerül, HTTP kéréseket intéz külső adatforrások felé, melynek válaszait aztán osztályobjektumokba szerializálja, és vissza kiírja őket a gyorsítótárba. Ez esetben is felmerül az összerendelés problémája, mely jelen esetben prezentációs rétegre hárul (ez a későbbiekben szintén említésre kerül majd).

9.4 A statisztikai adatokat számító szerviz megvalósítása

Ez a legbonyolultabb mikroszerviz az egész applikációban. Több eseményre figyel egyszerre, és ezek bekövetkezése, illetve bekövetkezésének sorrendje alapján határozza meg a televízió jelenlegi állapotát, melyre alapozva aztán különböző statisztikákat számít a metaadat mikroszerviz felhasználásának segítségével.

Annak ellenére, hogy üzleti logikáját tekintve ez a legkomplexebb komponense az applikációnak, itt sem történik másképpen az adatfeldolgozás, mint bármely más Java-s mikroszervizben, így csak a leglényegesebb pontokra térek ki.

A csatornanézési szokások statisztikáinak előállítása a következőképpen működik: amennyiben a jelenleg legutolsóként mentett csatornához (legyen az a gyorsítótárban vagy az adatbázisban) nincs társítva olyan időpont, amikor annak nézése véget ért volna (mert eddig nem változott meg a televízió állapota a bekapcsolás óta, vagy pedig mert nem váltottunk csatornát vagy nem nyitottunk meg alkalmazást), akkor a jelenlegi időpont lesz ennek a legutoljára eltárolt csatornának az az időpontja, amikor véget ért a nézése (140-es sor), illetve az a csatorna (vagy applikáció) is eltárolásra kerül, melyre

éppen váltottunk amennyiben viszont igen (mert például legutóbb kikapcsoltuk a tévét), akkor egy teljesen új rekord kerül létrehozásra az adatbázisban (illetve a gyorsítótárban), ahol a megtekintés kezdete a végrehajtás kezdetének az időpontja lesz (146-os sor). Az eltárolt rekordhoz társul metaadat is, például, hogy az adott csatorna milyen kategóriába tartozik (hírműsorok, sport és így tovább).

```
79      @KafkaListener(topics = KafkaTopic.POWER_STATE_CHANGES,
80              containerFactory = POWER_STATE_CHANGE_LISTENER_CONTAINER_FACTORY)
81      public void onPowerStateChange(ConsumerRecord<String, String> message) throws KafkaConsumeFailException {
82          final TransactionContext transactionContext = new TransactionContext( name: "on-power-state-change",  operation: "task");
83          final ISpan transaction = Sentry.startTransaction(transactionContext);
84          try {
85              this.handlePowerStateChange(message, transaction).subscribe();
86          } catch (Exception exception) {
87              transaction.setStatus(SpanStatus.INTERNAL_ERROR);
88              throw new KafkaConsumeFailException(exception.getMessage());
89          } finally {
90              transaction.finish();
91          }
92      }
93
```

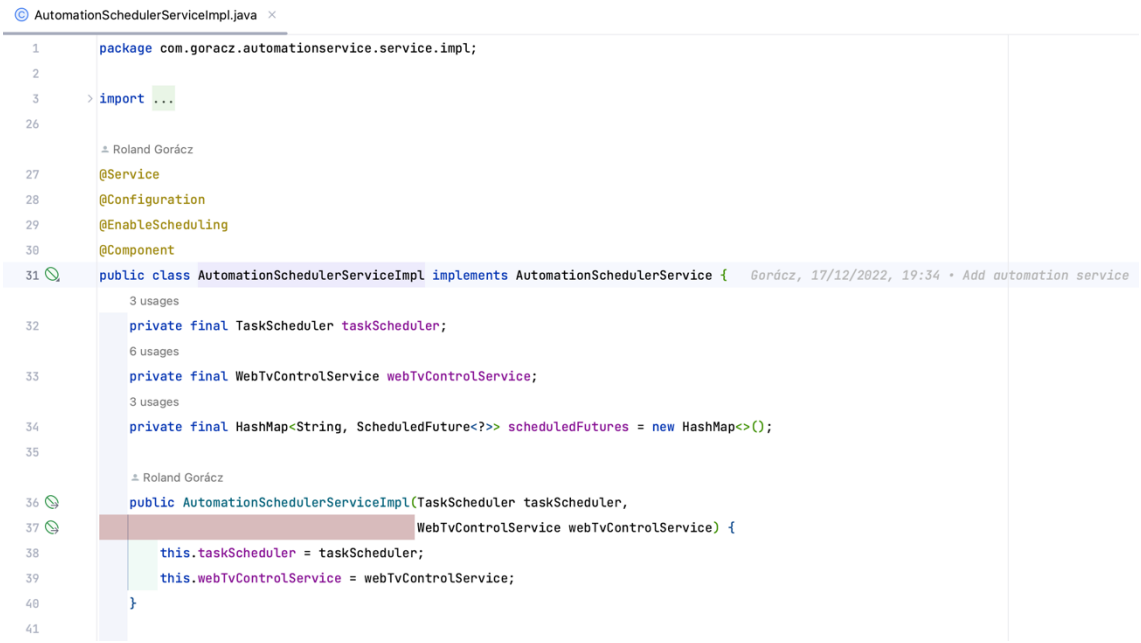
9.9 ábra A televízió állapotváltozását lekezelő metódus [saját szerkesztés]

A másik lényeges szolgáltatás, az annak nyomkövetése, hogy a tévé mikor, és mennyi ideig volt bekapcsolt állapotban. Ennek kezeléséről elsősorban a 9.11 ábrán látható metódus gondoskodik, mely a gyorsítótárban, az adatbázisban és a kapott eseményben található adatok segítségével határozza meg mindezt.

A gyakorlatban ez nem egy egyszerű feladat, ugyanis figyelembe kell venni azokat a szcenáriókat, amikor a tévé éppen kikapcsolt állapotból bekapcsolt állapotba kerül és fordítva, de azt is, amikor a rendszer éppen egy hibából tér vissza. A fenti sorok szintén nem állnak másból, mint nyers logikai kifejezések kiértékeléséből, adatbázisból, illetve gyorsítótárból, illetve azokba történő olvasásból és írásból.

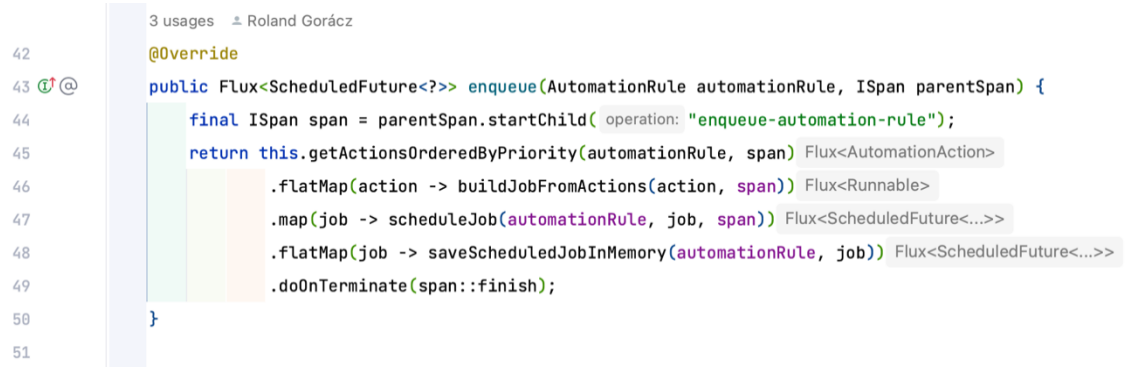
9.5 Az automatizálási szabályokat kezelő szerviz megvalósítása

Végül a második réteg legutolsó mikroszervize is terítékre kerül: ez szintén egy viszonylag kisebb alkalmazás, azonban tartalmaz egy-két bonyolultabb megoldást, hiszen a dinamikus ütemezés nem feltétlenül minősül könnyű feladatnak.



9.10 ábra Az ütemezésért felelős üzleti logikát tartalmazó Java osztály [saját szerkesztés]

A 9.12 ábrán látható, hogy az osztály tartalmaz egy privát hasítótábla mezőt, melyben kulcs-érték pár szerint kerülnek eltárolásra az éppen beütemezett feladatok, annak érdekében, hogy azonosítójuk alapján azokat lehessen mind ütemezni, mind törölni a végrehajtandó feladatok közül, ugyanis amikor egy adott szabály hozzáadásra kerül, az eltárolásra kerül mind ebben a mezőben, mind a gyorsítótárban, mind pedig az adatbázisban is, ahonnan aztán induláskor felolvasásra, majd ismételten beütemezésre kerülnek a rendszeresen végrehajtandó feladatok.



9.11 ábra A feladatokat ütemező metódus [saját szerkesztés]

A 9.13. ábrán látható a közvetlenül a feladatokat ütemező metódus, mely először egy „Runnable”, azaz kvázi futtatható, végrehajtható, ütemezhető feladatot csinál a

szabályból a 46. sorban, majd ténylegesen beütemezi azt (47. sor), végül pedig elmenti az osztály privát mezőjébe, tulajdonképpen a memóriába (48. sor).

A szabály ütemezési módjának megvizsgálását követően (azaz, hogy csak adott időpontban, vagy rendszeresen kell lefuttatni), végső soron a Spring keretrendszer ütemezőjén keresztül történnek ütemezésre az automatizálási szabályok, melyek aztán a megfelelő időben a „control service” igénybevételével kerülnek végrehajtásra.

9.6 A prezentációs réteg, a felhasználói felület megvalósítása

A prezentációs réteg, ahogyan azt már említettem, a NextJS keretrendszer és a Redux Toolkit könyvtár segítségével került megvalósításra oly módon, hogy annak működése esemény vezérelt legyen. Itt szintén nem szeretnék kitérni a keretrendszer- és könyvtárspecifikus implementációs részletekre, csupán a lényegibb elemekre.

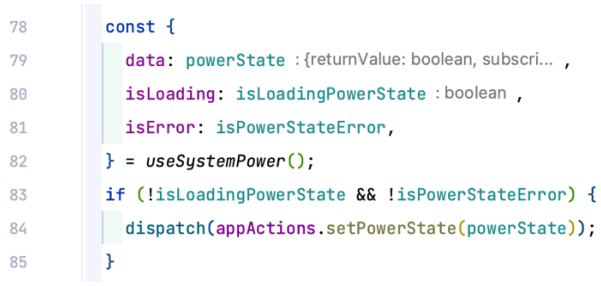
A felhasználói felület megvalósításának is két fontos aspektusa van: az egyik a szervertől érkező események lekezelése, a másik pedig az adatok valós idejű befrissítése (erről nagyrészt gondoskodik a keretrendszer).

Az események feldolgozása úgy történik, hogy feliratkozunk a második réteg által biztosított adatfolyam végpontokra, majd az események tüzelésének esetén hozzárendelünk egy az azt lekezelő metódust. Az üzenet jellegének, tartalmának megfelelően aztán befrissítjük a Redux tárat, melyre olyan horgok mutatnak, melyek végtére olyan komponenseken belül vannak használva, amelyek az azok által biztosított adatokat használják, így értem el, hogy amennyiben a Redux tár befrissül, illetve ott van adat, akkor az adott, azt az adatot használó komponens, és az az által megjelenített adat is automatikusan befrissüljön.

```
4 usages
72  const useSystemPower = () :{...}  => {
73    const powerState :{returnValue: boolean, subscri...  = useSelector( selector: (state: { app: AppSliceState }) => state.app.powerState);
74
75    const { data, error } = useSWR(
76      key: powerState ? null : `${CONTROL_SERVICE_BASE_URL}/system/power`,
77      fetcher
78    );
79
80    return {
81      data: powerState || data,
82      isLoading: powerState ? false : !error && !data,
83      isError: powerState ? false : error,
84    };
85  };
86
```

9.12 ábra Egyedi horog, mely a tévé állapotának visszaadásáért felelős [saját szerkesztés]

A 9.14 ábrán egy olyan egyedi horog látható, mely a 73-as soron megpróbálja kiolvasni a kért adatot a Redux tárból, és ennek alapján a 75-78. sorokban eldől, hogy szükség van-e a második réteghez fordulni a kéréssel (amennyiben a Redux tárban nem volt megtalálható az adat), vagy sem, és az adat rögtön visszaadható.



```
78  const {
79    data: powerState :{returnValue: boolean, subscri... ,
80    isLoading: isLoadingPowerState :boolean ,
81    isError: isPowerStateError,
82  } = useSystemPower();
83  if (!isLoadingPowerState && !isPowerStateError) {
84    dispatch(appActions.setPowerState(powerState));
85  }
```

9.13 ábra A horgot használó „Dashboard” komponens [saját szerkesztés]

A 9.15 ábrán pedig látható, hogy a horog hogyan használható: csupán a 9.14 ábrán látható metódust kell meghívni annak beimportálása után, mely visszatér az adattal, illetve olyan jelzőértekkel, melyek az adatlekérés státuszát és a hiba tényét indikálják; ezeket az úgynevezett „destructuring assignment” használatával rögtön ki lehet nyerni a metódus által visszaadott objektumból. Emellett szintén látható a 9.15 ábra 84. sorában, hogy a kapott érték garantáltan kiírásra kerül a Redux tárba, amennyiben nem lépett fel hiba.

10. A MEGVALÓSÍTÁS ELEMZÉSE, ALKALMAZÁSÁNAK ÉS TOVÁBBFEJLESZTÉSI LEHETŐSÉGEINEK SZÁMBAVÉTELE

Ebben a fejezetben áttekintésre kerül a megvalósított szoftver hatékonysága, illetve az ennek megbecsléséhez alkalmazott módszer, a terheléses tesztelés és a profilozás. Emellett megvizsgálom a program fejlesztési lehetőségeit, illetve a megvalósítás során felmerült nehézségeket.

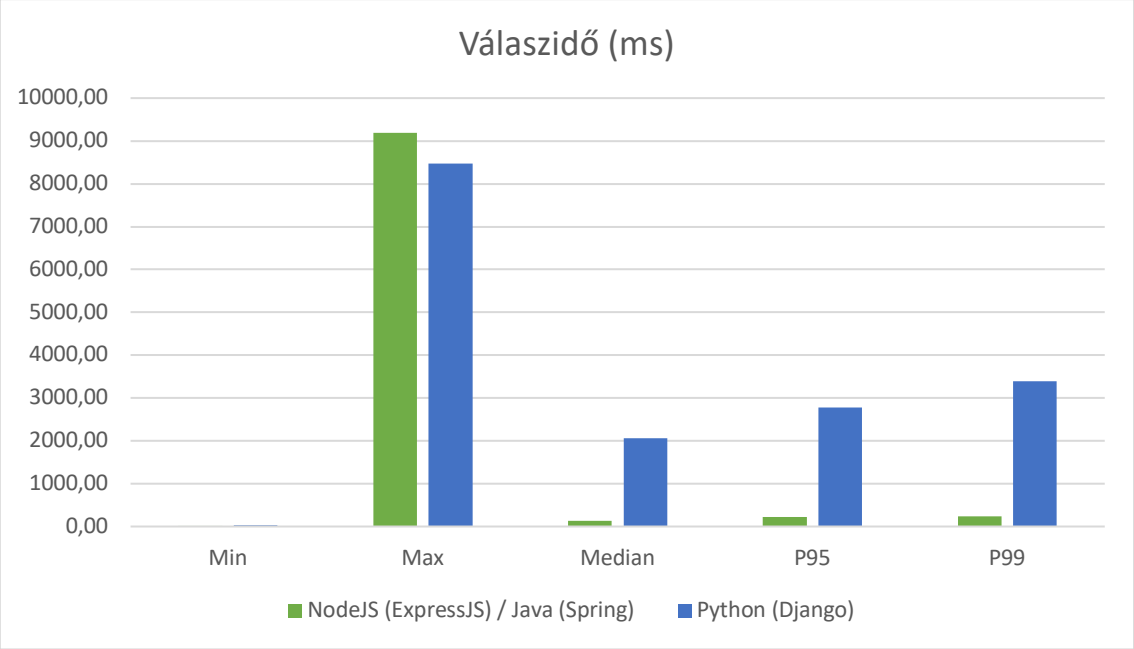
10.1 Hatékonyság, teljesítmény

Először is ki kell térnem dolgozatom egyik lényeges pontjára: tényleg energiahatékonyabb-e az általam felvázolt elvek alapján megkomponált szoftverem, mint amennyiben nem így tettem volna?

Összehasonlítás céljából lefejlesztettem az alkalmazás funkcionalitás szempontjából ekvivalens változatát a Python programozási nyelv és a Django illetve Django Rest Framework keretrendszerek használatával (figyelembe véve a Python feltörekvő népszerűségét az adattudományi területek felkapottsága végett), hogy egy vázlatos képet tudjak adni arról, hogyan viszonyul egymáshoz a kettő alkalmazás valós életben vett teljesítménye. Emlékeztetőként megjegyzem, hogy a Python egy „interpreted” programozási nyelv, tehát futási időben kerülnek értelmezésre a végrehajtandó utasítások, mely egy energiafelhasználás szempontjából kedvezőtlen képpel társul, illetve szintén megjegyzendő, hogy adott alkalmazás nem alkalmaz gyorsítótárazást, nem esemény vezérelt, és blokkoló, szinkron feladatvégrehajtással operál.

A teljesítmény kiértékeléséhez hibrid módszert alkalmaztam, mely kombinálja a szintetikus és manuális, valós tesztek módszerét; egy olyan automatikus terheléses tesztekhez használatos eszközt, mely bizonyos végpontokra, bizonyos késleltetéssel kéréseket intézett az alkalmazásokhoz, melyek tehát mind a kérések kiszolgálásával, mind a kiváltódó események, illetve bekövetkező állapotváltozások feldolgozásával foglalkozniuk kellett. Az általam használt eszköz nevezetesen az Artillery, illetve egy ezzel párosított Bash szkript (ezt a terjedelem korlátjai miatt nem taglalom), mely kiszámítja a relatív CPU időt a terheléses teszt elkezdésének és annak befejeződésének alapján.

A terheléses teszt ciklusaiban csatornaváltás és hangerőállítás is történik, ugyanis ezen kérések extra, egyébként erőforrásigényes utófeldolgozással járnak statisztikák előállítása céljából. Ugyanezen teszteset került lefuttatásra mindkét applikáció esetében.

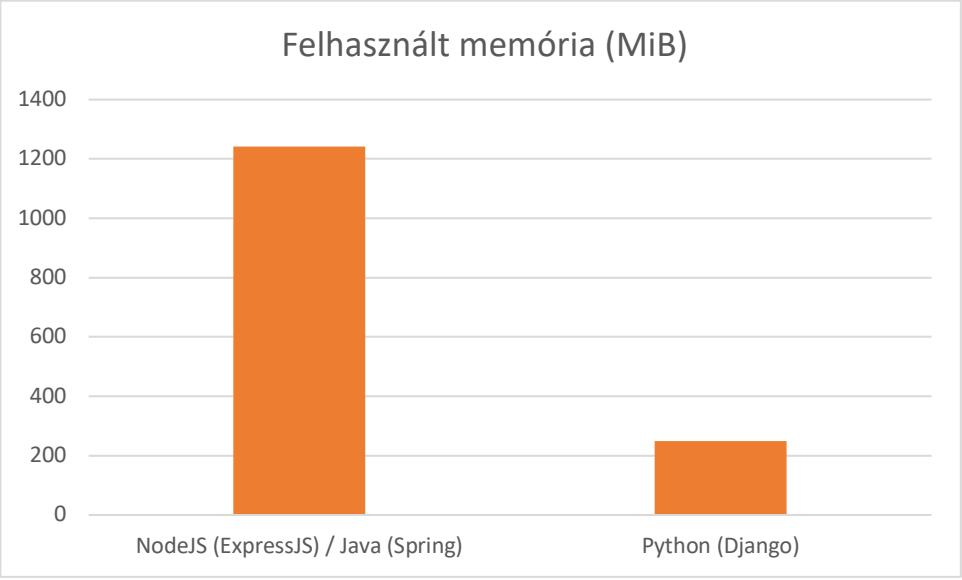


10.1 ábra Válaszidő alakulása a két applikáció között (a kisebb érték a jobb) [saját szerkesztés]

A teszt összesen 36 kérést tartalmazott. A 10.1 ábráról tisztán le lehet olvasni, hogy az eredeti applikáció jóval gyorsabban szolgálja ki a kéréseket, míg a Python applikáció közel 15-ször lassabban. A válaszidők minimuma, azaz a leggyorsabb válaszidő alkalmazásonként közel azonos volt, a köztük lévő minimális különbség az ábra alapján elhanyagolható, melyből következtethető, hogy amennyiben az applikációk nincsenek terhelés alatt, közel azonos teljesítményt tudnak nyújtani. A válaszidők maximuma a minimumhoz hasonlóan azt szemlélteti, hogy az alkalmazások által kiszolgált kérések válasz idejének maximuma hogyan alakult. Érdekesség, hogy a leghosszabb válaszidőt a kérések közül az eredeti applikáció produkálta, körül-belül 500 milliszekundumos hátránnyal, ami azt jelenti, hogy nagy terhelés esetében bizonyos felhasználók rosszabb felhasználói élménnyel szembesülhetnek az eredeti applikáció esetén, mint a Python-os megvalósítással. A medián a válaszidők növekvő sorrendben történő rendezését követően a középső érték. Látszik, hogy ez jóval kedvezőbb érték az eredeti szoftver esetében, ami azt jelenti, hogy a legtöbb felhasználó kedvezőbb válaszidőben részesül, gyorsabb lesz számára az alkalmazás, mint a Python-os megvalósítás esetében. A legszemléletesebb és

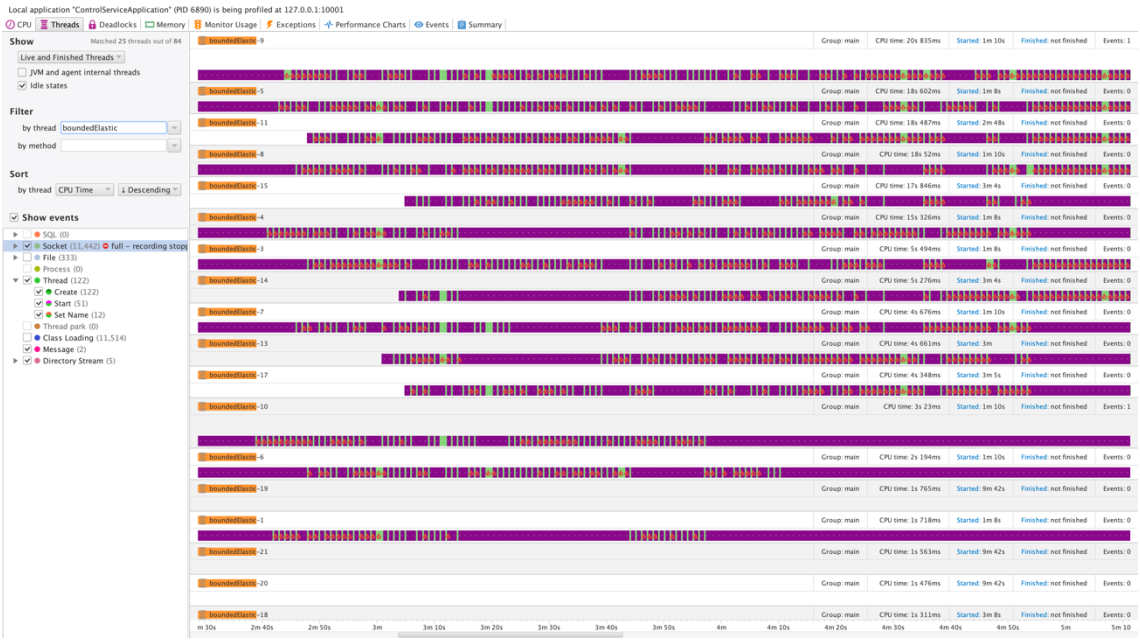
egyben legfontosabb különbség a 95. és 99. percentilis értékeknél mutatkozik meg, ugyanis ezek azt jelölik, hogy a válaszidők 95, illetve 99 százaléka az adott értéknél gyorsabbak. Megfigyelhetjük, hogy a kérések 95 százalékában a NodeJS illetve Java applikáció esetében 230 milliszekundum körül alakul, míg a Python megvalósítás esetében ez több, mint tízszerese ennek az értéknek, közel 2780 milliszekundum, ami azt jelenti, hogy nagy terhelés esetén sem növekedik meg, csak egy minimális mértékben az eredeti applikáció esetében a válaszidő, azonban a Python-os applikációról ez nem mondható el, ott exponenciálisan elkezd megnövekedni a válaszidő. A 99. percentilis esetében még nagyobb különbség jelentkezik, az eredeti megvalósítás esetén a válaszidő a 95. percentilishez képest éppen 15 milliszekundummal nő, míg a Python megvalósítás esetén közel 500 milliszekundumos a növekedés, ami azt jelenti, hogy a hatékonyságra optimalizált megvalósítás jóval konzisztensebb teljesítményt tud nyújtani terhelés esetén.

E teszt tekintetében – adott azonos hardverrel futtatva az alkalmazásokat – csak az időtényezővel számolva, körül-belül 10-15-ször hatékonyabbá tehetjük az alkalmazásunkat, amennyiben az ehhez megfelelő technológiákat, keretrendszereket, architektúrát és komponenseket választjuk meg, nemmellesleg a fejlesztés menetének során is szem előtt tartjuk az optimális feladatütemezést. Egyéb érdekesség, hogy az eredeti applikáció interfész komponense futása során 0 milliszekundumnyi processzoridőt igényelt, ebből a szempontból kiemelkedő teljesítményt nyújtva.



10.2 Az applikációk memória igénye, (a kisebb érték a jobb) [saját szerkesztés]

Meg kell jegyeznünk azonban, hogy míg nyers teljesítmény és hatékonyság szempontjából jobban teljesít az eredeti applikáció, megközelítőleg 5-6-szoros memóriaigénnyel rendelkezik JVM nyelv révén Python-os vetélytársához képest. Megállapítható azonban, hogy a memória modulok elenyésző energiaigénnyel rendelkeznek, és felhő környezetben az egyszeri, nagy költséggel járó megvásárlásukat is elkerülhetjük.



10.3 ábra A vezérlő mikroszerviz terheléses tesztje [saját szerkesztés]

Az 10.3-as ábrán megfigyelhető, hogy a programot végrehajtó szálak extenzív terhelés esetén sem blokkoltak, mivel a különböző sorokban, melyek a szálakat jelölik, sehol sincsen piros csík, csupán lila, mely a feladatra várakozást, illetve zöld, mely feladatvégrehajtást jelöl. Nagy terhelésnél blokkolás helyett dinamikusan, a thread pool-ból kerültek allokálásra miután már nem maradt olyan szabad szál, amely a sok felgyülemlett kérést ki tudta volna szolgálni blokkolás nélkül. Ezek a szálak aztán később, miután már adott idő elteltével egy feladat sem futott rajtuk, „eltakarításra” kerülnek.

10.2 Fejlesztési lehetőségek

A teljesítmény aspektusától eltekintve azonban továbbra is számos fejlesztési lehetőség adódik. A legelső említésre méltó az applikáció stabilitása. Időhiány miatt sajnos nem valósulhatott meg sem a teszt-vezérelt fejlesztés (TDD), sem pedig az utólagos tesztírás,

így maradtak adminisztrációs hibák az alkalmazásban, legfőképp az első nagy refaktorálás után, mely a végleges Proof of Concept elkészítése során történt meg. Tapasztalatként felróható tehát, hogy refaktorálást mindenképpen csak graduálisan, vagy kellőképpen manuális, előnyösebb esetben automatizált tesztelés mellett célszerű végezni.

További fejlesztési lehetőség rejlik a már a megvalósításról szóló blokkban említett összerendelések kapcsán, mely az elektronikus műsorújság mögöttes algoritmusát érinti. Itt ugyanis, a már taglaltak szerint nem igazán lehetséges ésszerű algoritmust írni szintaktikai értelemben vett, esetenként lényegesen eltérő sztringek komparálására, így ez esetben kiváló lehetőség nyílik a mostanában felágazóban lévő mesterséges intelligencia az applikációba történő integrálásában.

Szintén említésre méltó, jelentős javítási lehetőség több televízió kezelésének a lehetősége, azaz, hogy a felhasználói felületen keresztül könnyedén lehessen televíziók regisztrációját kezelni, illetve szükségszerűen váltogatni azok között.

Utolsóként említhető egy mobilapplikáció esetleges párosítása ezen webalkalmazáshoz, hogy ezen eszközméretekre specifikusan optimalizált, operációs rendszerhez natív felületen keresztül lehessen kezelni például az automatizálási szabályokat (akár az adott mobileszköz operációs rendszerének okosotthon-alkalmazásába történő integrálásával), vagy hogy mobileszközről is kényelmesen lehessen a begyűjtött statisztikákat böngészni.

10.3 Akadályok, nehézségek

Pár nehézséget is szeretnék megemlíteni, melyek további tanulsággal szolgálhatnak.

Dokumentumomban szerettem volna kitérni az elosztott nyomkövetésre, mély betekintést nyerve az applikáció belső működésébe metódusvégrehajtás, adatbázis-lekérések szintjén, kód szinten, azonban egyetlen ágens segítségével sem tudtam ezt megtenni automatikus nyomkövetéssel véges időn belül, a Dynatrace és az OpenTrace termékekkel próbálkozva.

Ez abból adódik, hogy feladatvégrehajtás során egyszerre több szálon futnak feladatok, nem pedig szinkron, soros módon, így az automatikus nyomkövetés konfigurálása nagyon körülményes úgy, hogy működjön is. Esetemben kellemetlen megoldást kellett eszközölnöm, manuális nyomkövetés formájában: a kontroller rétegben manuálisan

létrehozott „trace” objektumot tovább kellett adnom a legmélyebb metódushívásokig, mint a metódusok legutolsó paramétere.

További nehézséget okozott az alkalmazásban történő hibakeresés, hasonló okból kifolyólag: nehéz volt nyomon követni különböző utasítások végrehajtásának sorrendiségét, és emiatt valahol csak rész-adatok kerültek visszaadásra bizonyos metódusok által a parallelizmus miatt, így 1 milliszekundumos mesterséges késleltetést kellett bevezetni a hiba elhárítása érdekében, mert a példaesetben nem sikerült kideríteni annak eredeti, mélyre menőbb okát.

11. ÖSSZEFOGLALÓ

Célom volt olyan fejlesztési és tervezési minták felfedezése, megfogalmazása, melyek segítségével és követésével kevéssé energiaigényes szoftverek fejleszthetők az átlagszoftverekhez képest, melyek végül aztán hozzájárulhatnak a környezet kisebb mértékű szennyezéséhez, és a globális felmelegedés mértékének csökkentéséhez.

A projekt fejlesztése során megismerkedtem az esemény vezérelt alkalmazások működésével, a gyorsítótárazással, az aszinkron, nem-blokkoló alkalmazások működésének jellegével, illetve, hogy ezek használatával a tradicionális technológiáktól eltérően hogyan lehet nagyobb fokú energiahatékonyságot elérni szoftver szinten.

Sikerült egy jól működő applikációt lefejleszteni annak ellenére, hogy hátráltatott ebben a szakszerű, komplett fejlesztői dokumentáció hiánya a televízió gyártójától, a ma már nem karbantartott, a fejlesztést segítő, harmadik féltől származó könyvtárak, illetve a vállalkozásom komplexitása.

Szintén feljegyezhető, hogy olyan, korábban még nem igazán látott funkciókkal sikerült felvértezni az alkalmazást, mint a nézői szokásokról történő statisztika készítés vagy a teljesen valós időben működő felhasználói felület, melyben szintén gyorsítótárazás történik a Redux könyvtár használata révén.

Következtetésként megállapítható, hogy habár futási időt és teljesítményt tekintve valóban optimálisabbak lehetnek a bevezetésben említett vezérelvek alapján történő termékek, ez esetben is inkább egy arany középútról beszélhetünk csak, hiszen kompromisszumot kell kötni az optimális futási teljesítményért cserébe a magasabb memória igény tekintetében.

Emellett kutatásom során beigazolódott, hogy lényeges teljesítménybeli és energiafelhasználási különbségek lehetnek csak és kizárólag már a programozási nyelv megválasztásánál is, így nemcsak a megoldás architektúrája, vagy a mögöttes hardver karakterisztikái számítanak, de az a döntés is, hogy mely technológiákkal kívánjuk megvalósítani jövőbeli termékünket.

12.SUMMARY

My goal was to discover and identify such engineering and design patterns that make software consume less energy compared to traditional software when followed, so that such software contribute to lessening the pollution by consuming less energy and lessening the effects of global warming.

Thoughout developing the project, I got myself familiar with how event-driven applications work, with caching, with the characteristics of asynchronous, nonblocking applications, and how all of these could contribute to creating a more energy efficient software.

I have successfully made a correctly working application despite lacking technical documentation from the maker of the television, the fact that maintained, third-party libraries helping development were also lacking, and that I have made a difficult commitment with this project.

It is also commendable that I could feature before-unseen features in the application such as creating statistics of TV viewing habits and a completely real-time user interface featuring caching in form of using the Redux library.

In conclusion, despite a product developed with the patterns kept in mind mentioned in the Introduction section, we can only talk about the golden middle way, given that we have to sacrifice memory for better runtime performance.

In addition to that, it has been proven that it is possible to have meaningful performance and energy consumption differences not only between choosing different programming languages for implementation, but also the solution’s architecture and the hardware behind it, just as the decision with which technologies to implement our future product.

IRODALOMJEGYZÉK

[1] P. Reactor, "Project Reactor dokumentáció," [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#parallel--. [Accessed 5 10 2023].

[2] J. F. Adolfsen, F. Kuik, T. Schuler és E. Lis, „European Central Bank," 1 4 2022. [Online]. Available: https://www.ecb.europa.eu/pub/economic-bulletin/focus/2022/html/ecb.ebbox202204_01~68ef3c3dc6.en.html. [Hozzáférés dátuma: 29 5 2023].

[3] J. McDonald, B. Li, N. Frey, D. Tiwari, V. Gadepally és S. Samsi, „Great Power, Great Responsibility: Recommendations for Reducing Energy for Training Language Models," Association for Computational Linguistics, Massachusetts, 2022.

[4] Intel Corporation, „InfoQ," 1 April 2009. [Online]. Available: https://www.infoq.com/articles/power-consumption-servers/. [Hozzáférés dátuma: 9 December 2023].

[5] F. Pearce, „YaleEnvironment360," 3 April 2018. [Online]. Available: https://e360.yale.edu/features/energy-hogs-can-huge-data-centers-be-made-more-efficient. [Hozzáférés dátuma: 9 December 2023].

[6] A. Sayar, Ş. Arslan, T. Çakar, S. Ertuğrul és A. Akçay, „High-Performance Real-Time Data Processing: Managing Data Using Debezium, Postgres, Kafka, and Redis," IEEE, Sivas, 2023.

[7] S. Sha, W. Wen, G. A. Chaparro-Baquero és G. Quan, „Thermal-constrained energy efficient real-time scheduling on multi-core platforms," Elsevier B.V., Florida, 2019.

[8] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes és J. Saraiva, Ranking programming languages by energy efficiency, Elsevier B.V., 2021.

[9] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes és J. Saraiva, Energy Efficiency across Programming Languages, New York: Association for Computing Machinery, 2017.

[10] R. C. Martin és D. Wampler, Clean Code: A Handbook of Agile Software Craftsmanship, Pearson, 2008.

ÁBRAJEGYZÉK