NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Kučera Juraj**
2023 Hallgató törzskönyvi száma: **T657/FI38878/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Alkalmazott Informatikai Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Kučera Juraj**
Törzskönyvi száma: **T657/FI38878/N**

A dolgozat címe:

Valós idejű esemény folyamok feldolgozása, megfigyelése és szabályozása

Real-time event stream processing, monitoring and regulation

Intézményi konzulens: Dr. habil. Szénási Sándor
Külső konzulens:

Beadási határidő: 2022. december 15.

A záróvizsga tárgyai: Számítógép architektúrák
Szoftvertervezés és –fejlesztés

A feladat

Tervezzen meg és implementáljon egy web alapú rendszert, ami valós idejű események áramlásából egy felületet készít az eseményekből kinyert adatok összegzésével, csoportosításával, és egyéb hasznos információk feldolgozásával! Ehhez vizsgálja meg a már meglévő hasonló fejlesztéseket, illetve az azok által használt módszereket!
Ezt követően készítse el a saját rendszer terveit úgy, hogy az az alapfunkciókon túl lehetőséget adjon arra is, hogy a felhasználó interakcióba léphessen az eseményeket létrehozó felekkel! A választott eszközök segítségével implementálja az alkalmazást, majd a megfelelő tesztekkel igazolja annak helyes működését! Végül értékelje a fejlesztés eredményeit!

A dolgozatnak tartalmaznia kell:

- a feladat leírását,
- a meglévő rendszerek bemutatását,
- az alkalmazott eszközök, technológiák és eljárások bemutatását,
- a megvalósítandó feladat tervét,
- a tesztelés folyamatát és a teszteredményeket,
- eredmények bemutatását és értékelését,
- az architektúrában rejlő tovább fejlesztési lehetőségeket,
- a dokumentációt, valamint a rendszert bemutató prezentációt mellékletként.

.....
Dr. Vámossy Zoltán
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. december 15.**
(OE TVSz 32.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

……..	..…….
külső konzulens	intézményi konzulens

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022.12.14

.....

hallgató aláírása

KONZULTÁCIÓS NAPLÓ

ÓBUDAI EGYETEM

Neumann János Informatikai Kar

Hallgató neve: Kučera Juraj

Neptun Kód: [redacted]

Tagozat: Nappali

Telefon:

Levelezési cím (pl.: lakcím): [redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:
Valós idejű esemény folyamok feldolgozása, megfigyelése és szabályozása

Szakdolgozat / Diplomamunka[2] címe angolul:
Real-time event stream processing, monitoring and regulation

Kérjük, hogy az adatokat nyomtatott nagy betűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.09.11	Szakdolgozat téma meghatározás	
2.	2022.09.19	Feladatkiírás	
3.	2022.10.06	Fejezetek vázlatos bemutatása	
4.	2022.11.21	Kidolgozott szakdolgozat véleményezése	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4 [3] tantárgy követelményét teljesítette, beszámolóra / védésre [4]bocsátható.

.....
Intézményi konzulens

[1] Megfelelő aláhúzandó!

[2] Megfelelő aláhúzandó!

[3] Megfelelő aláhúzandó!

[4] Megfelelő aláhúzandó!

KONZULTÁCIÓS NAPLÓ

ÓBUDAI EGYETEM

Neumann János Informatikai Kar

Hallgató neve:	Neptun Kód:	Tagozat:
Kučera Juraj	[redacted]	Nappali

Telefon: Levelezési cím (pl.: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[5] címe magyarul:
Valós idejű esemény folyamok feldolgozása, megfigyelése és szabályozása

Szakdolgozat / Diplomamunka[6] címe angolul:
Real-time event stream processing, monitoring and regulation

Kérjük, hogy az adatokat nyomtatott nagy betűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.11.22	Az implementált szoftver bemutatása	
2.	2022.12.06	Elkészült szakdolgozat véleményeztetése	
3.	2022.12.11	Formai követelmények kijavítása	
4.	2022.12.12	Leadás előtti konzultáció	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4 [7] tantárgy követelményét teljesítette, beszámolóra / védésre [8]bocsátható.

.....
Intézményi konzulens

[5] Megfelelő aláhúzandó!

[6] Megfelelő aláhúzandó!

[7] Megfelelő aláhúzandó!

[8] Megfelelő aláhúzandó!

Tartalomjegyzék

1. BEVEZETÉS

A mai felgyorsult, digitalizált világban egyre nagyobb információ halmaz zúdul az emberekre a nap 24 órájában. Még a 20. század végén is a friss házasok rendszerint lexikonokat vásároltak, az emberek reggelente az újságárushoz mentek. Leginkább TV vagy nyomtatott formában jutottak információhoz.
Napjainkban, az előző évtizedekhez képest, az emberekre eső információ mennyisége exponenciálisan megnövekedett.
A felhő alapú rendszerek, lehetővé tették a web alapú alkalmazások tömeges megjelenését. A közösségi média és az online tartalomgyártás fénykorában mindenki harcol a felhasználó figyelméért. Ennek következtében a mai fiatalok figyelme jóval rövidebb, mint az előző generációké.
Ugyanakkor azt látni, hogy bizonyos ágazatok még nem adaptálódtak kellőképpen, és továbbra is a régi, mai szemmel elavult, módszereket használnak. Ezek a módszerek a megváltozott világban sajnos sokszor nem bizonyulnak hatékonynak.
Köztudott tény, hogy az iskolákban a tanárok egyre nehezebben tudják kezelni a diákokat. Magyarországon a közoktatásban még mindig papír alapú vizsgázási formát használnak, a tanárok krétával írnak a táblára, ennek eredményül a magas impulzushoz, és az azonnali visszajelzéshez szokott fiatalok figyelmét egyre nehezebb lekötni.

1.1. **Motiváció**

Szakdolgozatom célja, hogy a korunk gyors információáramlására reagálva egy valósidejű eseményfeldolgozó rendszert hozzak létre. Egy olyan architektúrát mutatok be, amely lehetővé teszi az adatok azonnali feldolgozását, továbbítását, megjelenítését és a felhasználók közötti gyors kommunikációt.

Ennek a modellnek a gyakorlati bemutatására egy online vizsgázó rendszert hozok létre, amin keresztül a tanár minden pillanatban pontos képet kap a vizsgázás aktuális állapotáról, és kommunikálni tud a diákokkal. Ezáltal a tanár valós időben látja melyik azok a kérdések, amik nehézségeket okoznak, a diákok hogyan haladnak a vizsgával, és nem utolsó sorban könnyebben észreveheti a csalást.

Bízom benne, hogy ennek a programnak segítségével sikerül jobban lekötni a diákok figyelmét, és egy élvezetesebb vizsgázási élményt biztosítani, mint a diákok, mint pedig a tanár számára. Meglátásom szerint nélkülözhetetlen az oktatási szektor korszerűsítése, és szeretnék a szakdolgozatommal is hozzájárulni.

1.2. **Szakdolgozat áttekintése**

A szakdolgozatomban első fejezetekben ismertetem a feladatot és a megvalósítandó rendszer részletes specifikációját. A továbbiakban megvizsgálom a webalkalmazások készítéshez leggyakrabban használt architektúrákat, kiemelve azok hiányosságait és bemutatva az olvasónak, hogy miért nem alkalmasak a valós idejű esemény vezérelt web alapú rendszerek fejlesztésére.

Azt követő fejezetben felvázolom az általam létrehozott architektúrát, és szemléltetem hogyan fogom felhasználni az általam implementált programban. Ez a fejezet általánosan az architektúráról, a felhasználásáról és a benne rejlő lehetőségekről szól. Törekszem az architektúrát úgy megtervezni, hogy az elvet minél szélesebb körben fel lehessen használni.
Az utána levő fejezetben ismeretem a rendszer implementációjához választott technológiákat, illetve módszereket.

Ezt követően végig vezetem az olvasót a példa program implementációján. Elosztott rendszer lévén az egyes modulok fejlesztését, illetve működését külön fejezetekben tárgyalom.

A fejlesztés után kitérek a rendszer tesztelésére. Bemutatom, a használt tesztelési technikáka. Továbbá ismertetem az általam épített tesztkörnyezetet, amivel a rendszer szerveroldali részének működését ellenőrizom. Azt követően a programot egy próba vizsga keretein belül kipróbálom és a futásból levő tapasztalatokat kiértékelem.

Munkámat egy lehetséges tovább fejleszthetőséggel zárom.
Egy éles környezetben biztonságos használható, automatikusan skálázandó szoftver lefejlesztése túlmutat a szakdolgozat keretein. Ugyanakkor szeretnék egy útmutatást nyújtanom, hogyan lehetne a rendszert viszonylag egyszerűen, minimális változtatásokkal továbbfejleszteni. A továbbfejlesztéssel szemléltetem, hogy az architektúra moduláris felépítésnek és a bevált gyakorlatban használt eljárások követésével, hogyan lehetne a prototípus alkalmazásból egy éles környezetben használható rendszert megtervezni, illetve implementálni.

A javasolt tovább fejlesztés után a példaprogram használható lehet online és helyszíni vizsgázások lebonyolítására.
Végezetül a dolgozatomat összegzéssel zárom.

2. A FELADAT PONTOS LEÍRÁSA

A szakdolgozat egy valós idejű interaktív elosztott rendszert, valamint annak a működési elvét mutatja be egy gyakorlati példán keresztül. A rendszernek két típusú szereplője van, a felhasználó és a kontroller, aki felügyeli a felhasználói tevékenységet.
Jellemzően a rendszert egy darab kontroller és számos felhasználó használja.
A feladat célja, hogy a kontroller fél valós idejű képet kapjon a felhasználók által végzett tevékenységek aktuális állapotáról és ezen információ birtokában visszajelzéseket tudjon adni számukra, valamint interakcióba léphessen velük. Ezen felül valós időben módosítani tudja a rendszer attribútumait.

2.1. **Feleletválasztó példaprogram**

A gyakorlati példa egy interaktív webes alkalmazást mutat be, amelynek segítségével egy csoport vizsgázását lehet lebonyolítani feleletválasztós kérdéssor formájában.

A programnak két típusú szereplője van: a felhasználók, jelen esetben a vizsgázó diákok, valamint a kontroller, azaz a tanár, aki felügyeli a vizsgázás menetét.
Mindkét fél számára egy külön webalkalmazás elkészítése a cél, amelyek valós időben esemény-vezérelt módon fognak kommunikálni egymással, valamint a felhasználókkal.

A prototípus a következő funkciókat fogja tartalmazni:

- Felhasználó (vizsgázó): vizsgázásra bejelentkezés, kérdések megválaszolása
- Eseményfeldolgozó rendszer: a kérdések megválaszolásával eltöltött idő átlagának feltüntetése, kérdésekre adott helyes válaszok átlagának megjelenítése
- Kontroller (tanár): vizsga kiválasztása és elindítása, a diákok válaszainak nyomonkövetése, a vizsgázók valós időben történő rangsorolása az eredményeik alapján, értesítések küldése a vizsgázók felé, a vizsga hátralevő idejének módosítása, menetközben új kérdéseket létrehozása és hozzáadása

3. MEGVALÓSÍTANDÓ PROGRAM RÉSZLETES SPECIFIKÁCIÓJA

A szakdolgozatomhoz elkészített szoftver architektúrájánál moduláris felépítést fogok alkalmazni. Célom, hogy a vizsgázók és a tanárok számára is egy külön alkalmazást készítsek el, amik aszinkron módon, eseményeken keresztül kommunikálnak egymással. A rendszer kidolgozása az alábbi lépések szerint történik:

- Az architektúra megtervezése
- Technológiák kiválasztása.
- A prototípus lefejlesztése
- A rendszer tesztelése

Az alábbiakban röviden bemutatom az említett két webalkalmazást:

3.1. **Vizsgázók webalkalmazása**

A vizsgázók webalkalmazása egy böngészőből elérhető interaktív szoftver.
A kezdőfelület a rendszerbe való bejelentkezéshez kéri a diák nevét és csoportszámát.
A bejelentkezést követően megjelenik és adott időpontban elindul egy feleletválasztós vizsgasor, amit a tanár választ ki és ő adja meg a teljesítésére rendelkezésre álló időt.
A diákok egyidőben egymástól függetlenül válaszolják meg a kérdéssort.

Az alábbi képernyők elkészítése a cél:

- Kezdőképernyő: Bejelentkezés névvel és csoportszámmal
- Vizsga kérdéssor: Minden egyes kérdés egy külön felületen jelenik meg. A tovább-előző gomb megnyomásával egy másik kérdés jelenik meg. A hátralévő idő egy visszaszámláló formájában jelenik meg a felületen.
- Eseménynapló: a kontrollertől kapott eseményeket tartalmazza időrendi sorrendben

Továbbá az alkalmazás interaktív jellegéből adódóan a kérdések és a hátralévő idő nincsen rögzítve. A tanár a vizsgázás során hozzáadhat új kérdéseket, módosíthatja a hátra levő időt, valamint szöveges értesítéseket küldhet a diákok felé.

3.2. **Tanárok webalkalmazása**

A tanárok alkalmazása is egy webalkalmazás a diákokéhoz hasonlóan. A kezdőképernyő kéri a tanár nevét, valamint a csoportszámát.
Bejelentkezés után a tanár kiválaszt egy kérdéssort, majd elindítja a vizsgát. Miután a vizsga elindult, a tanár a következő három felületen keresztül követheti a vizsgázás menetét: vizsgázás, kérdések és eseménynapló. A követés valós időben történik, minden egyes válasszal a frissülnek ezek a felületek, így a kontroller minden pillanatban pontos információt kap a vizsga aktuális állapotáról.

Az alábbi képernyők elkészítése a cél:

- Kérdések: az egyes kérdésekkel eltöltött idő átlaga, diákok válaszai összegezve
- Diákok: eredményük alapján rangsorolva
- Eseménynapló: a vizsgázóktól kapott eseményeket tartalmazza időrendi sorrendben

Az alábbi funkciókat tartalmazza:

- Értesítések küldése a vizsgázók felé
- Új kérdés hozzáadása
- Hátralévő idő módosítása

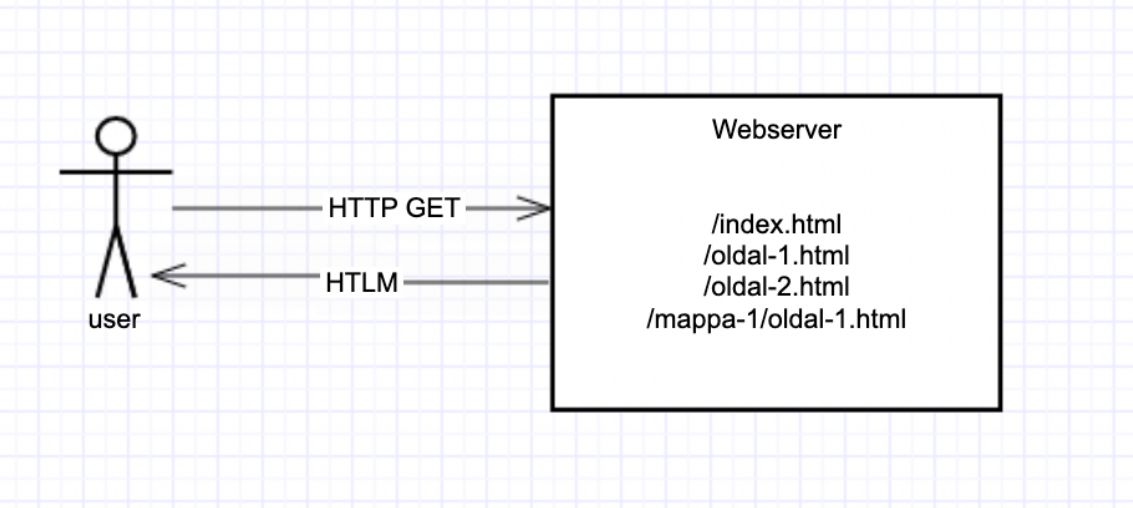
A tanárnak van lehetősége a diákok felé értesítéseket küldeni. Az értesítések megjelennek a diák webes felületén.

4. MEGLÉVŐ RENDSZEREK BEMUTATÁSA

Ebben a fejezetben a webfejlesztés leggyakrabban használt már meglévő architektúráit és fejlesztési módszereit mutatom be. A webfejlesztés rengeteget fejlődött az elmúlt két évtizedben, akár mondhatni, hogy többször is megújult.
Röviden bemutatom, a teljesség igénye nélkül, hogy az elmúlt években hogyan jutottunk el az egyszerű statikus weboldalaktól a komplexebb elosztott architektúrákig.
Szükségesnek tartom a korábban használt rendszerek megemlítését ahhoz, hogy a szakdolgozatom témáját, a valós idejű eseményfolyamok feldolgozását, megfigyelését és szabályozását, az általam bemutatott architektúrát és a benne rejlő lehetőségeket jobban megértsük. Az egyes alfejezetek végén kiemelem, hogy az adott architektúrát hogyan tudtam felhasználni a szakdolgozatomban a bemutatott probléma megoldására vagy éppen miért vetettem el azt. A fő szempontjaim a következők: alkalmas-e a szakdolgozat problémájának megoldására, amennyiben az, leírom, hogy mit használtam fel belőle, amennyiben nem, akkor megindokolom, hogy miért nem alkalmaztam.

4.1. **Statikus weboldalak**

Ez a legrégebben használt modell, a 80-as évek végén jelent meg és a 90-es években terjedt el. A statikus weboldalak fő jellemzője, hogy kizárólag olvasásra alkalmasak, a felhasználó nem tud interakcióba lépni a rendszerrel. [1]
Felépítését tekintve a modell jellemzően egy darab webszerverből áll, ami HTTP (Hypertext Transfer Protocol) kéréseket fogad és válaszként HTML (HyperText Markup Language) formátumban adja vissza a tartalmat, amit a felhasználó böngészője lefuttat és megjelenít.
A HTML fájlok a webszerver fájlrendszerén találhatók, amiket a felhasználó a szerver URL (Uniform Resources Locator) címzésén keresztül ér el.
Példakét a https://www.pelda-domain.hu/mappa-1/oldal-1.html kérés: a *pelda-domain.hu* tartománynévhez tartozó szerveren a *mappa-1* alatt található *oldal-1* fájlra hivatkozik és a tartalmát válaszként elküldi a felhasználónak.

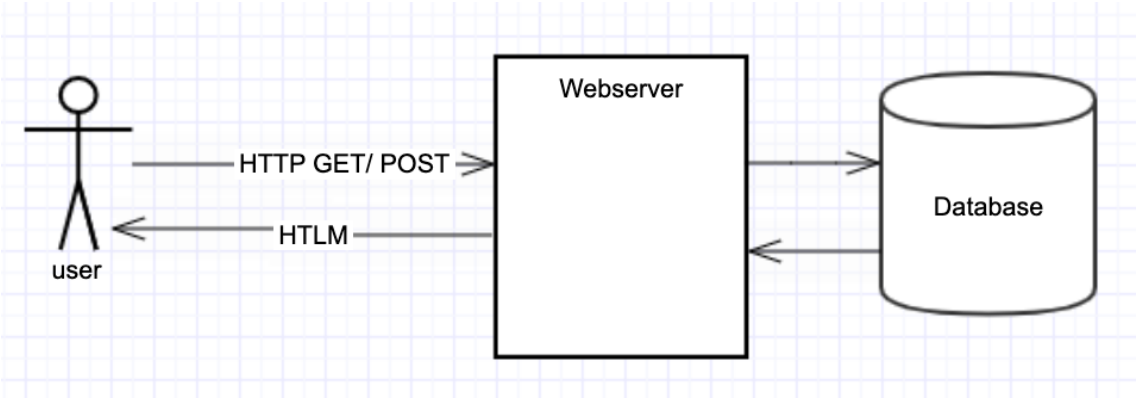


ábra 1 Statikus weboldal architektúra

A webszerver nincs interakcióban a felhasználóval, ezért nincs szükség állapotkezelésre sem. Ez a legegyszerűbb architektúra, de napjainkban már kevésbé használják, az interaktív weboldalak elterjedtebbek.
Legfőképp kisebb cégek vagy magánszemélyek weboldalánál találkozhatunk vele.
A statikus weboldal architektúra az általam fejlesztett rendszer létrehozására nem alkalmas, mivel nem ad lehetőséget arra, hogy a felhasználók interakcióba lépjenek a rendszerrel.

4.2. **Dinamikus weboldalak**

Ez még manapság is egy gyakran használt architektúra. A legfőbb különbség a statikus modellel szemben, hogy a felhasználó nem csupán adatokat fogad, hanem küld is a rendszer felé. Ehhez a dinamikus kiszolgáláshoz a rendszernek szüksége van egy adatbázisra is, amit a 2-es ábrán szemléltetek. Továbbá ez az architektúra lehetővé teszi a felhasználó számára az egyedi tartalom megjelenítését.
Ennél a felépítésnél a webszerver dinamikusan generálja a válaszul szolgáló HTML fájlt, és a kérés feldolgozása során az adatokat egy központi adatbázisból olvassa ki.
Ezen felül az adatbázis lehetőséget biztosít a felhasználó által küldött adatok mentésére.



ábra 2 Dinamikus weboldal architektúra

A felhasználók hitelesítése és az állapotkezelés a webszerver feladata. Modellünknél a leggyakrabban használt állapotkezelő módszer a **munkamenet** (session) [2] alkalmazása. A HTTP kommunikáció állapotmentes, ugyanakkor a munkamenet segítségével az alkalmazás és a felhasználó közötti állapot megőrizhető. Ez egy jelentős előrelépés az előző modellhez képest. A szakirodalomban a statikus weboldalakra gyakran Web 1.0-ként hivatkoznak, míg a dinamikusan generált tartalomra pedig már Web 2.0-ként [3].
Napjainkban leginkább fórumoknál, egyszerűbb webshopoknál használják a Web 2.0-át.
A programomban ezt az architektúrát sem tudtam alkalmazni, mivel ez a rendszer mindössze egy szerver komponensből áll, így nem használható elosztott komponensek közötti adatfeldolgozásra.

4.3. **Dinamikus kiszolgálás AJAX segítségével**

Az AJAX (Asynchronous JavaScript and XML) egy olyan technika, ami aszinkron adatátvitelt biztosít a felhasználó böngészője es a webszerver között. A gyakorlatban ennek a technikának az alkalmazása lehetővé teszi a weboldal egy bizonyos részének frissítését az aszinkron érkezett AJAX hívásból, anélkül, hogy az egész oldalt újratöltenék. Ez egy reszponzívabb, kellemes felhasználói élményt biztosít, és nagy szerepe volt a webalkalmazások elterjedésében.
Az architektúra ábrázolásában nincs nagy eltérés az előbb említett dinamikus weboldalakhoz képest, mivel jellemzően ez is egy webszerverből és egy adatbázisból áll. Az AJAX előnye, hogy nem szükséges minden felhasználói kérésnél újratölteni az oldalt, azonban a dinamikusan generált HTML tartalmat még mindig a szerveroldal hozza létre. A tartalom ebben az esetben már aszinkron frissül, azonban még mindig nem egy önállóan működő kliens alkalmazás jeleníti meg.
Fontosnak tartom megemlíteni az AJAX technikát, mivel a felhasználói élmény szempontjából egy jelentős javulást hozott, azonban architektúra szempontból továbbra is egy monolitról beszélünk.

4.4. **REST alapú elosztott rendszer**

Ez egy elosztott architektúra, ahol a rendszer több komponensből és azok interakciójából áll össze, amik egymással REST interfészen keresztül kommunikálnak.
A REST (Representational State Transfer) egy HTTP alapú kommunikációs megoldás elosztott rendszerek számára. A gyakorlatban egy szinkron kommunikációról van szó a kliens és a szerver között, ahol a kliens HTTP kéréseket küld a szerver felé és a választ jellemzően JSON formátumban kapja meg.
A REST kommunikációt alkalmazva a szerverek egy szabványos interfészt biztosítanak (más szóval API-t, Application Programming Interface) a kliensek felé, akik ezen keresztül tudják hívni a szervert. Az alkalmazás két részre oszlik, a kliens oldalán futó alkalmazásra és a szerveroldalra.

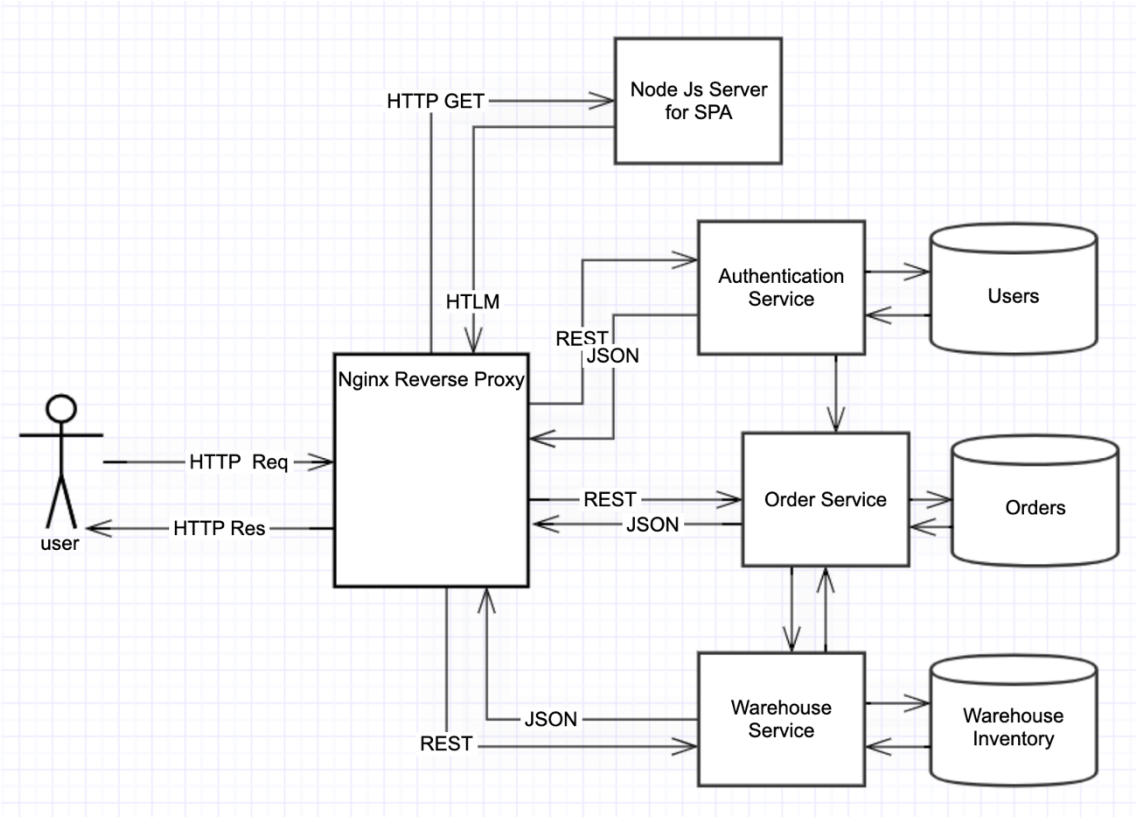
4.4.1. Kliensoldal - egyoldal alkalmazások

A REST elterjedése következtében megjelentek az elosztott webarchitektúrák és a modern JavaScript keretrendszerek, mint például az Angular vagy a React, amin keresztül egyoldal alkalmazásokat -angolul Single Page Application (SPA)- fejleszthetünk. Az SPA-k használata kiküszöböli azt a problémát, amit az AJAX technika alfejezetében említettem, mivel itt már teljesen elkülönül a kliensalkalmazás a szervertől. Gyakorlatban ez úgy néz ki, hogy az egyoldal alkalmazás egy önállóan működő entitás, amely akár különböző webszervízekhez kapcsolódhat. Ezáltal egy sokkal lazább kötésről van szó a kliens és a szerver között. Ez egy vastagabb klienst eredményez, ahol a kliens feladata, hogy biztosítsa a megjelenítést és az interakciókat a felhasználókkal. A kliens számára pedig a szerver biztosítja az adatokat. Ez az elkülönítés egy jóval komplexebb és

dinamikusabb webalkalmazás tervezését teszi lehetővé, ezért manapság a modern webfejlesztésben előszeretettel használják az egyoldal alkalmazásokat. A SPA működésének jellege, hogy a weboldal mindössze egyszer töltődik be, és az összes többi interakció a szerverrel REST alapon történik (lásd a 3. ábra alatti leírást), ez egy sokkal gyorsabb működést és ezáltal jobb felhasználói élményt biztosít.
Az általam bemutatott architektúrában a kliens komponensek SPA-k. A komponensek szétválasztása lehetőséget ad egy szabadabb tervezésre, valamint az SPA-k jobban támogatják a kétoldalú aszinkron jellegű kommunikációt (lásd a következő fejezetben).

4.4.2. Szerver oldal – elosztott rendszer

A szerver oldal további komponensekből tevődik össze, amik REST interfészen keresztül kommunikálnak.
Ez jelentős különbség az előző architektúrához képest, ahol a webszerver egyedülálló monolitként szolgálta ki az egész webalkalmazást. Az alábbi ábra például szolgál egy lehetséges REST alapú elosztott rendszer bemutatására.



ábra 3 REST alapú elosztott rendszer architektúra

Az ábrán jól látható, hogy nagyságrendekkel komplexebb architektúráról van szó az előző megoldásokhoz képest. Példaként egy kitalált webáruház architektúráját vázolom fel. Ez a rendszer a következőképpen működik: a felhasználó amikor megnyitja a weboldalt,

akkor a fordított proxy továbbítja a kérést a NodeJS szerver felé, ami visszaad egy JavaScript tartalmat. A válaszul kapott JavaScript-et lefuttatja a böngésző, és egy HTML kódot generál belőle, ami végül vizuálisan megjelenik a felhasználó képernyőjén. Az alkalmazás használata közben a böngészőben futó kliensalkalmazás REST hívásokat indít a szerver felé, az adatok kérése vagy küldése céljából. Ezeket a kéréseket a proxy az egyes szervizekhez továbbítja.
Ennél a felépítésnél jellemzően a szervízek webszerverei állapotmentesek, nem ők tárolják a működéshez szükséges adatokat és az állapotot, hanem a hozzájuk csatlakozó adatbázis. Az adatbázis használata, és az állapotmentesség lehetővé teszi azt, hogy az egyes komponensek akár több példányban is futhassanak. A komponensek általában akkor futnak több példányban amikor nagyobb terhelést kapnak. Gyakorlatban ezeket a megemelkedett számú kéréseket egy terheléselosztó (Load Balancer) menedzseli. Az ábrán ezeket az egyszerűsítés kedvéért nem tüntettem fel, de minden egyes komponens elé rajzolnék egyet egy valós rendszer esetében. Összeségében ez a működés egy nagy előnye az elosztott rendszereknek a monolit alkalmazásokhoz képest, és szinte mondhatni, hogy elengedhetetlen olyan komplex rendszerek kialakításánál, amit egyidőben akár több ezer felhasználó kezel.

Ez a modell már közelít az általam javasolt megoldáshoz. Ennek az architektúrának a legfőbb hiányossága, hogy minden hívás szinkron történik, ami oly módon teszi törékennyé a rendszert, hogy az egyes komponensek túlságosan függenek egymástól a kommunikáció jellegéből adódóan.
A csak és kizárólag REST alapon kommunikáló rendszerek nem alkalmasak valós idejű adatvizualizációra és eseményvezérelt architektúrák létrehozására, ezért a REST kommunikációt limitáltan használom a programomban. A következő fejezetben részletesen bemutatom az általam létrehozott rendszertervet, ami alkalmas a valósidejű adatfolyamok megjelenítésére és feldolgozására.

5. AZ ARCHITEKTÚRA BEMUTATÁSA

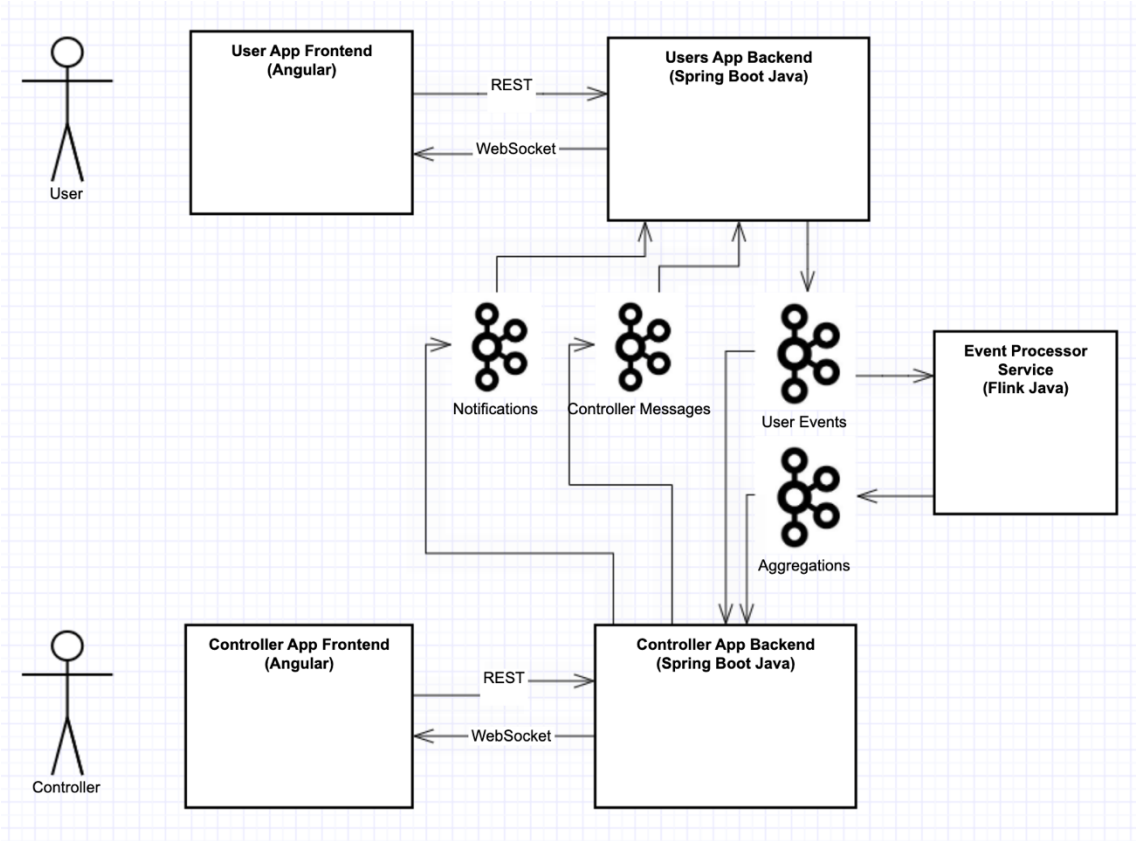
Ebben a fejezetben röviden szemléltetem az architektúrát, amit később az említett feleletválasztós alkalmazáson keresztül is bemutatok.

5.1. **Architektúra sajátossága**

Az architektúra kialakításakor fő szempontom volt többek között, hogy valós időben történjen az adatfeldolgozás, valamint nagy mennyiségű adatfolyam processzálására is alkalmas legyen az általam elképzelt rendszerterv.
Valósidejű adatáramlásnak nevezzük, amikor a rendszer folyamatosan valósidőben dolgozza fel a beérkező eseményeket, és további komponenseknek továbbítja őket.
Már a megnevezés is sejteti, hogy ezeknél a rendszereknél jellemzően az adatok feldolgozásának sebessége egy kritikus tényező. A nagy mennyiségű adatfolyam alatt pedig azt értem, hogy a feldolgozandó adatok mennyisége akár meghaladhatja a több ezer eseményt másodpercenként. Ez a két fő szempont, az adatfeldolgozás sebessége és az adatok mennyisége speciális technikákat kíván meg, mint például az adatpárhuzamos feldolgozás vagy aszinkron programozás.
Továbbá célom volt az is, hogy a megtervezett architektúrát könnyen lehessen továbbfejleszteni, bővíteni, és minél szélesebb körben felhasználni. Ezt elősegítem azzal is, hogy a feladat kidolgozásban kizárólag nyílt forráskódú szoftvereket használok, amelyeknek leginkább angol az elfogadott nyelve. Ebből adódóan az osztályokat, függvényeket és változókat angolul neveztem el és a kommentek is angol nyelven íródtak. Továbbá ezt a konvenciót követve, angol nyelvű megnevezéseket lehet találni az architektúra diagramokban is. A programkódot, illetve a telepítési útmutatót is angol nyelven írtam, az utóbbi megtalálható a forráskód readme.md fájlja alatt.

5.2. **Architektúra felépítése**

Hasonlóan a REST alapú mikroszervízekhez, ez a modell is egy elosztott felépítést követ. A könnyebb áttekinthetőség kedvéért egy egyszerűsített rendszertervet mutatok be, illetve fejlesztek le a szakdolgozatomban. Az éles környezetben használható változatot a továbbfejlesztés fejezetben szemléltetem.
A lenti diagramon ábrázolom a feladat rendszerének struktúráját, valamint az egyes komponensek közötti kommunikációt. Az ábrán a komponensekhez használt technológiák lecserélhetők, zárójelben tüntetem fel az általam választott technológiákat. Például a szerveroldali komponensek Java programozási nyelvet és Spring Boot keretrendszert használnak, de helyette használhatunk JavaScript programozási nyelvet és NodeJs keretrendszert is, vagy .NET ökoszisztémát.



ábra 4 Valósidejű eseményfeldolgozó architektúra

5.2.1. Mikroszervíz architektúra

Az ábrán bemutatott modell egy mikroszervíz architektúrát követ, ami a szolgáltatásorientált architektúra egy speciális ágazata. A rendszer kisebb önálló komponensekből tevődik össze, amik egymástól függetlenek és egymással kommunikálnak. Az egyes komponensek között laza kapcsolat áll fenn, azaz minimálisan függenek egymástól.

A mikroszervíz architektúra egyik legfőbb előnye a platform- és nyelvfüggetlen implementáció. Az egyes komponensek különböző technológiákat, adatbázisokat használhatnak a megvalósításhoz. Ez egy nagyobb szabadságot biztosít és lehetővé teszi, hogy a számunkra legoptimálisabb technológiákat válasszuk az egyes komponensek implementációja során. Erre jó példa, hogy a bemutatott architektúrában a felhasználó felületért felelős komponensek TypeScript nyelvet Angular keretrendszerrel használnak, míg a szerveroldalon lévő komponensek Java programozási nyelven íródnak és Spring Boot illetve Apache Flink keretrendszerekre épülnek. [4] [5]

5.2.2. Konzisztens felépítés

A szoftverfejlesztésben célszerű törekedni a konzisztenciára. A konzisztens rendszerek általában jobban átláthatóak és egyszerűbben követhetőek. Az architektúra tervezése során ezt igyekeztem figyelembe venni, mind a technológiák kiválasztásánál, mind a komponensek közötti kommunikációnál. Az ábrából is jól kivehető egy szimmetrikus felépítés, hiszen mindkét webalkalmazásnál az Angular keretrendszert használom, a szerver oldalon pedig a Spring Boot-ot. Ez megkönnyíti a komponensek közötti átjárhatóságot, bizonyos implementáció újrafelhasználását, illetve csökkenti a program komplexitását. A konzisztens architektúrák általában kevesebb kódot eredményeznek, a kevesebb kód pedig jellemzően lecsökkenti az esetleges hibalehetőségeket.
A gyakorlatban a konzisztencia különösen fontos, ha csapatban dolgozunk.
Bár a szakdolgozatomat egyedül készítettem, ugyanakkor a tervezésénél törekedtem arra, hogy hasonló rendszerek implementálása akár csapatban is történhessen.

5.3. Kommunikáció

Az általam bemutatott megoldás leginkább a kommunikációjában tér el a korábban említett hagyományos elosztott architektúrától. A kizárólag REST alapú kommunikációra épített rendszerek nem alkalmasak eseményvezérelt valósidejű adatfeldolgozásra és megjelenítésre. A hagyományosan REST alapú rendszerek egyik nagy hátránya, hogy a komponensek túlságosan függenek egymástól. Többek között ennek a problémának a kiküszöbölése ihlette a létrehozott architektúrát. A tervezésnél nagy hangsúlyt fektettem arra, hogy minimalizáljam az elemek közötti függőségeket.
A REST alapú kommunikáció további nagy hiányossága, hogy nem támogatja kellően a valósidejű értesítéseket. Az értesítések REST alapú implementációja úgy nézne ki, hogy a fogadó fél periodikusan GET hívásokat kezdeményezne a szerver felé, hogy megnézze, érkezett-e új értesítés az utolsó hívás óta. A periodikus lekérések között nem tudjuk pontosan, hogy mikor érkezik meg új adat, így egyrészről ezek a háttérben zajló REST hívások feleslegesen terhelik a rendszert, másrészről az esetek többségében késleltetve fogadják az adatokat.
Az általam tervezett rendszerben a következő három típusú kommunikáció jön létre a komponensek között: az adatok küldése és lekérése, felhasználók valós idejű értesítése és az eseményvezérelt kommunikáció. Az ezekhez tartozó kommunikációs megoldásokat a továbbiakban részletesen bemutatom.

5.3.2. Adatok küldése és lekérése

Az általam ismertetett architektúrában nem minden esetben használhatók a REST hívások, azonban, ahogy a 4-es ábrán is látható ezeknek is meg van a helye.
Kétféle helyzetben használunk REST hívásokat. Közös bennük, hogy mindkét alkalommal a böngészőben futó kliensalkalmazás és egy szerverkomponens között történik a kommunikáció.

Az első eset, amikor a kliens információt kér a szervertől. A prototípusomban erre példa az, amikor a tanár által használt kliens GET hívással lekéri az elérhető vizsgákat a szerver egyik komponensétől ahhoz, hogy a felhasználófelület megjelenítse az elérhető vizsgatípusokat a tanár számára.
A második eset pedig az, amikor a kliens küld adatokat a rendszernek. Erre példa az, amikor a vizsgázó megválaszol egy kérdést, vagy a kontroller értesítés parancsot küld a felhasználóknak.

5.3.3. Valósidejű értesítések

Az értesítések azonnali fogadására tökéletes megoldást nyújt a WebSocket kapcsolat.
Működését tekintve a kliens kiépít egy WebSocket-csatornát a szerverrel. Ezt követően ezen a csatornán keresztül a kliens, illetve a szerver is tudnak egymásnak adatokat küldeni, tehát egy kétirányú kommunikáció jön létre. Jelenleg a WebSocket kommunikáció HTTP/1.1 kapcsolaton alapszik.

Minden esetben, amikor a szerverkomponensek adatokat küldenek a webalkalmazás kliensei felé, WebSocket kommunikációt használok.
Egyszerre több csatorna is létrejöhet a kliensek és a szerver között. Az összes kapcsolódott kliens megkapja az üzeneteket, amelyeket a szerver a WebSocket csatornán keresztül küld. Ezért ügyelni kell, hogy minden kliens kizárólag egy WebSocket kapcsolatot alakítson ki a szerverrel, különben könnyen túlterhelhetjük a rendszert.

Az architektúrámban az alábbi esetben használom a WebSocket alapú kommunikációt:

- A feleletválasztós programban a rendszer WebSocket formában értesíti a vizsgázókat amikor a tanár kiküld egy értesítést.
- Szintén WebSocket kommunikáción keresztül értesülnek a vizsgázók, amikor a tanár elindítja a vizsgát, vagy akkor, amikor meghosszabbítja a hátralevő időt.
- A kontroller is fogad üzeneteket a WebSocket-csatornán. A tanár képernyőjén valós időben megjelenik, ha egy vizsgázó bejelentkezik.
- Továbbá minden egyes kérdés megválaszolása után a kontroller üzeneteket kap az eseményfeldolgozó modultól, a megválaszolt kérdésekkel kapcsolatos információkkal és statisztikákkal.

5.3.4. Eseményvezérelt kommunikáció

A komponensek közötti függőségek csökkentésére az eseményvezérelt kommunikációt használom megoldásként.
Felépítését tekintve egy feladó félből es az eseményekre feliratkozó fogadófelekből áll.
A kommunikáció a következőképpen működik: a feladó komponens a feldolgozási folyamat közben eseményeket küld, amik egy eseménytovábbító rendszeren keresztül eljutnak a feliratkozó komponensekhez.

A 4-es ábrából is észrevehető, hogy az architektúrában az egyes szerveroldali komponensek nincsenek közvetlen kapcsolatban egymással, mivel közöttük az Apache Kafka helyezkedik el, ami az események fogadásáért, tárolásáért, illetve továbbításáért felel.
Működését tekintve a feladó egy bizonyos Kafka topic-ba küldi el az üzenetet, ami közvetítő szerepet játszik, és azokat az összes feliratkozónak valós időben továbbítja.
A kommunikáció TCP (Transmission Control Protocol) alapon működik, ahol az átvitt üzenet binárisan van szerializálva mind a feladó és a Kafka között, mind a Kafka és az üzenet fogadó között.
Ez a megoldás egy sokkal lazább kötést biztosít, ami lehetővé teszi új feliratkozó komponensek hozzáadását, anélkül, hogy a feladó komponensét módosítani kéne. Ez a tulajdonság különösen fontos lehet, nagyobb, több fős projekteknél, ahol nem ritka, hogy az üzenetfeladó rendszerét egy másik csapat birtokolja. Ezáltal a komponensek programkódja egyszerűbb és kevésbé törékeny, mint a szinkron kommunikáció esetén, ugyanis ott a feladónak egyesével értesítenie kéne a komponens minden egyes feliratkozóját.

Az eseményvezérelt kommunikáció egy másik előnye az aszinkronitás. Ezzel ellentétben a REST hagyományosan szinkron kommunikációt biztosít. Ez a gyakorlatban azt jelenti, hogy a hívott fél a feldolgozás végén visszaválaszol a hívó félnek, amit a hívó fél megvár, másszóval a hívás blokkolja a további feldolgozást amíg a válasz meg nem érkezik. Eközben könnyen előfordulhat, hogy a hívott fél egy további REST hívást kezdeményez egy harmadik komponens felé, amit szintén meg kell várnia a hívónak. Egyszerűen be lehet látni, hogy ez egy láncreakcióhoz vezethet, és a hívási lánc bármikor megszakadhat, ha az egyik komponens nem elérhető.
Az eseményvezérelt aszinkron hívásoknál az üzenetküldő komponens nem áll közvetlen kapcsolatban az üzeneteket fogadókkal, ezért a választ sem kell megvárnia a további feldolgozáshoz. A másik nagy előnye, hogy az üzenetek tárolva vannak a Kafka-ban, tehát ha az egyik feliratkozó átmenetileg nem elérhető, vagy le van terhelve, akkor később is, amikor újra elérhető, megkapja a régebben küldött, de még nem feldolgozott üzeneteket.

5.4. **A rendszer szereplői**

A modellünk kétszereplős. Az egyik fél a felhasználók csoportja, akik a rendszert használják és általa eseményeket generálnak. A másik fél pedig a kontroller, aki felügyeli a rendszer működését, és szükség esetén szabályozza azt.

Felhasználó (vizsgázó)

A vizsgázó a rendszer alapértelmezett felhasználója. Jellemzően egy időben több vizsgázó használja a rendszert a Felhasználó alkalmazáson (User App) keresztül.

Felhasználó műveletei:

- Bejelentkezés
- Vizsgakérdés megválaszolása

Kontroller (tanár)

A tanár reprezentálja a kontroller felet, és a feladata, a vizsgázás felügyelése és irányítása a kontroller alkalmazáson (Controller App) keresztül.

Tanár műveletei:

- Bejelentkezés
- Vizsga kérdéssor kiválasztása
- Értesítés küldése
- Parancsok küldése (új kérdés hozzáadása, hátralevő idő meghosszabbítása)

5.5. **Komponensek**

A komponensek a rendszer alkotóelemei. Ez egy rövid ismertetés, az egyes komponensek részletes bemutatását a következő fejezetekben tárgyalom.

Felhasználó App (Frontend)

A vizsgázó által használt webes felület, amin keresztül interakcióba lép a rendszerrel és megválaszolja a feltett kérdéseket.

Felhasználó App (Backend)

A felhasználói alkalmazás szerveroldali komponense. Elsődleges célja a vizsgázó által megválaszolt kérdések kiértékelése és továbbküldése elemzésre. Továbbá feladata a kontroller parancsainak végrehajtása (új kérdések létrehozása, hátralévő idő módosítása), valamint az általa küldött értesítések továbbítása a vizsgázók felé.

Vizsgázó által generált események

- kérdés megválaszolva
- vizsga befejezése

Eseményfeldolgozó komponens (Event Processzor)

Ez a komponens felel a válaszok összegzéséért és átlagolásáért. Minden egyes vizsgakérdés megválaszolása után az eseményfeldolgozó komponens frissíti az aggregált események állapotát és az újonnan kiszámított értéket egy üzenet formájában továbbítja.

Vizsgára adott válaszokból létrehozott aggregált események (Aggregations)

A következő aggregált eseményeket hozza létre a feldolgozó komponens:

1. Kérdés követése – az esemény tartalmazza az adott kérdésre megadott válaszokat százalékos lebontásban, az adott kérdés megválaszolásával eltöltött idő átlagát, illetve a helyes válaszok átlagát kérdésenként
2. Diákok követése – az esemény tartalmazza a diákok minden válaszát, az egyes kérdésekkel eltöltött idejüket, a helyes megoldásaik arányát

Kontroller Alkalmazás (Frontend)

A tanár által használt webes felület. Ezen keresztül felügyeli a tanár a vizsga menetét. Valós időben ábrázolja az aggregált eseményeket, így pontosan tudja, melyik diák teljesít jobban, rosszabbul, valamint melyek azok a kérdések, amelyek könnyebben, illetve nehezebben megoldhatók.

Kontroller App (Backend)

Szerver oldali alkalmazás, ami a kontroller felület számára biztosítja az adatokat. Továbbá a kontrollertől fogadott parancsok és értesítések továbbításáért felel.

Értesítések

A kontroller által küldött értesítések, amik valós időben jelennek meg a felhasználó oldalán.

Kontroller parancsok

Hasonlóan az értesítésekhez, a kontroller küldhet parancsokat, amik a felhasználók által használt rendszer attribútumait módosítják.

5.6. Az architektúra felhasználása

Az architektúra felhasználható olyan többszereplős elosztott rendszereknél, ahol egymástól független szereplők küldenek adatokat a rendszernek és ezt a folyamatot egy kontroller fél valós időben felügyeli és szabályozza. A következő néhány példa által szeretném bemutatni milyen típusú alkalmazásokat lehetne az architektúrát felhasználva lefejleszteni.

5.6.1. Tőzsdei kereskedő asszisztens

A tapasztalt tőzsdei kereskedő betanított asszisztenseivel szeretne kereskedni a kriptopénz-piacon. Az alkalmazás, amin keresztül az asszisztensek kereskednek, esemény formában adatokat továbbít az adás, illetve vételi tranzakciókról a rendszernek. Ezeket az adatokat feldolgozás után a kontroller valós időben látja egy webalkalmazáson keresztül. A felület ábrázolja, mely asszisztensek kerestek a legtöbb pénzt, illetve mutatja, hogy mely valuta párok kereskedése a leginkább nyereséges. Ezáltal a szenior kereskedő (kontroller) szabályozhatja, hogy a program mely valutapárok kereskedését engedje, valamint értesítéseket küldhet az asszisztensek felé.

5.6.2. Online edző asszisztens

Az alkalmazás az edzőt egy csoportos online edzés lebonyolításában segíti. Az edző célja, hogy az edzés minél hatékonyabb legyen, a sportolók kellően lefáradjanak, de ne legyenek túledzve a másnapi verseny előtt. A rendszer felhasználói a sportolók, akik egy okosóra segítségével bizonyos időközönként küldik az aktuális pulzus számukat a rendszernek. A kontroller pedig az edző, aki egy táblagépen futó webalkalmazáson látja, mennyire vannak a sportolók lefáradva és melyek azok a sportolók, akik legjobban bírják az edzést. Az edző ezeknek az információknak a tudatában szabja a csoportra az edzésprogramját, tart rövidebb, illetve hosszabb pihenőket. Az edző az utasításokat a webalkalmazásokon keresztül küldi, amit a versenyzők az órájukon keresztül fogadnak.

5.6.3. Kocsma 2.0

A bárban ülő vendégek táblagépen keresztül adják le a rendeléseket, ők a rendszer felhasználói. A kocsmának az a célja, hogy minél többet költsenek a vendégek, de azért a részeg vendégek ne okozzanak károkat. A bár tulajdonosa a kontroller, aki egy

webalkalmazáson keresztül követi, milyen rendeléseket adtak le. Az alkalmazás szemlélteti melyek italokból rendeltek a legtöbbet és hogy melyik asztal hagyott ott legtöbb pénzt. Ezeknek az információknak a birtokában a kocsma tulajdonos kedvezményeket, új akciókat tud létre hozni ösztönözve a további fogyasztást, vagy akár szüneteltetni bizonyos italok rendelését.

6. ALKALMAZOTT ESZKÖZÖK ÉS TECHNOLÓGIÁK BEMUTATÁSA

Ebben a fejezetben ismertetem a szakdolgozat implementálásához használt technológiákat és metodikákat.

6.1. **Programozási nyelvek, könyvtárak és keretrendszerek**

Az egyes komponensek fejlesztéséhez különböző programozási nyelveket, illetve keretrendszereket használtam fel. Ezeknek a működését az alábbiakban röviden ismertetem.

6.1.1. Java

Egy széles körben felhasznált objektumorientált programozási nyelv. A Java programok futási ideje versenyezik a C illetve C ++ íródott alkalmazásokkal. A működésének lényege, hogy a fordító a Java alapú alkalmazásokból bájtkód formátumú *.jar* kiterjesztésű fájlt hoz létre. A bájtkódot pedig futási időben a Java virtuális gép (továbbiakban: JVM - Java Virtual Machine) a processzor számára értelmezhető, gépi kóddá alakítja. Ez lehetővé teszi a Java alkalmazások futását minden olyan platformon, ahol a java virtuális gép telepítve van. A mai modern JVM-ek olyan szinten optimalizáltak, hogy a Java programozási nyelven írt alkalmazások futási sebessége versenyezik az alacsonyabb szintű nyelvekkel.
A szakdolgozat programjának a megírásához számomra a Java programozási nyelv bizonyult a legmegfelelőbb választásnak, részben a JVM magas teljesítményének köszönhetően. Másrészről pedig sok nyíltforráskódú alkalmazás áll rendelkezésre az elosztott webalapú rendszerek fejlesztésére és a valós idejű eseményfolyam feldolgozás programozására. A feladat fejlesztésénél a szerveroldalon, illetve a rendszertesztelésénél használom. [6]

6.1.2. Spring Boot

A Spring az egyik legelterjedtebb nyílt forráskódú ökoszisztéma a Java programoknál.
Egyik fő tulajdonsága, hogy a konvenciót részesíti előnyben a konfigurációval szemben. Ez egy módszertan, ami röviden arról szól, hogy a Spring alapértelmezetten konfigurációkat és megoldásokat nyújt a leggyakrabban használt problémákra, mint például az adatbázishoz levő kapcsolódás, vagy tranzakciók kezelése. Szükség esetén lehetőséget biztosít a konfiguráció tetszőleges felülírására is. Ez a módszertan jelentősen meggyorsítja a fejlesztést. [7]
A Spring Boot célja, hogy egyszerűvé tegye a Spring alapú alkalmazások fejlesztését. Jellemzően a gyors alkalmazásfejlesztésre (RAD - Rapid Application Development) lett megalkotva, ezért manapság az agilis szoftverfejlesztésben sűrűn használják.

A Spring Boot, alapértelemezetten tartalmaz egy beépített alkalmazásszervert, ezáltal könnyedén létrehozhatunk webszervízeket. Ehhez csupán el kell indítani a lefordított Spring Boot alapú alkalmazás futtatható *.jar* fájlját. Ez leegyszerűsíti az alkalmazás telepítését, ezért előszeretettel használják a mikroszervíz architektúráknál, ahol több webszervíz kommunikál egymással. [8]

6.1.3. Netty

A SpringBoot alapértelmezetten Tomcat-et használ alkalmazásszerverként. Az általam lefejlesztett architektúra SpringBoot komponensei jellemzően aszinkron kommunikálnak a többi komponenssel WebSocket illetve Kafka topic-on keresztül. Mivel a Netty felépítéséből adódóan jobban támogatja ezt a típusú kommunikációt, mint a Tomcat, így az architektúra kontroller és a felhasználó szerveroldali komponensei Netty alatt futnak.
A Netty egy keretrendszer, ami implementálja a Reactor programtervezési mintát, ezáltal párhuzamos, és aszinkron kommunikáló alkalmazások fejlesztését teszi lehetővé. Biztosítja és jelentősen leegyszerűsíti a TCP [9] illetve UDP [10] kommunikációt, ami feltétele az aszinkron üzenetküldésnek.
A Netty használatát megkönnyíti az is, hogy a Spring Boot egy beépített integrációt biztosít hozzá. Ezáltal minimális változtatásokkal, illetve konfigurációval a Spring Boot alapú szervíz átalakítható úgy, hogy Tomcat helyett Netty-t használjon. [11] [12]

6.1.4. Lombok

A Lombok egy segédkönyvtár, ami fordítási időben legenerálja az osztály mezőihez tartozó getter, setter, konstruktor, builder és egyéb hasznos függvényeket. A lombok használata olvashatóbb rövidebb forráskódot eredményez és szintén segített meggyorsítani a fejlesztést.

6.1.5. TypeScript

A TypeScript a JavaScript egy speciális típusokkal kiterjesztett változata. A modern böngészők szinte kivétel nélkül támogatják JavaScript programozási nyelvet. Mivelhogy a TypeScript a JavaScript egy kiterjesztése, ezért egyszerűen lefordítható a böngésző számára is értelmezhető JavaScript-re. A JavaScript-el ellentétben a nyelv típusos jellegéből adódóan fordítási időben visszajelzést kapunk a típushibákról. Napjainkban használt legelterjedtebb webes technológiák támogatják és javasolják a TypeScript használatát. Az implementáció során a kliens oldali komponenseknél használom.

6.1.6. Angular

A felhasználó felület programozásához használt TypeScript-ben írt nyílt forráskódú modern keretrendszer. Erőssége a komponens alapú fejlesztés, általa könnyedén létrehozhatunk több helyen felhasználható, építőkövekként szolgáló komponenseket. Az Angular Material előre legyártott komponenseit- többek között a táblázatok, kérdés kártyák, gombok, menüsor- felhasználtam a webalkalmazások implementációjánál.
Továbbá az Angular jó támogatást biztosít a WebSocket kommunikációhoz és az RxJs (Reactive Extensions for JavaScript) könyvtár által a kétoldalú adatvezérléshez. Ezen érvek mentén a bemutatott rendszernél a webmodulok implementációjához az Angular keretrendszert használom.
[13] [14]

6.1.7. Apache Kafka

A Kafka egy elosztott eseménytár, ami üzeneteket fogad, tárol és valós időben továbbít az üzenetekre feliratkozók számára. Eredetileg a LinkedIn által lett kifejlesztve, de ma már nyílt forráskódú. A Kafka az eseményeket redundánsan partíciókban tárolja, ezáltal az eseményekre feliratkozó kliensalkalmazások párhuzamosan dolgozhatják fel az eseményeket.
A például szolgáló programban az Apache Kafka-t használom kommunikációs csatornaként a komponensek közötti esemény vezérelt kommunikációhoz, továbbá a rendszerben használt adatáramlásnak is a szerves része.
[15]

6.1.8. Apache Flink

A Flink az eseményfolyamok feldolgozását segítő nyílt forráskódú rendszer. A Flink Java illetve Scala programozási nyelveken íródott. Napjainkban egyre népszerűbb, mivel jelentősen megkönnyíti az adatfolyamok programozását azáltal, hogy egy könnyen használható keretrendszert biztosít.
Példánkban a valós idejű adatfolyam feldolgozást a Flink rendszerre épülő komponens végzi. Az alkalmazásban a vizsgázó válaszainak összegzéséért, aggregálásáért és transzformálásáért felel. [16]

6.2. Felhasznált programozási paradigmák

Röviden bemutatom az implementáció során az általam alkalmazott programozási paradigmákat.

6.2.1. Objektum Orientált Programozás

A Java egy objektum orientált programozási nyelv. A Java-ban írt programok osztályokból, objektumokból és a köztük levő interakciókból tevődik össze. Az objektum orientált programozás alappilléréi az egységbezárás, öröklődés és a többalakúság.
[17]

6.2.2. Funkcionális programozás

A funkcionális programozás egy programozási paradigma, ami a függvények köré épül. Eredetileg a Java egy tisztán Objektum Orientált nyelvként indult, ugyanakkor a napjainkban használt modern változat tartalmaz funkcionális elemeket, mint például a Stream API, vagy a Lambda függvények, amiket előszeretettel használtam az adattranszformációk során.
A felhasználó felület komponenseinek programozásánál pedig a reaktív RxJs funkcionális könyvtárat használtam az adatok vezérlésére.
[18]

6.3. Függőségek kezelése

6.3.1. Apache Maven

A Maven egy olyan szoftver, amit a fordítási folyamatok automatizálására, illetve forráskód által használt függőségek importálására használok a szerveroldali komponenseknél.
[19]

6.3.2. NPM (Node Package Manager – csomagkezelő)

Hasonlóan a *Maven*-hez, az *NPM* is a program működéséhez szükséges csomagok importálásáért felel. Az *NPM*-et a Javascript és a TypeScript alapú webkomponensek menedzselésére használom.
[20]

6.4. Forráskód tárolás és verziókövetés

A szoftverfejlesztésben a forráskód tárolására célszerű verziókövető programot alkalmazni. A verziókövető használatával könnyedén készíthetünk biztonsági mentéseket a programunkról, továbbá segíti a fejlesztők közötti kommunikációt, mivel lehetővé teszi

a forráskód központi helyen levő tárolását. Napjainkban a **Git** a legelterjedtebb verziókövető program.

6.4.1. Git és Gitlab

Példánkban a *Git* programot használjuk verziókövetésre, a forráskódot pedig a GitLab felhő alapú rendszerben tároljuk. [21]

6.5. **Telepítés (Docker)**

A telepítéshez a legelterjedtebb konténeralapú virtualizációs technológiát, a **Docker-**t használom. A Docker szintén egy nyílt forráskódú projekt, ami a Linux konténerekre épít, és azt kibővítette.

A Docker egyik legfőbb előnye a szállíthatóság. A konténerbe csomagolt komponensek bármilyen környezetben futtathatók, ahol a Docker Engine telepítve van.

Erről a technológiáról részletesebben írok a telepítés fejezetben. [22]

7. ADATMODELL

Ebben a fejezetben az alkalmazás adatmodelljét mutatom be. Az elosztott rendszereknél külön kihívás az adatsémák szinkronizációja. Ugyanakkor a helyes működéshez elengedhetetlen, hogy minden komponens ugyanazt az adatsémát használja. A komponensek közötti kommunikáció során az adatokat küldő fél binárisan, vagy karakterlánc formában szerializálja a küldött adatokat. A fogadó fél pedig a megérkezett üzeneteket visszafelé konvertálja (de-szerializálja) objektumokká. Ez a gyakorlatban annyit jelent, hogy ha a küldő és a fogadó fél nem ugyanazt a sémát használja, akkor a fogadó nem tudja fogadni az üzenetet. Az elosztott rendszereknél ez egy gyakori probléma, aminek kiküszöbölésére egy közös könyvtár által nyújtott megoldást alkalmaztam. További nehézséget nyújt, hogy a webalkalmazás és a szerver oldali különböző programnyelvet használ.

7.1. **Közös programkönyvtár (common-service)**

Létrehoztam a *common-service* nevű programkönyvtárat, ami tartalmazza a rendszer összes nyilvános adatmodelljét. Ezeket az adatmodelleket az elosztott rendszer számos komponense használja. Megjegyezném, hogy az egyes komponenseknek lehetnek egyéb privát adatmodelljei, de azok csak és kizárólag a komponens belső működésére szolgálnak.
Ahhoz, hogy a Java alapú szerveroldali komponensbe importálhatók legyenek a programkönyvtár modelljei, a *common-service* könyvtárat egy *.jar* kiterjesztésű fájlra fordítottam le. A könyvtárfüggőség hivatkozásnál ügyelnünk kell arra, hogy a könyvtárat használó összes komponens a programkönyvtár ugyanazon verzióját használja. A szerver oldali komponensekkel szemben a felhasználói felület programozása Typescript nyelven Angular alapú keretrendszerben történt.
Sajnos az Angular alapú komponensekkel nem kompatibilis a java alapú *common-service* programkönyvtár. Erre a problémára megoldásként a GitHub-on talált *typescript-generator-maven-plugin* nyílt forráskódú plugin-t használom.

7.2. **typescript-generator-maven-plugin**

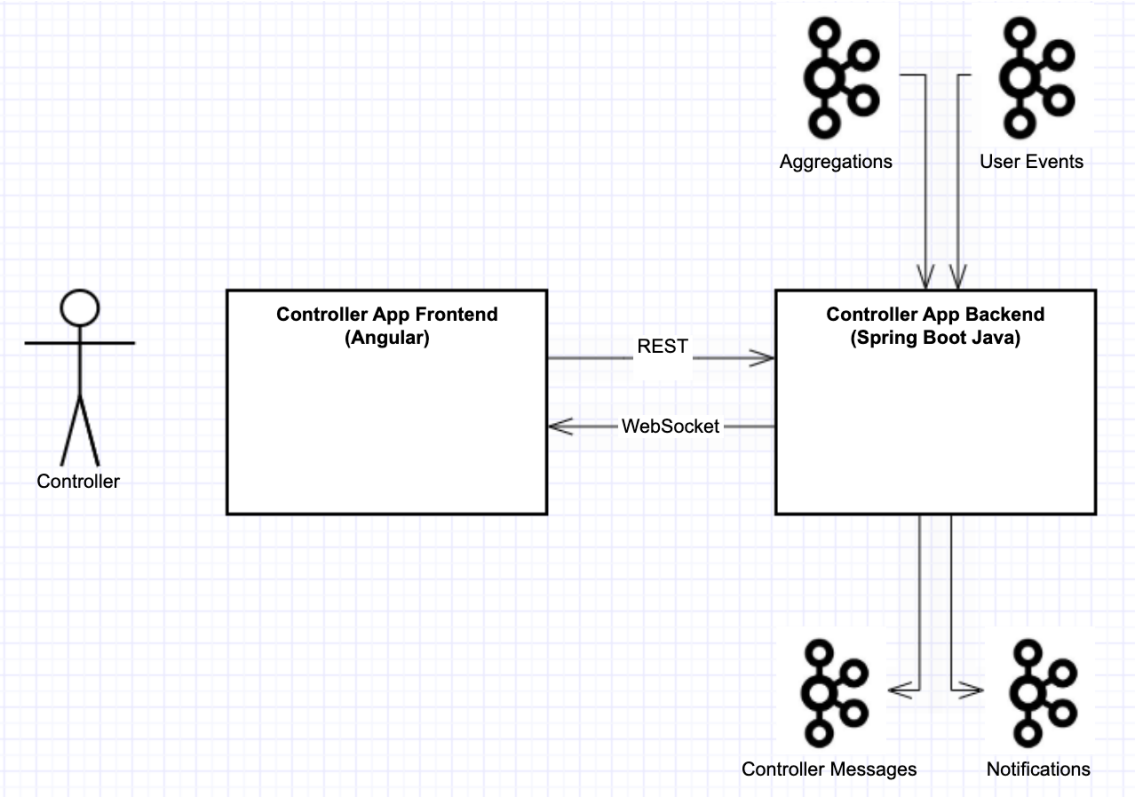
A *typescript-generator-maven-plugin* a *common-service* fordítási folyamata közben a Java osztályokból legenerálja a megfelelő TypeScript modelleket. Ezek a modellek egy generált *.ts* kiterjesztésű fájlban találhatóak, ami pedig már bemásolható a frontend modulokba.

8. KONTROLLER ALKALMAZÁS

A kontroller alkalmazáson keresztül a vizsgáztató irányítja, illetve felügyeli a vizsgázási folyamatot. Az alkalmazás egyik fő tulajdonsága, hogy a vizsgáztató számára folyamatosan adatot szolgáltat a vizsgázás menetéről.

8.1. Felépítése

A kontroller program két komponensből áll össze, a Frontend és Backend komponensből, ami az ábrából is jól látható.



ábra 5 Kontroller alkalmazás felépítése

8.1.2. Webalkalmazás

Először a webalkalmazást mutatom be. A program az Angular keretrendszeren alapul. Az Angular a Typescript programnyelvet használja, ugyanakkor a böngésző mindössze HTML, CSS (Cascading Style Sheets) és JavaScript kódot tud interpretálni. Ennek kiküszöbölésére a telepítés előtt a Typescript alapon írt Angular alkalmazást a böngésző számára értelmezhető JavaScript és HTML fájlokká fordítom le az NPM (Node Package Manager) segítségével.

A webes felület a Google által létrehozott és népszerűsített materiál dizájnt és beépített komponens gyűjteményt használ, ami egy egységes, modern felhasználói élményt biztosít a kontroller számára. Továbbá a fejlesztésnél törekedtem a reszponzív megjelenítésre. Ezt a flexbox által nyújtott CSS-n keresztül értem el. Így az alkalmazás egyaránt használható különböző méretű eszközökön: telefonon, táblagépen, laptopon, vagy akár széles kijelzőjű monitoron is.
Fontosnak tartom megjegyezni, hogy a kontroller webalkalmazás kizárólag a szerver oldali kontroller szerverrel kommunikál. Ez architektúrai szempontból kevesebb függőséghez vezet.

8.1.3. Szerveroldali komponens

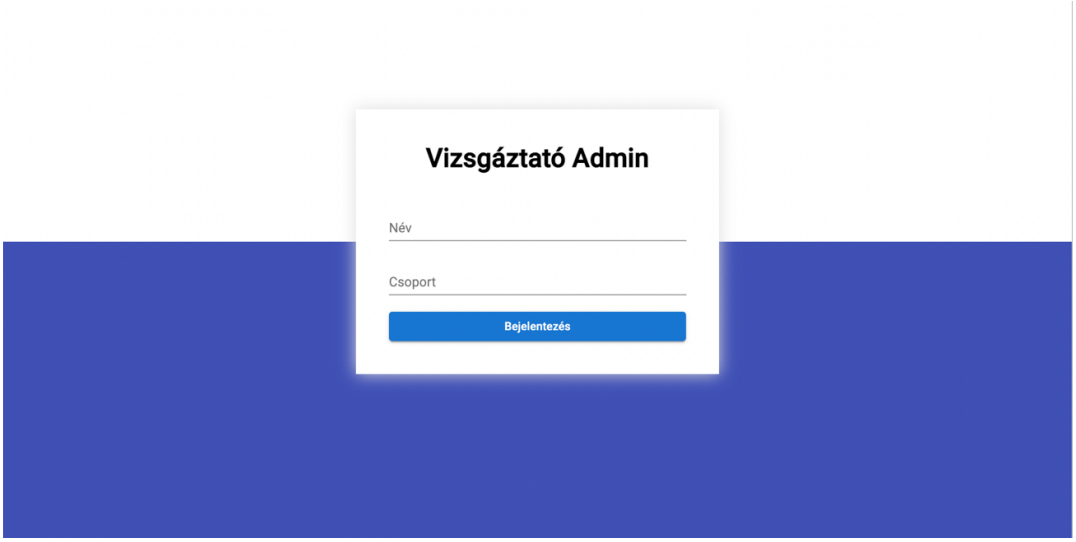
A másik modul a webalkalmazáshoz társuló szerver oldali komponens.
Ez Java programnyelven íródott a Spring keretrendszer előnyeit felhasználva. Az alkalmazás a Netty által biztosított aszinkron környezetben fut, ami eltér a gyakorlatban leginkább használt MVC (Model View Controller) alapú webes rendszerek programozásától. A Netty környezet jellemzően sokkal kevesebb feldolgozó szálat biztosít a kérések kiszolgálására. Ezáltal törekednem kellett, hogy az implementáció során ne blokkoljam az adatot feldolgozó szálakat mivel abban az esetben a program nem tudna párhuzamosan több kérést kiszolgálni. A Netty használata azért volt szüksége, mivel az alkalmazás WebSocket-en keresztül küldi az üzeneteket a felhasználó böngészőjén futó kliens alkalmazásnak, valamint mert a komponens aszinkron módon, Kafka üzeneteken keresztül, kommunikál a rendszer többi részével.

8.2. **Funkcionalitása**

Ebben az alfejezetben képernyőképek segítségével bemutatom a kontroller alkalmazás működését.

8.2.1. Bejelentkezés

A bejelentkező képernyő a tanár nevét, illetve a csoport azonosítóját kéri.



ábra 6 Bejelentkezés képernyő

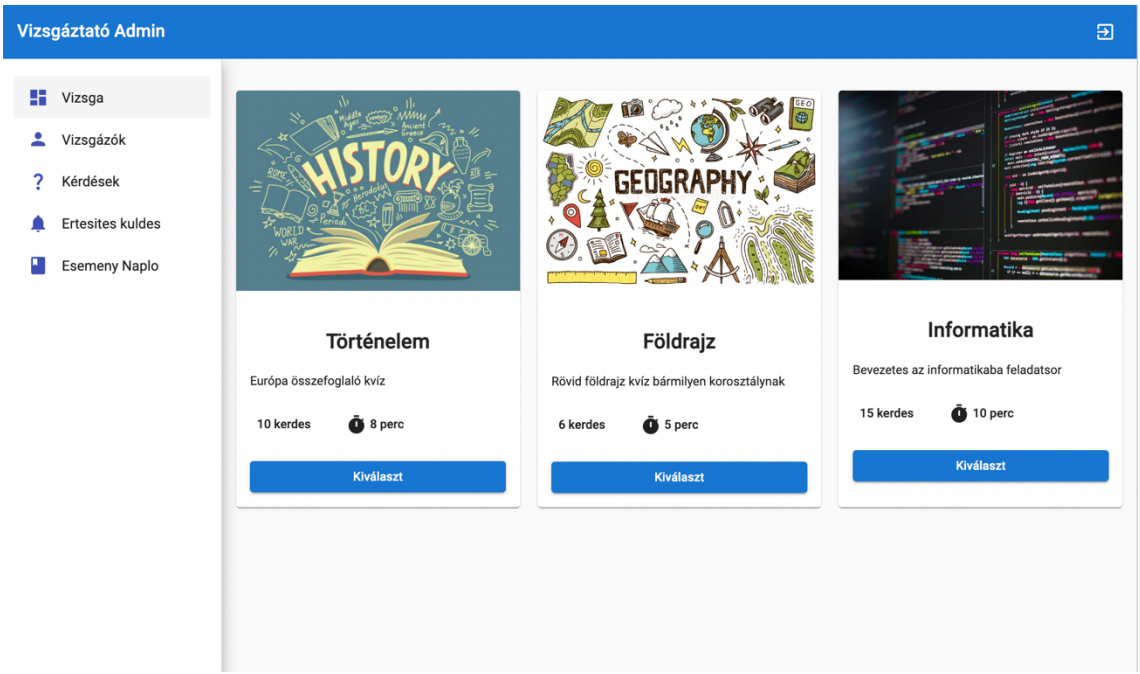
A Bejelentkezés gomb megnyomása után megjelenik a vizsga választó felület.
A háttérben pedig a webalkalmazás süti formában eltárolja a kontroller nevét, illetve a csoport azonosítóját és kiépíti a WebSocket kapcsolatot. A süti használatával oldottam meg, hogy a WebSocket kapcsolatnál a szerver oldalon azonosítsam a kontrollert, illetve a csoportot.
A jelenlegi implementáció nem tartalmaz hitelesítést, mivel az a szakdolgozat terjedelmén már túlmutat. Ugyanakkor kitérek egy lehetséges megoldásra a továbbfejlesztés fejezetben.

8.2.2. Navigáció

A belépést követően a kontroller alkalmazás vizsga választó felülete látható. Ahogy a 7. ábrán látható, az oldal felépítését tekintve három részből áll össze. A felső kék sáv tartalmazza az alkalmazás nevét és a kijelentkezés gombot a jobb sarokban.
Baloldalon látható a navigációs menü. A menüpont kiválasztását követően a kiválasztott pontnak megfelelő tartalom jelenik meg a felületen.
Az alkalmazás használata közben a tanár tetszőlegesen navigálhat a menüpontok között.

8.2.3. Vizsgaválasztás

A vizsgát a vizsgamenüponton keresztül lehet kiválasztani. A megjelenített vizsgákat a böngésző egy REST híváson keresztül kéri le az alkalmazásszervertől.
A képernyőmentésből is látható, hogy minden vizsgának van neve, leírása. Továbbá minden vizsga tartalmazza a hozzárendelt kérdéseket, és a vizsgázásra rendelkezésre álló időt.



ábra 7 Vizsgaválasztás képernyő

A vizsgát a kiválaszt gombbal elindítja el a tanár. A gombnyomást követően a böngésző egy REST parancsot küld az alkalmazás-szerver felé, ami továbbítja a parancsot a *Controller messages* kafka topic-ba, és végül a parancs üzenet elindítja a vizsgát a bejelentkezett vizsgázóknál.

8.2.4. Vizsgázás nyomonkövetése

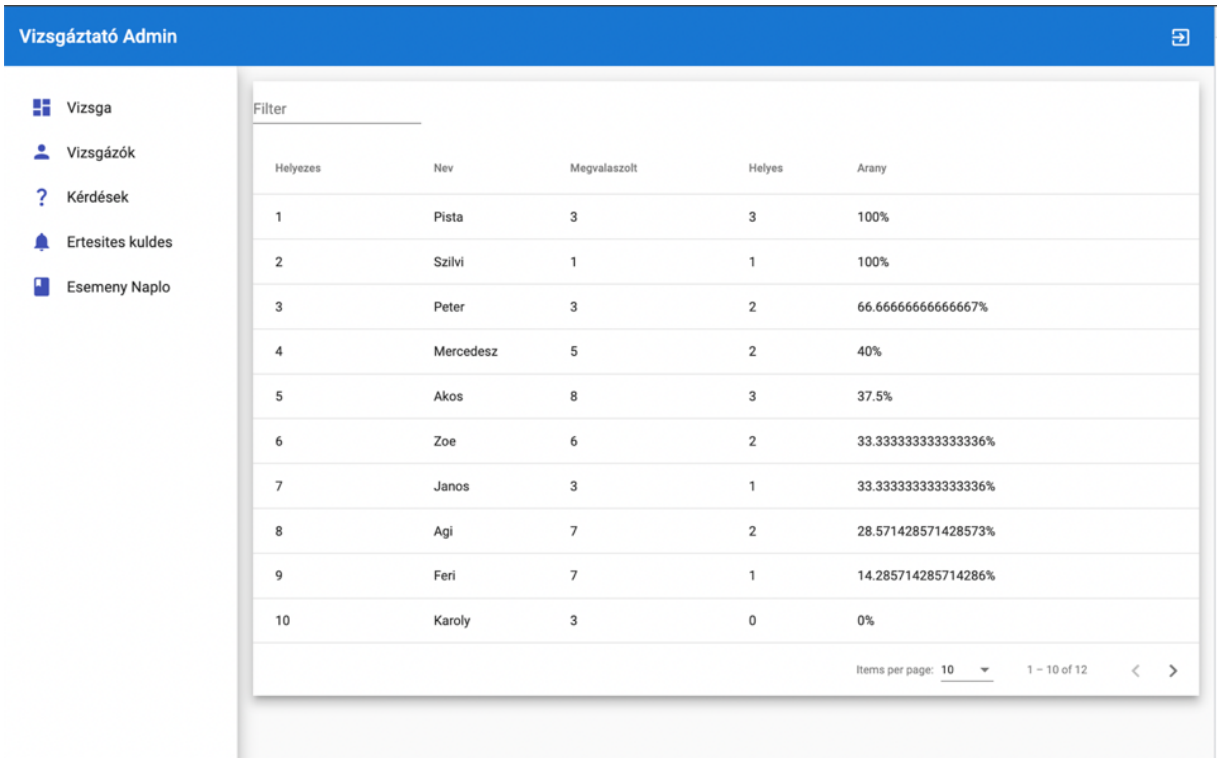
A webes felületen keresztül követheti a kontroller felhasználó a vizsga menetét, ezeket a vizsgázók, kérdések, illetve az eseménynapló menüpontokon keresztül éri el, amiről a következő alfejezetekben részletesen beszélek.

A vizsgázás nyomonkövetése során az adatáramlás menete a következőképpen működik. Az üzenetek valós időben érkeznek a kiépült WebSocket-csatornán. Minden üzenet érkezését követően a felhasználó felület automatikusan frissül. Ezeket az üzeneteket az alkalmazás szerver aszinkron módon a *User Events* illetve az *Aggregations* Kafka topic-ból kapja. Ezeket a szerver, mint egy proxy szerepet játszva, továbbítja WebSocket-csatornákon keresztül a kontroller böngészőjéhez.

8.2.5. Diákok felügyelése

Egy dinamikusan frissülő táblázat, ami a Vizsgázók menüpont alatt érhető el.

A diákok megoldásait a rendszer rangsorolja, így a tanár valós időben látja, hogy kik azok, akik jobban, illetve kik azok, akik rosszabbul teljesítenek a vizsgázás során.
Erre megoldásként az Angular-Material egy beépített táblázatát használom.
Alapértelemezetten a felület a legjobban teljesítő 10 diák adatait jeleníti meg.
A feltüntetett oszlopok tartalmazzák a vizsgázók helyezését, nevüket, összesen megválaszolt kérdéseik számát, a helyes válaszok számát, illetve egy arányszámot, ami a helyes válaszok arányát mutatja. Ezek dinamikus értékek, új események érkezése után a táblázat automatikusan befrissül. A jobb alsó sarokban található nyilakra való kattintással a táblázat lapozható, így a többi vizsgázó eredményei is megtekinthetőek. A felület ezen felül tartalmaz egy szűrőt a bal felső sarokban (filter), ennek a segítségével a tanár név szerint is rákereshet az egyes vizsgázókra.

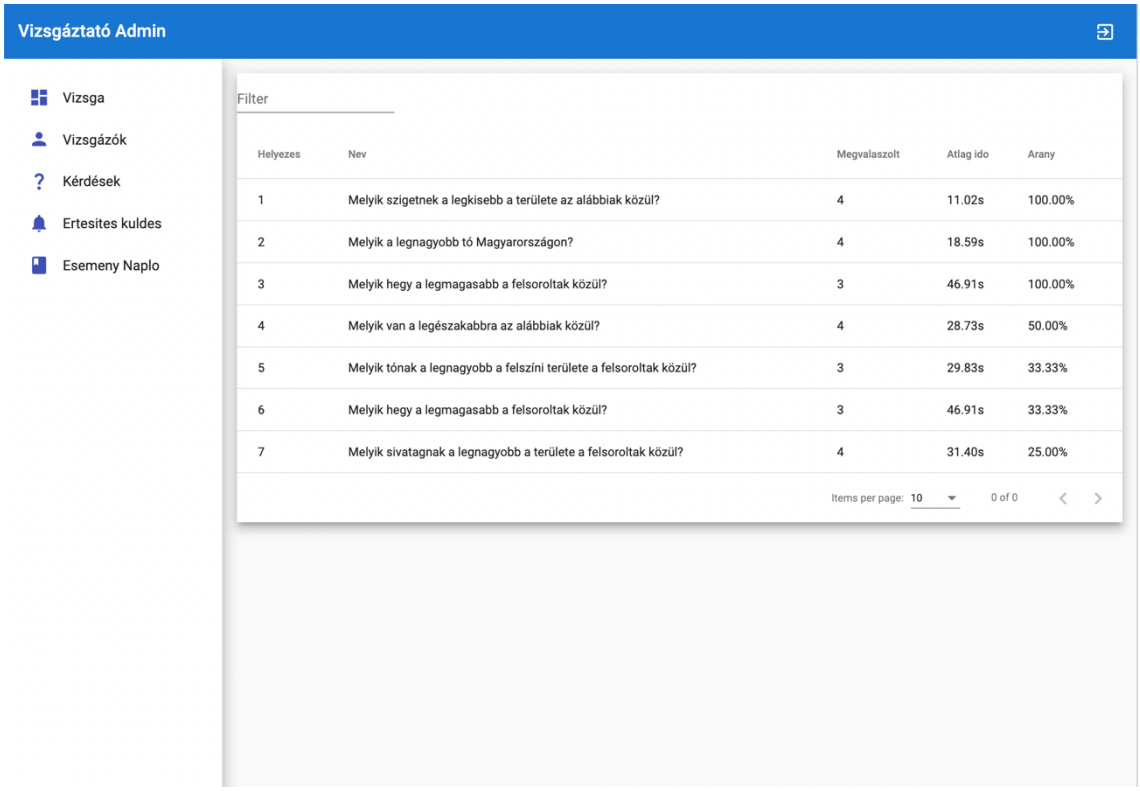


ábra 8 Vizsgázók táblázat

8.2.6. Kérdések nyomon követése

A kérdések nyomon követése úgyszintén az Angular-Material beépített táblázatát használom, hasonlóan, mint a diákok felügyelésénél. Ez a táblázat is lapozható, és ugyanúgy tartalmaz egy szűrőt, ami segítségével az egyes kérdésekre rákereshetünk.
A táblázat minden egyes sora egy vizsga kérdést tartalmaz. A kérdés mellett a szerepel a kérdésre adott válaszok száma, a kérdés megválaszolással átlagosan eltöltött idő, valamint a válaszok helyességének aránya százalékos lebontásban. A táblázat a válaszok helyessége szerint van rendezve, ezáltal a vizsgáztató könnyen látja melyek azok a

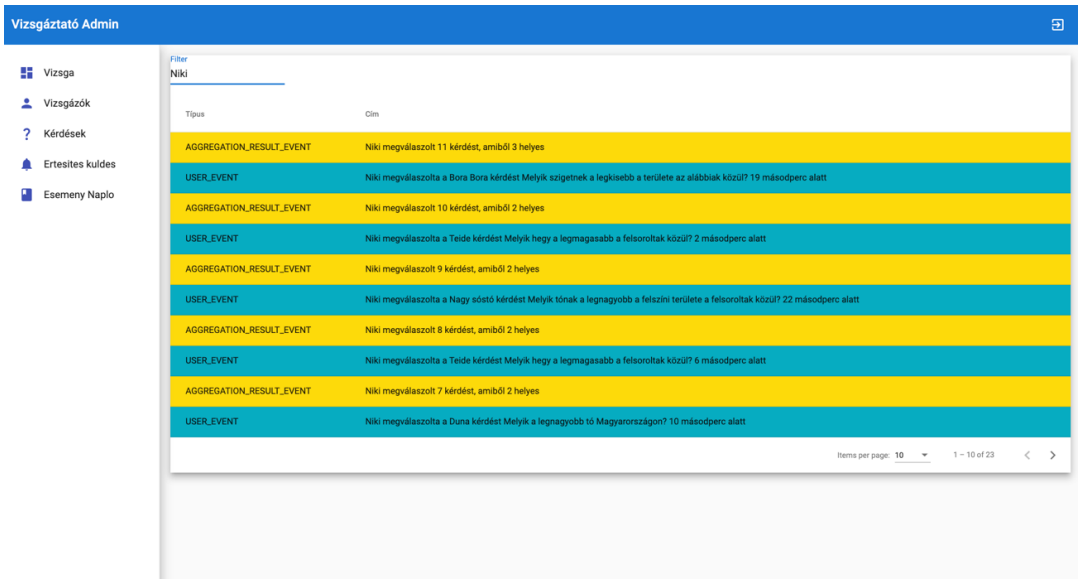
kérdések, amiket a vizsgázók könnyebben megválaszoltak, és mely kérdések okoztak több fejtörést.



ábra 9 Kérdések táblázat

8.2.7. Eseménynapló

Az eseménynapló egy segéd képernyő, ami összesítve tartalmazza az eseményeket egy bizonyos felületen. Ez segít a vizsgáztatónak egy összesített képet kapnia a vizsga aktuális állapotáról. Hasonlóan az előző két felülethez ez is ugyanazt az *Angular Material* lapozható táblázatkomponensét használja. Az alábbi ábrán láthatóak az eseménynapló “Niki”-re szűrt eseményei.



ábra 10 Eseménynapló táblázat

8.2.8. Idő meghosszabbítása

A hátralévő időt a vizsga komponens segítségével a tanár módosíthatja. Egy legördülő menü különböző értékeket ajánl fel (5, 10, 15, 20, 30 perc). A vizsga meghosszabbítása gomb megnyomásával a böngésző egy REST parancsot küld a szerver oldali kontroller komponens felé, amit a kontroller a *userEvents* Kafka topic-ba továbbít.
Ezt a parancsot a vizsgázó alkalmazás szervere valós időben fogadja, és módosítja az időt a szerver oldalon, és végső sorban WebSocket-en keresztül értesíti a vizsgázók böngészőjében futó programot.

8.2.9. Új kérdés hozzáadása

A vizsga közben a tanár bármikor hozzáadhat új kérdéseket a vizsgasorhoz. Az új kérdés gomb megnyomására egy dialógus komponens nyílik meg. A kérdés szövege, lehetséges válaszok, illetve a helyes válasz megjelölése után a mentés gombbal hozható létre az új kérdés, ami egy REST parancsot küld az alkalmazás szervernek.
Hasonlóan, mint az idő meghosszabbításánál, ez a parancs is a *userEvents Kafka topic-ba* továbbítja az üzenetet, amit a felhasználó alkalmazás szervere feldolgoz, és WebSocket-en keresztül értesíti az összes vizsgázót. Végül az új kérdés automatikusan megjelenik a diák vizsgalapján.

8.2.10. Értesítés küldés a felhasználó felé

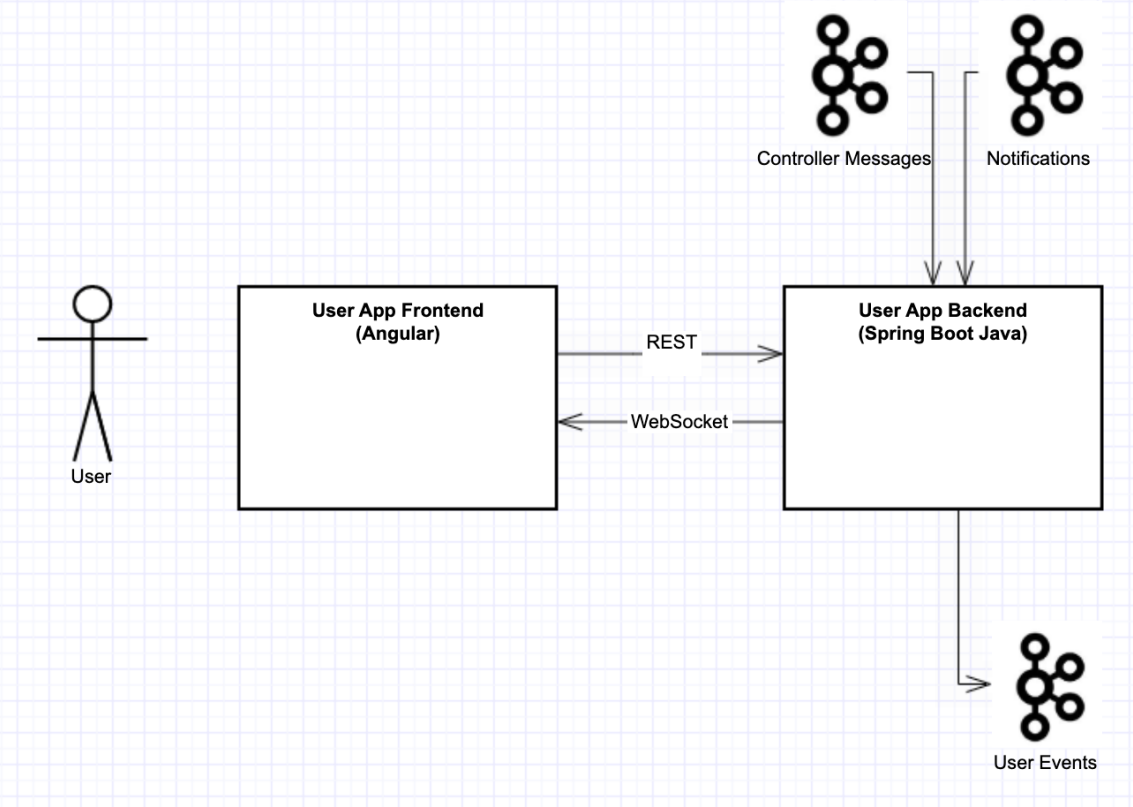
A tanárnak lehetősége van szöveges értesítéseket küldeni a vizsgázók felé. Az értesítések komponensen láthatóak az eddig küldött értesítések és a szövegmező kitöltésével lehet új értesítést küldeni a vizsgázóknak.
Az értesítést egy REST híváson küldi a szervernek, ami azt továbbítja a *Notifications Kafka topic-nak*. Végül az értesítések WebSocket-en keresztül jutnak el a vizsgázókhoz.

9. FELHASZNÁLÓ ALKALMAZÁS

A felhasználó alkalmazáson keresztül kommunikálnak a rendszerrel a vizsgázók, azaz vizsgáznak.

9.1. **Felépítése**

Felépítését tekintve két komponensből áll össze, *User App* webalkalmazásból, illetve a *User App Backend* szerveroldali komponensből, hasonlóan, mint a kontroller alkalmazás. Az alábbi ábra szemlélteti a felépítést: A szerveroldali komponens a kontroller parancsaira, illetve értesítéseire feliratkozik, és WebSocket-en keresztül továbbítja őket a kliensalkalmazásnak. A kliensalkalmazás REST hívásokon keresztül küldi a válaszokat a szervernek, amikből a szerver oldali komponens eseményeket hoz létre és elküldi a *UserEvents* Kafka topic-ba.



ábra 11 Felhasználó alkalmazás felépítése

9.2. **Funkcionalitása**

9.2.1. Bejelentkezés

A bejelentkező képernyő a diák nevét, illetve a csoport számát kéri. A csoportszám azonosítja a vizsgázást. Egyidőben több csoport is vizsgázhat a rendszeren keresztül, és a vizsgák egymástól eltérőek lehetnek.

9.2.2. Vizsgázás elindítása

A bejelentkezést követően a diák addig nem látja a kérdéseket amíg a tanár el nem indít egy vizsgát. A háttérben ez úgy valósul meg, hogy a kontroller alkalmazás egy parancs eseményt hoz létre (vizsga elindítása), amit a felhasználó szerver oldali komponense fogad és feldolgoz. Ez azt jelenti a gyakorlatban, hogy eltárolja a kérdéssort és elküldi WebSocket kapcsolaton keresztül a csoport bejelentkezett kliens webalkalmazásainak.
Ezt követően a vizsgázó Angular alkalmazás betölti a vizsgakérdéseket, és a felhasználónak megjeleníti az első kérdést. Továbbá a felületen a hátra levő idő is megjelenik.

9.2.3. Vizsgázás folyamata

A vizsgázás feleletválasztós kérdések megválaszolásával történik egy megadott időkereten belül, amit a fejlécben található visszaszámláló jelenít meg. Az idő lejárta után a vizsgázó nem tud több választ küldeni. A szerver oldali komponens validációja ezt megakadályozza.
Az alkalmazás egyszerre egy kérdést jelenít meg. A kérdés alatt szerepelnek a lehetséges válaszok, amelyek közül a diák rádiógomb megnyomásával választhat egyet, amelyet az idő lejártáig módosíthat.

ábra 12 Vizsgakérdés felület

9.2.4. Kérdések közötti navigáció

A diák a kérdések között gombok megnyomásával tud navigálni (következő, illetve előző kérdés). A diák tetszőleges sorrendben, az idő lejártáig tudja megválaszolni a kérdéseket.

9.2.5. Idő meghosszabbítása

Szüksége esetén a rendelkezésre álló időt a tanár az idő lejártáig meghosszabbíthatja.

9.2.6. Értesítések fogadása

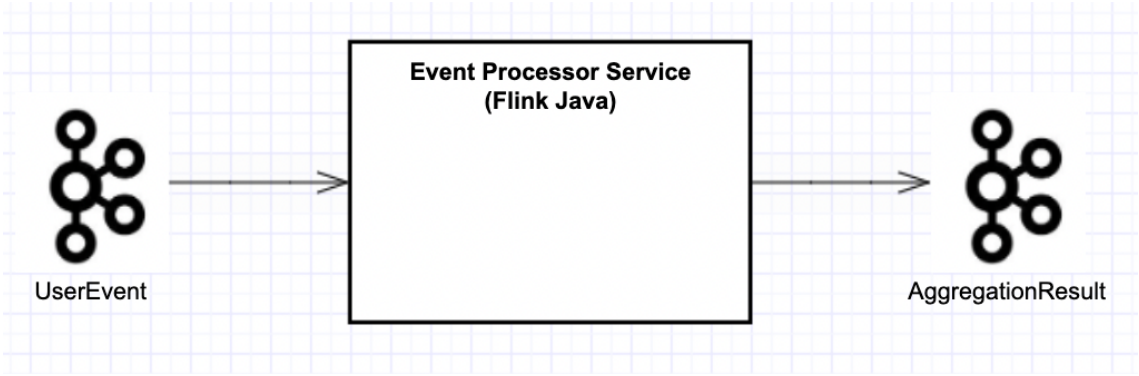
A felhasználók értesítéseket fogadnak a tanártól. Célja, hogy a tanár közbe tudjon lépni, kommunikálni tudjon a csoporttal. Például, ha a tanár észreveszi, hogy bizonyos kérdésekkel több időt töltenek a felhasználók, mint indokolt lenne, vagy jellemzően helytelen válaszokat adnak, akkor tud küldeni egy értesítést, ami azonnal megjelenik minden felhasználó képernyőjén. A tanár által küldött üzenetek pop-up üzenet formájában megjelennek a felhasználó képernyőjén.

10. ESEMÉNYFELDOLGOZÁS

Azt, hogy a kontroller fél a vizsga menetéről valós idejű statisztikát kapjon, és aggregált formában lásson a vizsgázásról adatokat az eseményfeldolgozó modul teszi lehetővé.
Az eseményfeldolgozó egy láthatatlan szerveroldali komponens, ami a program motorjaként szolgál. Ez egy háttérben futó alkalmazás, ami a felhasználó által létrehozott eseményeket aszinkron módon dolgozza fel.
Bemutatom az architektúrát, majd részletezem az eseményfeldolgozó komponens és a Flink működését kitérve az implementáció sajátosságaira.

10.1.1. Felépítése

Az architektúra szempontjából egy rejtett komponensről van szó, ami annyit jelent, hogy a rendszer többi komponense nincs vele közvetlen kapcsolatban. Ez nagy rugalmasságot biztosít. Erre jó példa lehet, hogy a komponens könnyedén lecserélhető egy más eseményfeldolgozó komponenssel, vagy akár több eseményfeldolgozó rendszer is futhat párhuzamosan, anélkül, hogy az egyes komponensek tudnának egymás létezéséről.
A komponens kizárólag a Kafka üzeneteken keresztül kommunikál a rendszerrel, amit az alábbi ábra szemléltet.



ábra 13 Eseményfeldolgozás architektúra

A komponens bemenetele a felhasználó események, a kimenete pedig az aggregált események, amik a kontroller alkalmazásnak szolgálnak majd bemenetként.
Az adatok feldolgozását a Flink szoftver segítségével végeztem, amit a következő fejezeben szemléltetek.

10.1.2. Eseményfeldolgozás a Flink segítségével

Az implementációt nagyban elősegítette a nyílt forráskódú Apache Flink eseményfeldolgozó szoftver használata. A Flink egy keretrendszert biztosít, ami segít az állapotkezelésben, adatok csoportosításában, az ismételten küldött üzenetek kiszűrésében.

A telepített Flink szervernek feladatokat tudunk adni (job), amiket .jar kiterjesztésű fájlba csomagolva tud értelmezni. Ahhoz, hogy a komponens feladatként tudjon szolgálni a Flink szerver számára, a *flink-java flink-connector-base* és a *flink-connector-kafka* függőségekre van szüksége a komponensnek.

A program belépési pontja a *main* függvény, aminek az implementációját az ábra mutatja.

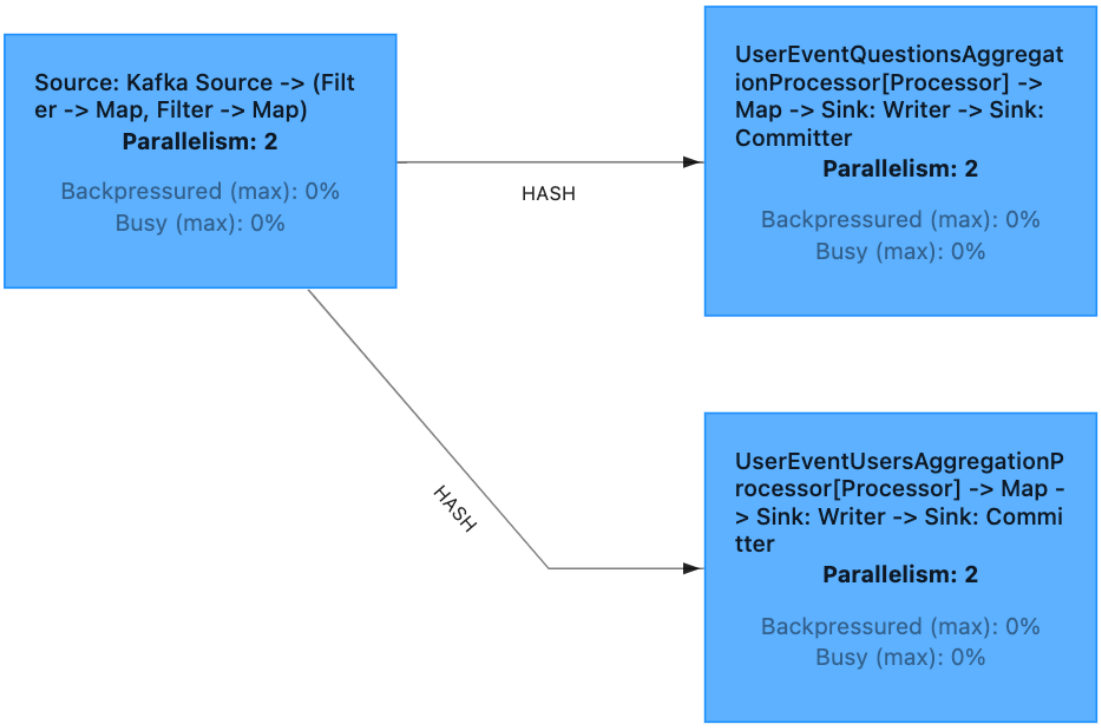
```
Juraj *
public static void main(String[] args) {
    log.info("Flink Application starting... ");
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    var userEventSource : KafkaSource<UserEvent> = userEventSource();

    var userEventDataStreamSource : DataStreamSource<UserEvent> = env.fromSource(userEventSource,
            WatermarkStrategy.noWatermarks(), sourceName: "Kafka Source");

    userEventDataStreamSource
            .filter(App::isActionSubmitQuestion) SingleOutputStreamOperator<UserEvent>
            .map(UserEvent::getAnswerSubmittedUserEvent) SingleOutputStreamOperator<AnswerSubmittedUserEvent>
            .keyBy(event -> getKeyUsingSessionAndSecondaryKey(event.getSessionId(), event.getQuestion().getId())) KeyedStream<AnswerSubmittedUserEvent, String>
            .process(new UserEventQuestionsAggregationProcessor()) SingleOutputStreamOperator<QuestionDetailsAggregation>
            .name("UserEventQuestionsAggregationProcessor[Processor]")
            .map(App::fromQuestionDetailsAggregation) SingleOutputStreamOperator<AggregationResultEvent>
            .sinkTo(aggregationResultsSink());

    userEventDataStreamSource
            .filter(App::isActionSubmitQuestion) SingleOutputStreamOperator<UserEvent>
            .map(UserEvent::getAnswerSubmittedUserEvent) SingleOutputStreamOperator<AnswerSubmittedUserEvent>
            .keyBy(e -> getKeyUsingSessionAndSecondaryKey(e.getSessionId(), e.getUserId())) KeyedStream<AnswerSubmittedUserEvent, String>
            .process(new UserEventUsersAggregationProcessor()) SingleOutputStreamOperator<UserAggregation>
            .name("UserEventUsersAggregationProcessor[Processor]")
            .map(App::fromUserEventAggregation) SingleOutputStreamOperator<AggregationResultEvent>
            .sinkTo(aggregationResultsSink());

    log.info("Flink Application started successfully!");
}
```

ábra 14 Adatfolyam feldolgozás forráskód

Maga a Flink adatfolyamokkal dolgozik. Az ábrán látható implementáció létrehozza az adatcsatornákat, amik az üzenetek által aszinkron módon aktiválódnak.

Az esemény feldolgozás három részből tevődik össze. Első körben létrehoztam egy *userEventDataSource* forrást, ami szerializálja a bemenetként szolgáló felhasználó eseményeket. Ezeket az üzeneteket binárisan kódolva kapja a komponens a Kafka topic-ból. Ez a lépés létrehozza az adatfolyamot, szükséges, ahhoz, hogy a Flink transzformációkat tudjon végrehajtani a bejövő eseményeken.

Ezt követően a *userEventDataSource* forrásból két transzformációs csatornát hozok létre, amik egymástól függetlenek. Mindkét csatorna megkapja az összes felhasználók által generált eseményt a bemenő forrásból, amint az üzenetek elérhetőek. Az alábbi ábra szemlélteti az említett három komponensből kiépített adatcsatornát. Jól látható, hogy első lépésben a beérkező Kafka eseményeket dolgozza fel, ami a két másik transzformációs komponens számára bemenetként szolgál. Továbbá észrevehető, hogy a két feldolgozó folyamat egymástól függetlenül párhuzamosan tud működni, ez egy gyorsabb végrehajtást eredményez



ábra 15 Flink feldolgozó komponensek

10.1.3. Üzenetek szűrése

A bemenő eseményfolyam tartalmazza az összes felhasználó által generált eseményt, mint például, a felhasználó bejelentkezett, vagy hogy beküldte a vizsgát.
A feladatomban kizárólag a vizsgakérdésekre adott válaszokat célom feldolgozni. Ahhoz, hogy kiszűrjem ezeket az eseményeket, létrehoztam egy *isActionSubmitQuestion* szűrőt. A rendszer tervezésnél úgy állítottam össze az üzenetek adatsémáját, hogy minden üzenet tartalmazza *enum* formában annak a típusát, ezáltal a rendszer könnyen ki tudja szűrni a feldolgozáshoz szükséges üzeneteket.

10.1.4. Megválaszolt kérdések összegzése

A kérdésekre adott válaszok összegzéséért létrehoztam a *UserEventQuestionsAggregationProcessor* feldolgozó csatornát, ami ezeket csoportosítja. Minden beérkezett kérdésből kalkulál egy eseményt, a következő mezőkel:

1. kérdésre adott válaszok számát az egyes válaszok eloszlásával abszolút érték, illetve, százalékos lebontásban
2. kérdés megválaszolásának átlag idejét
3. kérdés sikeres megválaszolását abszolút érték és százalékos formában

10.1.5. Felhasználók válaszainak összegzése

Ahhoz, hogy a válaszokat a felhasználók szerint csoportosítsam létrehoztam egy csatornát *UserEventUsersAggregationProcessor* néven. Az előző csoportosításhoz hasonló értékeket számít:

1. hány darab kérdést válaszolt meg a felhasználó
2. átlagosan mennyi időt töltött a kérdésekkel
3. helyesen megválaszolt kérdések száma, illetve azok aránya százalékos lebontásban

10.1.6. Kulcs

A kérdések, és a felhasználók csoportosításához szükségszerű az előző számításokat tárolni. Szerencsére a Flink-nek van beépített támogatása az állapotkezelésre. A Flink kulcs-érték párként kezeli az állapotot, ahol a kulcs szerint csoportosítjuk az üzeneteket.

A programban mindkét feldolgozó adatcsatornának megadtunk egy-egy kulcsot. Ami a kérdések csoportosításánál egy összetett kulcs, a kérdés azonosítója és vizsgamenet összegzéséből tevődik össze. A felhasználókra levő csoportosításnál pedig a felhasználó azonosítója és szintén a vizsgamenet azonosítójának összegzéséből áll össze.

10.1.7. Állapotkezelés

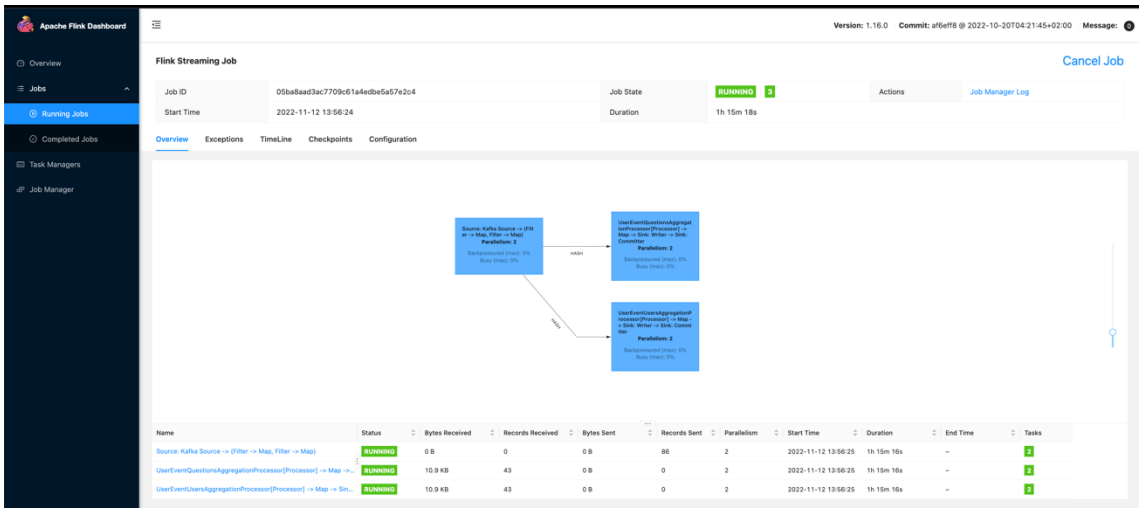
A Flink jellemzően több szerveren egy klaszter formában van telepítve. A Flink elsősorban a szerver memóriájában tárolja az állapotot és rendszeres időközönként az állapot egy biztonsági mentés keretein belül egy központi helyre kerül. A biztonsági mentések tárhelye konfigurálható, ugyanakkor a Flink jellemzően egy előre definiált AWS S3 mappát használ tárolásra.

Az állapot memóriában tárolása egy gyorsabb feldolgozást eredményez. Ha az állapot kizárólag központi helyen lenne tárolva, mondjuk egy Redis klaszterben, az szükségessé tenné az állapot lekérését minden egyes üzenetfeldolgozásnál. Ez a megoldás jóval több hálózati forgalmat eredményezne, valamint lassabb feldolgozást.

Szükséges, hogy a csoportosított üzenetek minden alkalommal ugyanarra a szerverre kerüljenek. A kulcs feladata ezt garantálni.
A szakdolgozatomban a Flink mindössze egy darab példányban fut. A Flink klaszterkezelése túlmutat a szakdolgozatom témáján, ugyanakkor fontosnak tartottam megemlíteni a Flink működésének szemléltetése végett.

10.1.8. Flink adminisztrációs felület

A Flink egy adminisztrációs webes felületet biztosít, ahol a telepített feladatokat felügyelhetjük és az adatfolyam áramlást valós időben nyomon követhetjük a Flink beépített, illetve az általunk létrehozott metrikákkal. Továbbá megtekinthetjük a konfigurációkat és az egyes feladatokhoz tartozó naplófájlokat.



ábra 16 Flink adminisztrátor felület

11. TELEPÍTÉS ÉS FUTTATÁS

Ebben a fejezetben ismertetem az alkalmazás fordítását, illetve telepítését. Egy elosztott rendszer lévén az alkalmazás több kisseb programból és azok interakciójából áll össze, tehát ez egy összetettebb telepítési stratégiát kíván meg. Napjainkban az elosztott rendszereket jellemzően Docker konténerek segítségével futtatjuk.

11.1. Fordítás

Első lépésben, még az egyes alkalmazás komponensek fordítása előtt, a forráskódban levő *common-service* modult szükséges lefordítani, aminek fordítási eredménye egy *jar* kiterjesztésű programkönyvtár. Mivel a *common-service* Java programozási nyelven íródott, így a fordításhoz az *Apache Maven* programot használtam.
A fordítás az alábbi parancs lefuttatásával történik, ami a lefordított könyvtárat a *Maven* lokális adattárába menti.

mvn clean install

A programkönyvtár mint függőségként szolgál a többi szerveroldali komponens számára. Így a *common-service*, lefordítása után már le tudjuk fordítani a rendszer java alapú komponenseit és a komponensek lefordítását követően egy futtatható *.jar* kiterjesztésű fájlokat kapunk.

11.2. Konténerek használata

A platformfüggetlenség elvét követve konténerek formájában történik az alkalmazás futtatása. A konténerek használata egy izolált transzparens környezetet biztosít, ami lehetővé teszi, hogy az alkalmazás ugyanúgy fusson különböző operációs rendszeren és változó konfigurációjú szervereken. A konténerek használata kiküszöböli az eltérő operációs rendszer konfigurációjából adódó hibákat. A konténerek használata előtt gyakran előfordult, hogy az alkalmazás működött a fejlesztő gépén, az éles rendszerben pedig hibára futott. Ez több szempontból súlyos probléma. Először is ezeket a hibákat sokkal nehezebb felismerni, másodszor pedig nehezebb reprodukálni és ezáltal kijavítani. Például a szerver eltérő verziójú java virtuális gépet vagy eltérő karakterkódolást használ.

11.3. Képfájlok létrehozása

A konténerek elindítása a képfájlok futtatásával történik, ezért a komponensek lefordítása után létrehozom a Docker virtuális környezetében futtatható képfájlokat.
A képfájlokat leginkább mint egy virtuális gép állapotmentéseként lehet elképzelni.
A rendszer minden egyes komponense tartalmaz egy szöveges Dockerfile-t, amiből a Docker segítségével futtatható képfájlokat készítek. Példaként az alábbi parancs által hozom létre a kontroller szerver oldali komponens képfájlját:

docker build --tag=dashboard:latest .

Az alábbi példában pedig a bemenetként szolgáló Dockerfile-a található:

```
FROM arm64v8/amazoncorretto:11-al2-jdk
MAINTAINER jurajkucera
COPY target/dashboard-0.0.1-SNAPSHOT.jar dashboard-0.0.1.jar
COPY src/main/resources/static/exams /app/exams
ENTRYPOINT ["java","-jar","/dashboard-0.0.1.jar"]
```

A képfájlok tulajdonsága az öröklődés. Minden egyes képfájl kiterjeszthető, másszóval egy új képfájl alapjául szolgálhat. A példánkban levő Dockerfile az amazoncorretto:11-al2-jdk képfájlt használja alapul, ami tartalmazza a JDK 11-nek az Amazon által létrehozott Corretto nevű nyíltforrású implementációját. A JDK 11 szükséges ahhoz, hogy a Maven által létrehozott futtatható *.jar* kiterjesztésű fájlon keresztül elindítsuk a kontroller szerver oldali dashboard nevű komponensét.
A Dockerfile első COPY parancsa átmásolja a lefordított *.jar* fájlt a docker virtuális konténerébe, a második COPY parancs pedig a vizsga sablonokat másolja át.
A gyakorlatban az elkészült képfájlokat egy központi tárhelyen, egy Docker registry-ben tároljuk. Ezáltal nem szükséges ugyanazon a szerveren létrehozni a képfájlt mint ahol a konténert futtatjuk. Továbbá így egyszerűen kiterjeszthetünk és futtathatunk mások által létrehozott tetszőleges képfájlokat.

11.4. **Konténerek felépítése**

Minden egyes komponens, az egyes képfájlok elindítása után, külön konténerben fut. A Docker működéséből adódóan az egyes konténerek egymástól elkülönített virtuális környezetben futnak. A konténerek rétegesen épülnek fel, és a virtualizáció során, a Docker engine felhasználja a konténerek közös rétegeit. Így az egyes konténerek futtatása kevesebb memóriát igény, mintha külön-külön virtuális gépeken futnának a komponensek.
A példaprogramban ez annyit jelent, hogy a mind a három szerver oldali komponens ugyanazt a képfájlt használja a konténer alapjául. Ezáltal a mindössze egy Linux alapú virtuális operációs rendszert hoz létre a Docker Engine, amire egyszer telepíti a JDK-11-et. Ezt a virtuális operációs rendszert a Docker újra felhasználja a szerver oldali konténerek futtatásánál.

11.5. **Konténerekben futtatása**

A képfájlok létrehozása után a program komponensei konténerekben futtathatók.
Az architektúra elosztott jellege lévén az egyes komponenseket egyidőben szükséges futtatni, ahhoz, hogy az alkalmazás helyesen működjön.

A Docker eszköztár *Docker Compose* alkalmazásával egy több konténerből álló alkalmazást hozhatunk létre, amit lefuttatva a Docker elindítja a megfelelő sorrendben az összes komponens konténerét.
A konténerek beállításait és a konténerek közötti függőségeket egy *docker-compose.yml* fájl tartalmazza. A példaprogram elindításához használt fájl kilenc konténert indít el. Az alábbi példában látható a használt **docker-compose.yml** fájlból kiragadott kódrészletek (nem teljes):

```
dashboard-server:
    depends_on:
    - init-kafka
    container_name: dashboard
    build:
        context: dashboard
        dockerfile: Dockerfile
    image: dashboard:latest
    ports:
    - 8181:80
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8181/exam"]
      interval: 30s
      timeout: 10s
      retries: 5

exam-server:
    depends_on:
    - init-kafka
    container_name: exam
  build:
      context: exam
      dockerfile: Dockerfile
  image: exam:latest
  ports:
  - 8182:8182

controller-ui-app:
  depends_on:
    - dashboard-server
  container_name: controller_ui_app
  build:
    context: controller-ui-app
    dockerfile: Dockerfile
  image: controller_ui_app:latest
  ports:
  - 4201:80

jobmanager:
  image: flink:latest
```

```
...
taskmanager:
  image: flink:latest
...
```

A konténerek futtatását leíró fájlt a következő paranccsal indítom el:
docker-compose down && docker-compose up --build

Amint említettem ez a parancs összesen az kilenc konténert hoz létre a következő sorrendben:

1. Apache Zookeeper konténer: az Apache Kafka üzemeltetéséhez szükséges
2. Apache Kafka konténer
3. Init-Kafka konténer: létre hozza az alkalmazásban használt kafka topic-okat
4. Felhasználó szerver oldali alkalmazás konténere
5. Konténer szerver oldali alkalmazás konténere
6. *Apache Flink jobmanager* konténer: maga a Flink környezet
7. *Apache Flink taskmanager* konténer: az *event-processor* jar fájlt futtató Flink konténer
8. Felhasználó kliens oldali alkalmazás konténere
9. Kontroller kliens oldali alkalmazás konténere

12. TESZTELÉS

A tesztelés a modern szoftverfejlesztés szerves része. Általánosságban elmondható, hogy minél később veszünk észre egy hibát egy programban, annál költségesebb és annál több időt vesz igénybe a kijavítása. Ezért a program fejlesztésének kezdete óta folyamatosan különböző típusú automatizált tesztekkel vizsgálom a rendszer működését. Ebben a fejezeben az általam használt tesztelési módszereket mutatom be.

12.1. Egységtesztelés

Az egységtesztelés a nevéből is adódóan a program legkisebb egységének (osztályok, függvények) teszteléséről szól. Az egységtesztelést legtöbbször a fejlesztők végzik.
Ezeket komponenseket a *JUnit* könyvtár segítségével teszteltem. A tesztelés ennél a formájánál nincs telepítve az alkalmazás. Az egységtesztelés legnagyobb előnye, hogy azonnal megbizonyosodhatunk arról, hogy az adott egység elvárt módon viselkedik. Hátránya, hogy a komponensek közötti kommunikáció tesztelésére nem alkalmas.
Az alábbi ábra a *CookieParser* osztály *extractField* statikus metódusát teszteli. Két tesztesetet írtam, egy pozitív teszteset, amikor a süti tartalmazza mezőt, illetve egy negatív tesztesetet amiker a keresett mező nincs a sütiben.

```
public class CookieParserTest {

    Juraj *
    @Test
    public void extractFieldShouldReturnTheFieldsValue() {
        var cookie = "deviceId=57235430-2B9E-42EE-A90E-295818B42CF3; groupId=historyClass";
        assertEquals(CookieParser.extractField(cookie,  key: "groupId"),  actual: "historyClass");
    }

    Juraj *
    @Test
    public void extractMissingFieldShouldReturnEmptyString() {
        var cookie = "Idea-37ad5b0d=70111830-ea42-4d49-8175-a470e7d7cf5e; deviceId=57235430-2B9E-42EE-A90E-295818B42CF3;";
        assertEquals(CookieParser.extractField(cookie,  key: "groupId"),  actual: "");
    }
}
```

ábra 17 Példa egységtesztre

12.2. Feltelepített rendszer tesztelése

A rendszer elosztott jellege lévén külön fontosnak tartom a feltelepített rendszert tesztelni. Erre a *feketedobozos tesztelési módszert* alkalmaztam, amely módszernek a lényege, hogy a teszt kizárólag a rendszer nyilvános felületein (interfészein) keresztül kommunikál az alkalmazással, és a rendszer belső működését figyelmen kívül hagyja. [23]

Gyakorlatban ez úgy néz ki, hogy létrehoztam egy külön Java Spring alapú modult a feltelepített rendszer tesztelésére, ami segítségével automatizált teszteket futtatok le a rendszer egyes funkcióinak vizsgálatára.

A modul felhasználja a *common-service* komponenst, ami egy *könyvtár (maven)* függőségként van beimportálva a tesztmodulba, így a modell POJO osztályok (Plain Old Java Object) és a közös függőségek könnyedén felhasználhatók.

12.2.1. Viselkedés tesztelése

A tesztelésnél a viselkedés vezérelt fejlesztési módszer (BDD) irányelveit követtem. [24] A módszer használatával a teszteket egy meghatározott sablon alapján, mint narratívaként definiáljuk. Az egyes tesztek rendszerint az *Adott (Given) – Mikor (When) – Ezután (Then)* sablon alapján épülnek fel a következőképpen:

- **Adott -** a vizsga el van indítva
- **Mikor** - a kontroller létrehoz egy új kérdést
- **Ezután -** a vizsgázó WebSocket-en keresztül megkapja a kérdést

Alapesetben a módszer egy tartományspecifikus nyelven írja le magát a teszteseteket. A szakdolgozatomban az egyszerűsítés kedvéért a narratívát kommentek segítségével írtam le. A következő kódrészlet az említett vizsgakérdés létrehozást teszteli.

```
@Test
public void createNewQuestion() throws InterruptedException {
    // GIVEN the exam has been started
    var examResponseResponseEntity :ResponseEntity<ExamResponse> = startExam(StartExamDTO.builder().groupId("group1").examId("test-exam-1").build());
    var webSocketClient :SimpleWebSocketTestClient = createWebsocketClientForGroup1AndWaitsForTheFirstMessage(userWebSocketUrl);
    webSocketClient.reset( messageCount: 1);

    // WHEN the controller adds a new question
    testRestTemplate.postForEntity( url: controllerHost + "/exam/question", Question.builder()
            .id(UUID.randomUUID().toString())
            .examId("test-exam-1")
            .sessionId(examResponseResponseEntity.getBody().getExamSessionId())
            .title("Population of Hungary")
            .answers(List.of(
                    answer( id: "ans-1",  title: "5_000_000",  correct: false),
                    answer( id: "ans-2",  title: "10_000_000",  correct: true),
                    answer( id: "ans-3",  title: "15_000_000",  correct: false)))
            .build(), String.class);

    // THEN the user receives the Question over the WebSocket connection
    webSocketClient.waitToReceiveMessages( seconds: 5);
    assertThat(webSocketClient.getReceivedMessages()).singleElement().satisfies(m -> {
        var event :UserWebSocketEvent = OM.readValue(m, UserWebSocketEvent.class);
        assertThat(event.getAction()).isEqualTo(UserWebSocketAction.COMMAND_EVENT);
        var commandEvent :CommandEvent = event.getCommandEvent();
        assertThat(commandEvent.getAction()).isEqualTo(CommandAction.NEW_QUESTION_ADDED);
        assertThat(commandEvent.getQuestion()).satisfies(q -> {
            assertThat(q.getTitle()).isEqualTo( expected: "Population of Hungary");
            assertThat(q.getAnswers()).hasSize( expected: 3)
                    .allSatisfy(a -> assertNull(a.getCorrect())) // not sending which is correct to the client
                    .anySatisfy(a -> assertThat(a.getTitle()).isEqualTo( expected: "5_000_000"))
                    .anySatisfy(a -> assertThat(a.getTitle()).isEqualTo( expected: "10_000_000"))
                    .anySatisfy(a -> assertThat(a.getTitle()).isEqualTo( expected: "15_000_000"));
        });
    });
    webSocketClient.closeBlocking();
}
```

ábra 18 Példa integrációs tesztre

12.2.2. Interakció a rendszerrel

Az alkalmazással REST hívásokon keresztül lehet interakcióba lépni, ezért minden teszt egy vagy több REST hívásból fog állni. Rendszernél két típusú REST API-t különböztetünk meg. Az egyik típus a lekérdezések – a GET hívások- amiken keresztül adatokat (vizsgákat, felhasználókat) tudunk lekérni a rendszerből, a másik típus pedig a parancsok. Ezek a parancsok jellemzően egy eseményt generálnak, amire a rendszer többi komponense reagál és végső soron WebSocket-en keresztül vannak értesítve a kliensek.

12.2.3. Ellenőrzés az AssertJ könyvárral

Az eredmények ellenőrzéséhez az *AssertJ* könyvtárat használtam a tesztelés során.
A könyvtár nyújtotta statikus függvények által egyszerű és tisztán olvasható ellenőrzéseket írtam.
A GET hívások tesztelése egyszerűen szinkron módon történik. A kérésre visszaadott HTTP válasz fejlécét, illetve tartalmát ellenőrizzük, amit a fenti ábrából is jól látható.
Az egyes parancsok tesztelése kissé bonyolultabb volt. Itt a REST parancs küldést követően, jellemzően egy másik komponens WebSocket csatornáján érkező adatait vizsgáljuk. A fenti példa az új kérdés hozzáadását teszteli, ahol a kontroller létrehoz egy kérdést, amit a csoport felhasználói a WebSocket-csatornán keresztül megkapnak.

12.2.4. SimpleWebSocketTestClient osztály

A WebSocket-en keresztül érkező üzenetek tesztelésére létrehoztam a *SimpleWebSocketTestClient* osztályt, ami a *WebSocketClient* ősosztályból öröklődik. Az osztály a fogadott üzeneteket egy listába gyűjti, amik a *getReceivedMessages()* metódussal lekérhetők. Továbbá a WebSocket kommunikáció aszinkron jellegéből adódóan a teszteléshez létrehoztam a *waitToReceiveMessages(seconds)* függvényt, ami várakozik amíg az üzenetek megérkeznek vagy pedig várakozik az átadott maximális időtartamig. A szinkronizáló a *CountDownLatch* osztály segítségével történik. Ezeknek a metódusoknak köszönhetően az aszinkron üzenetfogadást szinkron módon tudtam tesztelni.

12.2.5. Automatikus tesztelés

A rendszer automatikus tesztelése nagy segítséget nyújt a fejlesztés során. A tesztesetek lefuttatása által folyamatosan visszajelzést kapunk a rendszer működéséről.

Automatikus tesztek nélkül a kódrefaktorálás egy veszélyes tevékenység, mivel nem tudhatjuk, hogy a változások miként befolyásolták a rendszer működését.
Ezért egy összetettebb rendszernél a tesztek írása és a feltelepített rendszer automatikus tesztelése elengedhetetlen.

13. SZIMULÁCIÓ ÉS ÉRTÉKELÉS

Ebben a fejezetben a feltelepített alkalmazás kipróbálásáról és teszteléséről osztom meg a tapasztalataimat. A program fő funkciója a vizsgázás lebonyolítása, amit jellemzően több vizsgázó és egy tanár teljesít. Az, hogy az alkalmazást lokális környezetben telepítettem fel, ez kissé megnehezítette a program többszereplős használatát.

13.1. **Szimuláció**

A többszereplős vizsgázás lebonyolításához egy olyan hibrid tesztelési módszert alkalmaztam, ami manuális tesztelésből és szimulációk lefuttatásából áll össze. Két böngésző ablakon keresztül bejelentkeztem egy kontrollerrel és egy vizsgázóval. A többi vizsgázó bejelentkezéséhez és a kérdéseik megválaszolásához szimulációt alkalmaztam. Ez lehetővé tette, hogy kipróbáljam az alkalmazást egy többszereplős környezetben. A szimulációhoz létrehoztam a *test-event-generator* modulon belül a *Simulations* osztályt. Az egyes szimulációkon belül a kommunikáció REST hívásokon keresztül történik.

13.2. **Bejelentkezés**

Először a Chrome böngésző segítségével bejelentkeztem a kontroller alkalmazásba. Ezt követően egy másik böngésző ablakban bejelentkeztem a vizsgázó alkalmazásba egy "Aladár" nevű vizsgázóval. A bejelentkezés után a kontroller felületén a vizsgázó menüpont alatt automatikusan megjelent az Aladár nevű vizsgázó. Ezután további 15 felhasználó bejelentkezését imitáltam a *loginWith15Users()* szimuláció lefuttatásával, ami félmásodpercenként bejelentkezett a következő vizsgázóval. Minden egyes bejelentkezés után valós időben befrissült a kontroller Vizsgázók táblázata az új felhasználóval. Továbbá az eseménynaplóban is megjelentek a bejelentkezés események.

13.3. **Vizsga indítása**

Miután az összes vizsgázó bejelentkezett, a kontrollerrel kiválasztottam a földrajz vizsgát. A kiválasztást követően az „Aladár" vizsgázó képernyőjén elkezdődött a földrajz vizsga, megjelenítve az első vizsgakérdést és a hátra levő időt. A visszaszámláló másodpercenként frissült és jelezte Aladár számára, hogy mennyi idő van még hátra a befejezésig.

13.4. **Vizsgázás szimulálása**

Aladár nevű felhasználóval ezután megválaszoltam az első kérdést. Ezt követően az alkalmazás megjelenítette a második kérdést Aladár számára.
A kontroller oldalon pedig megjelent Aladár válasza a vizsgázók, a kérdések és az eseménynapló képernyőkön.
Ezt követően lefuttattam a *answerAllTheQuestionsFor15Users()* szimulációt ami által véletlenszerűen másodpercenként egy vizsgázó megválaszolt egy kérdést. A szimuláció futása közben folyamatosan frissülnek a kontroller alkalmazásban az említett három képernyő táblázatai.

13.5. **Értesítés küldése**

A kontroller felhasználóval az "Értesítés küldés" menüponton keresztül egy új értesítést hoztam létre. Ezt követően az értesítés automatikusan megjelent az Aladár nevű vizsgázó felületén. Gyakorlatban az összes felhasználó automatikusan megkapná a kontroller üzenetét.

13.6. **Hátralevő időt meghosszabbítása**

A kontroller felhasználóval a Vizsga menüpont alatt 3 perccel meghosszabbítottam a vizsgát. A meghosszabbítást követően az Aladár vizsgázó felületén a visszaszámlálóhoz automatikusan hozzáadódott a 3 perc.

13.7. **Új kérdés hozzáadása**

Az idő meghosszabbítása után szintén a Vizsga menüből a kontrollerrel hozzáadtam egy új kérdést a vizsgához. Ezt követően Aladár vizsgafelületén módosult a vizsga kérdések száma.

13.8. **Értékelés**

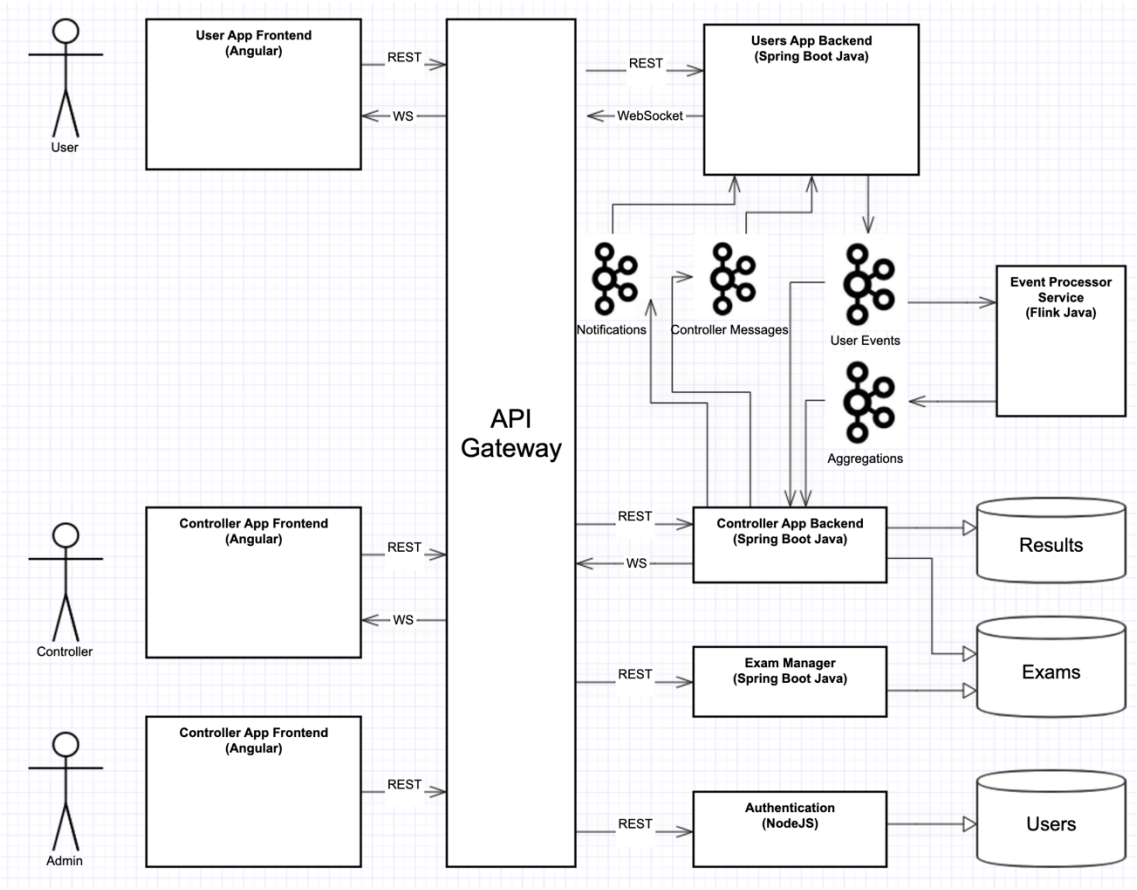
Az alkalmazás kipróbálásával megbizonyosodtam, hogy az implementált elosztott rendszer a várakozásoknak megfelelően működött és alkalmasnak bizonyult a megtervezett feleletválasztós vizsgázás lebonyolítására.

14. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

A példaprogram célja elsősorban a bemutatott architektúra gyakorlatba való átültetése volt. A szakdolgozatom keretein belül egy leegyszerűsített modellt terveztem meg és fejlesztettem le, ami jól szemlélteti a dolgozatom tárgyát. Ebből adódóan az implementált rendszernek vannak hiányosságai, hiszen egy valós életben használható szoftver létrehozása túllépne a szakdolgozat keretein. Továbbá ezek a fejlesztések nem kapcsolódnak szorosan a témaválasztásomhoz. Ezek közé sorolnám többek között azt, hogy az implementált rendszerből hiányzik a vizsgázók, illetve a tanár hitelesítése, a felhasználók megválaszolt kérdéseinek adatbázisban tárolása. Ezen felül jelenleg nincs lehetőség új kérdőívet létrehozni. Szintén hiányzik egy adminisztrációs felület, ahol egy tanár vagy egy adminisztrátor létrehozhat új vizsgasorokat. Ezeknek a problémáknak a kiküszöbölésére megalkottam egy kiterjesztett elosztott architektúrát, ami az implementált prototípusra épít. Természetesen a programot még számos más funkcióval lehetne kibővíteni.

14.1. **Kiterjesztett architektúra**

A rendszer architektúrájának moduláris jellegéből adódóan a programot a következőképpen lehetne továbbfejleszteni, így akár már egy éles környezetben is működhetne.



ábra 19 Lehetséges továbbfejlesztett architektúra

A prototípushoz képest az új fejlesztéseket röviden ismertetem.

14.2. API Kapu

Minden egyes felhasználói interakció az API kapun keresztül történik. Az API kapu egy felületet képez a külvilág, illetve a rendszer között. Többek között, egy proxyként szolgál, ami továbbítja a beérkező kéréseket az egyes szerver oldali komponensek felé. Ez az egyetlen belépési pont, amin keresztül a külső felhasználók, illetve külső rendszerek interakcióba tudnak kerülni a programmal. Továbbá az API kapu jellemzően egyéb hasznos funkciókat is tartalmaz, mint például korlátozza a nem kívánt felhasználókat, ha túl sok kéréssel terhelik a rendszert.

14.3. Hitelesítés

A vizsgázó, és a tanár hitelesítéséért a hitelesítő komponens felel. Legfőbb feladata a rendszerhez és az adatokhoz történő illetéktelen hozzáférés megakadályozása. A komponens a gyakorlatban jól bevált OAuth 2.0 [25] nevű állapotmentes hitelesítési módszert használja. Gyakorlatban az API kapu felügyeli a hitelesítési folyamatot, ami annyit jelent, hogy minden egyes beérkező kérést továbbít a hitelesítő komponens felé.

14.3.1. Regisztráció

A regisztráció a felhasználókezelés szerves része és szorosan kapcsolódik a hitelesítéshez és a jogosultságkezeléshez. Az adminisztrátor, vagy pedig maga a felhasználó kezdeményezi a regisztrációt a webes komponenseken keresztül. A regisztráció végeztével a hitelesítő komponens elmenti a felhasználót a hozzá tartozó adatbázisba.

14.3.2. Bejelentkezés

A felhasználó a webes felületen keresztül megadja a felhasználó nevét, valamint a jelszavát, amit a felület továbbít a hitelesítő komponens felé. A rendszer megvizsgálja, hogy a felhasználó szerepel-e az adatbázisban. Ha igen akkor összehasonlítja a jelszó SHA-256 hasító függvény a felhasználó mellett tárolt jelszó hasított értékével és egyezés esetén létrehoz egy JWT (JSON Web Token) karakterláncot, amit továbbít a kliens oldali webes alkalmazásnak.

14.3.3. További kérések

Minden egyes REST hívásnál a kliensalkalmazás elküldi egy HTTP fejléc formájában a JWT karakterláncot, amit a szerver oldalon levő API kapu továbbít a hitelesítő

komponens felé. A hitelesítő komponens pedig ellenőrzi a JWT karakterlánc érvényességét.

14.4. **Adminisztrációs felület**

Ez egy új komponens az adminisztrátorok számára, akik ezen a felületen keresztül létrehozhatnak új vizsgasorokat, amiket a vizsga menedzser komponens az adatbázisba eltárol. Ezt követően a kontroller fél kiválaszthatja és elindíthatja a vizsgát az új feladatsorral.

14.5. **Adatok tárolása**

Az adatokat célszerű tartósan adatbázisban tárolni, elsősorban azért, hogy az adatok ne vesszenek el, például, ha az alkalmazás szerver újraindul.
Továbbá, ha a komponens a feldolgozáshoz szükséges adatokat és az állapotot az adatbázisban tárolja akkor a komponens architektúra szempontból állapotmentesnek mondható. Az állapotmentesség lehetőséget ad arra, hogy a komponenst egyidejűleg több példányban futtassuk, illetve jelentősen megkönnyíti a szervíz teherelosztását az egyes komponensek között.

14.5.1. Felhasználók tárolása

A regisztrációt követően a felhasználók adatait adatbázisban tárolja el a rendszer. A bejelentkezést követően pedig ezeket hasonlítja össze a rendszer a felhasználó által megadott értékekkel.

14.5.2. Vizsgák tárolása

Új vizsgákat az adminisztrációs felületen keresztül lehet létrehozni. Ezeket a kéréseket a vizsga menedzser komponens menti el az adatbázisba. A létrehozott vizsgák kiválasztása továbbra is a kontroller feladata. A kontroller szempontjából az eredeti architektúrához képest a különbség annyi, hogy a kontroller szerveroldali komponense ez esetben az adatbázisból olvassa ki a vizsgákat, nem pedig a fájlrendszerből.

14.5.3. Válaszok tárolása

A kontroller alkalmazás a hozzárendelt adatbázisban tárolja el a felhasználók válaszait.
A prototípusban a válaszok a kontroller alkalmazás memóriájában vannak tárolva, tehát elvesznek abban az esetben, ha az alkalmazás újraindul.

14.6. **Kubernetes**

A Kubernetes a konténerek menedzselésére szolgáló program. A bemutatott alkalmazás több Docker konténert használ. Ahhoz, hogy ezeket a konténereket több szerverre telepítsük és egy tényleges elosztott alkalmazásként működjön a Kubernetes egy megoldást nyújt. A Kubernetes jellemzően egy számítógépfürt (cluster)-ben van telepítve és a *cluster worker node*-ra telepíti a futtatható konténereket. Az egyes konténerek a redundancia, illetve a terheléselosztás végett általában több példányban futnak.

15. ÖSSZEFOGLALÁS

A szakdolgozatom célja egy olyan többszereplős rendszer bemutatása volt, ami valós időben továbbítja, feldolgozza, valamint megjeleníti a betáplált adatokat, emellett egy központi szereplő felügyeli és szabályozhatja az egész rendszer működését. Ezt a rendszert egy vizsgáztató alkalmazás keretein belül szemléltettem.
Az egyetemi tanulmányaim és a szoftverfejlesztési szakirány is motivált, illetve jó alapot nyújtott ahhoz, hogy egy teljes szoftverfejlesztési folyamatot bemutassak az architektúrán keresztül az implementáción át a tesztelésig.

A feladat megoldásához a webfejlesztésben hagyományosan használt megoldások nem bizonyultak alkalmasak, ezért egy egyedi elosztott architektúrát terveztem. Törekedtem az architektúrát általánosan összerakni, hogy minél több hasonló problémára felhasználható legyen. A tervezésnél bemutattam az elosztott rendszer sajátosságait, és részletesen ismertettem a komponensek közötti kommunikációs technikákat. A célom egy olyan modern architektúra létrehozása volt, ami a legújabb technológiákat használva egy jó alapot biztosít akár komplexebb, teljesítményigényes rendszerek felhasználására. Az alkalmazás tervezésénél fontosnak tartottam, hogy az egyes komponensek között a függőségeket amennyire lehet, minimalizáljam. Ez egy jól bevett szokás az elosztott rendszerek felépítésénél, ami rugalmas és könnyen tovább fejleszthető architektúrát eredményez. Ezért bemutattam, hogy miként lehetne a rendszerhez hozzáépíteni további komponenseket anélkül, hogy nagyobb változtatásokat végeznék rajta.

Az implementáció bemutatása során nagyobb figyelmet fordítottam a valós idejű eseményfeldolgozásra. Ezt különösen fontosnak tartottam szemléltetni, mivel az adatfolyam működése szemléltette a valós idejű adatábrázolást és szabályozást. A vizsgaválasztó program a javasolt továbbfejlesztéseket követően vizsgaközpontokban, vagy akár iskolákban is felhasználható lehet és egy modern felhasználói élményt biztosít. Tapasztalataim szerint napjainkban a valós idejű kommunikáció és adatábrázolás az élet számos területén megtalálható. Már a hétköznapi felhasználók is push üzeneteken keresztül azonnal megkapják okosórájukon vagy telefonjukon keresztül a legfrissebb híreket. Egyre okosabbak az otthonaink, reagálnak a külső eseményekre és rendszeresen értesítik a lakót a legfrissebb állapotokról. Úgy gondolom, hogy sikerült egy megoldást adnom a modern felhasználói igényekre.

16. SUMMARY

The aim of my thesis was to present a multi-actor system that transmits, processes, and displays data in real time. The system is supervised by a central controller user, who monitors the actions and overviews the whole process. This behavior is demonstrated using a new generational exam application.
My university studies with the Software Engineering specialization motivated me and gave me a strong background to design and implement a distributed system using the best practices within the entire software development lifecycle.
The traditionally used approaches in web development, are not suitable to implement an event-driven distributed system with real-time communication capabilities. Therefore, there was a need to come up with an architecture that can be used to develop such systems. The uniqueness of the architecture lies mainly in the communication between the components. I described in dept how I achieved real-time event processing using Apache Flink, the open-source stream processing framework. The distributed architecture uses three different communication approaches, which I explained in detail within my thesis. My goal was to create a generic system design using the latest cutting-edge technologies, which can be reused to build various other even more complex highly performant systems. During the application design, I put emphasis on minimizing the dependencies across the components. It's a best practice in the distributed system design that allows the creation of more easily extendable systems. The implemented exam application is a proof-of-concept lacking data persistence, and authentication. However later in my thesis, I enhanced the architecture into a more complex system to prove the extensibility of my design and to demonstrate how can the prototype be turned into a production application.

Overall, the implemented application successfully showed the capabilities of the architecture, how it could be used to implement real-time data flow processing and how to visualize the computed data. The exam application could be used in schools or in exam centers after productizing it using the application as it's suggested in my thesis.

Real-time communication, data visualization, and event processing are hot topics these days and can be found in many different aspects of our lives. The everyday end users receive the news via push notifications either on smartphones or in their smartwatches. We produce massive amounts of data each day which is processed by different internal and external systems. Our homes are getting smarter, processing external events, and notifying the residents. We are living in a complex world where processing and delivering the right data to the right users is more important than ever. I think my thesis provided some insights into how to approach these challenges and also provided a potential solution to a set of problems with real-time event processing needs.

17. IRODALOMJEGYZÉK

[1] EVOLUTION OF THE WORLD WIDE WEB: FROM WEB 1.0 TO 6.0
https://www.edtech1.com/Evolution%20of%20the%20World%20Wide%20Web,%20From%201.0%20to%206.0.pdf

[2] Munkamenet (Session)
https://hu.wikipedia.org/wiki/Munkamenet (utoljára megtekintve: 2022. 12. 09)

[3] Web 2
https://en.wikipedia.org/wiki/Web_2.0 (utoljára megtekintve: 2022. 12. 09)

[4] Sam Newman: Building Microservices, 1st Edition, 2014

[5] The Difference Between Tight Coupling and Loose Coupling
https://nordicapis.com/the-difference-between-tight-coupling-and-loose-coupling/
(utoljára megtekintve: 2022. 12. 11)

[6] Joshua Bloch: Effective Java 3rd Edition, 2017

[7] Convention over configuration
https://en.wikipedia.org/wiki/Convention_over_configuration (utoljára megtekintve: 2022. 12. 11)

[8] Craig Walls: Spring Boot in Action, 2016

[9] TCP IP
https://hu.wikipedia.org/wiki/TCP/IP (utoljára megtekintve: 2022. 12. 10)

[10] UDP
https://en.wikipedia.org/wiki/User_Datagram_Protocol (utoljára megtekintve: 2022. 12. 10)

[11] Netty
https://netty.io/ (utoljára megtekintve: 2022. 12. 11)

[12] Reactor tervezési minta
https://projectreactor.io/ (utoljára megtekintve: 2022. 12. 11)

[13] Jeremy Wilken: Angular in Action, 2018

[14] RxJS
https://rxjs.dev/ (utoljára megtekintve: 2022. 12. 11)

[15] Neha Narkhede: Kafka The Definitive Guide, 2017

[16] Apache Flink
https://en.wikipedia.org/wiki/Apache_Flink (utoljára megtekintve: 2022. 12. 10)

[17] Objektum Orientált Programozás
https://hu.wikipedia.org/wiki/Objektumorient%C3%A1lt_programoz%C3%A1s
(utoljára megtekintve: 2022. 12. 10)

[18] Funkcionális Programozás
https://hu.wikipedia.org/wiki/Funkcion%C3%A1lis_programoz%C3%A1s (utoljára megtekintve: 2022. 12. 10)

[19]Apache Maven
https://hu.wikipedia.org/wiki/Apache_Maven (utoljára megtekintve: 2022. 12. 10)

[20] Npm
https://en.wikipedia.org/wiki/Npm_(software) (utoljára megtekintve: 2022. 12. 10)

[21] Git
https://en.wikipedia.org/wiki/Git (utoljára megtekintve: 2022. 12. 10)

[22] James Turnbull: The Docker Book: Containerization is the new virtualization, 2014

[23] Fekete doboz tesztelés
https://hu.wikipedia.org/wiki/Feketedobozos_tesztel%C3%A9s (utoljára megtekintve: 2022. 12. 10)

[24] Viselkedés tesztelés
https://hu.wikipedia.org/wiki/Viselked%C3%A9svez%C3%A9relt_fejleszt%C3%A9s
(utoljára megtekintve: 2022. 12. 10)

[25] OAuth 2.0
https://www.rfc-editor.org/rfc/rfc6750 (utoljára megtekintve: 2022. 12. 11)

18. RÖVIDÍTÉSEK

AJAX - Asynchronous JavaScript and XML
API – Application Programming Interface
BDD - Behavior-Driven Development
CSS – Cascading Style Sheets
HTML - HyperText Markup Language
HTTP - Hypertext Transfer Protocol
JDK 11 - Java Development Kit
JSON – JavaScript Object Notation
JVM – Java Virtual Machine
JWT – JSON Web Token
MVC - Model View Controller
NPM - Node Package Manager
POJO – Plain Old Java Object
RAD - Rapid Application Development
REST - Representational State Transfer
RxJs – Reactive Extensions for JavaScript
SPA - Single Page Application
TCP – Transmission Control Protocol
UDP - User Datagram Protocol
URL – Uniform Resources Locator

19. ÁBRAJEGYZÉK