



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Programozási nyelvek és Fordítóprogramok
Tanszék

Tudástér – felhő alapú oktatási rendszer

Témavezető:

Kitlei Róbert László

mestertanár

Szerző:

Tóth Zalán

programtervező informatikus BSc

Budapest, 2023

Tartalomjegyzék

1.	Bevezetés	3
2.	Felhasználói dokumentáció.....	4
2.1	Az alkalmazás felépítése	4
2.2	A rendszer erőforrás igénye	5
2.3	Telepítés.....	6
2.4	A program használata	7
2.4.1	Adminisztrátor.....	8
2.4.2	Tanár	9
2.4.3	Diák.....	15
3.	Fejlesztői dokumentáció	20
3.1	Mikroszolgáltatások.....	20
3.2	Tudástér felépítése	22
3.2.1	Magas szintű rendszerterv	22
3.2.2	Spring Boot.....	26
3.2.3	Kotlin	29
3.2.4	Általános funkcionalitás	29
3.2.5	Api Gateway	45
3.2.6	Felhasználók kezelése	47
3.2.7	Tantárgy szolgáltatás.....	50
3.2.8	Tantárgy felvétele szolgáltatás.....	61
3.2.9	Videók lejátszása	62
3.2.10	Kérdéssor szolgáltatás	65
3.2.11	Jegyek szolgáltatás	68
3.2.12	Értesítések szolgáltatás	71
3.2.13	Felhasználói felület.....	72
3.3	Futtatható állomány készítése.....	74

3.4	Kubernetes alapú futtatás	77
3.4.1	Kubernetes alapok	78
3.4.2	Telepítés leírók	79
3.4.3	JVM és Kubernetes	88
3.5	Az alkalmazás monitorozása	89
3.6	Tesztelés.....	92
3.7	Kubernetes telepítése.....	99
3.8	Hasznos Kubernetes parancsok	102
3.9	Egyéb telepítési módszerek	102
3.10	Jelszavak.....	105
3.11	Felhasznált eszközök.....	105
3.12	Továbbfejlesztési lehetőségek.....	106
	Irodalomjegyzék.....	108

1. fejezet

Bevezetés

Napjainkban egyre több tevékenységünket végezhetjük az online térben. Gondoljunk csak a vásárlásokra, kapcsolattartásra barátainkkal vagy akár az oktatásra.

Az ezen funkcionalitásokat biztosító szoftvereket több millió ember használja a nap 24 órájában. A kiélezett piaci verseny miatt ezeknek a rendszereknek magas rendelkezésre állásúnak, hibatűrőnek és megfelelő sebességűnek kell lenniük, egyébként a felhasználók alternatív szolgáltatás után néznek.

Szakdolgozatomban egy online oktatási rendszer példáján keresztül mutatom be egy ilyen szoftver felépítéséhez szükséges alapvető tervezési-, és megvalósítási mintákat, a rendszer előnyeit, illetve hátrányait.

A Tudástér egy felhőben futó, szoftver mint szolgáltatás - SAAS - típusú oktatási felület. Az adminisztrátorok által regisztrált oktatási intézményekhez az adott intézményhez tartozó tanárok és diákok hozzárendelhetők. A tanári hozzáférési joggal rendelkező felhasználók létrehozhatnak új tantárgyakat, melyekhez tanórákat rendelhetnek. Lehetőségük van videók feltöltésére a tanórákhoz, illetve kérdéssorok összeállítására. A diák jogosultsággal rendelkező felhasználók az intézmény által meghirdetett tantárgyak felvétele után az órákhoz tartozó videó tananyagon keresztül elsajátíthatják a kérdéssor megválaszolásához szükséges tudást, melyről a kérdéssor kitöltésével adnak számot. Ennek kiértékelése, illetve a végső jegy meghatározása automatikusan történik.

A szoftver működéséhez szükséges szolgáltatásokat mikroszolgáltatás architektúra segítségével valósítottam meg, mely futtatása konténer alapú alkalmazáskezelő szoftver segítségével történik.

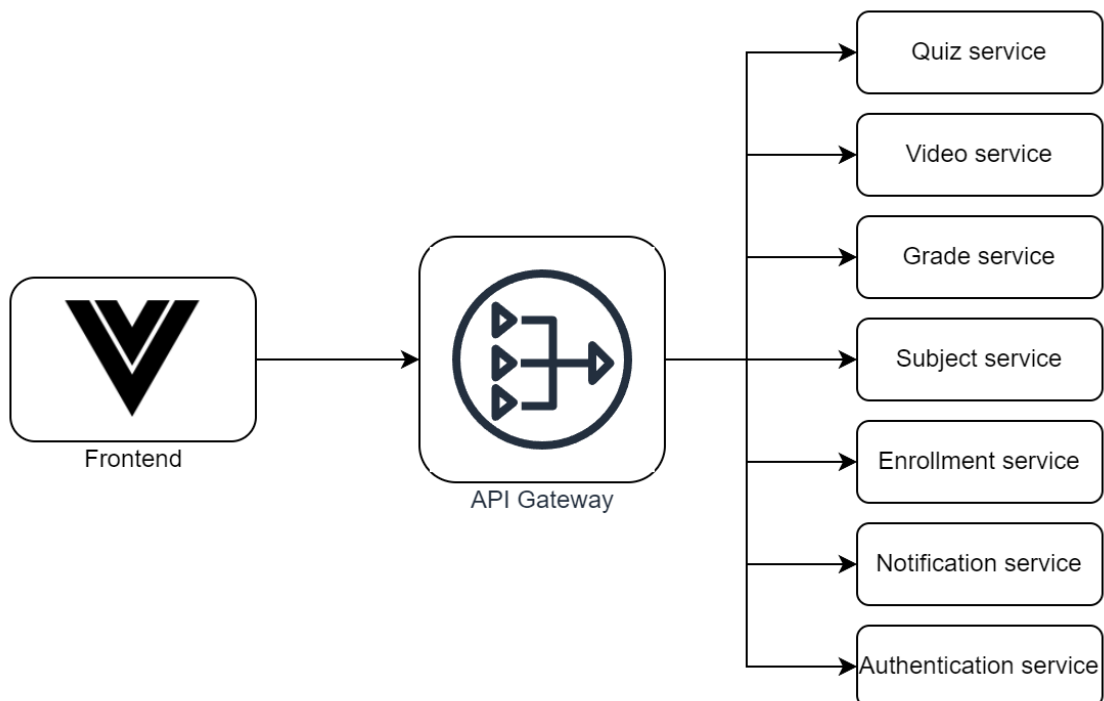
2. fejezet

Felhasználói dokumentáció

A fejezetben kerül bemutatásra a szoftver alapvető felépítése és a telepítéséhez, valamint használatához szükséges információk.

2.1 Az alkalmazás felépítése

Az alkalmazás összesen nyolc különálló mikroszolgáltatásból, – röviden ms – illetve a felhasználói felületből áll, melyek együttesen biztosítják a rendszer működéséhez szükséges funkcionalitást. Minden ms a szoftver egy jól definiált tartományáért – más néven domain - felelős. Ennek adatait tárolja, módosítja, illetve szolgálja ki a felhasználó, valamint a többi ms felé.



1. ábra Komponensek

A 1. ábra bemutatja az alkalmazást felépítő szoftver komponenseket. A könnyebb áttekinthetőség érdekében az alkalmazás futtatásához szükséges háttérszolgáltatásokat - mint például az adatbázisok és üzenetsorok - és az ms-ek közötti kapcsolatokat az ábra nem tartalmazza. A felhasználó a Frontend webalkalmazáson keresztül küldhet kéréseket a háttérszolgáltatások felé. A kérések belépési pontja az API Gateway, mely ezeket a megfelelő ms felé irányítja, valamint biztosítja a felhasználók hitelesítését. Az ms-ek a saját domain modelljük kezeléséért és kiszolgálásáért felelnek. Így például a Grade Service a jegyek beírását és lekérdezését végzi, míg a Subject Service felelős a tantárgyak összeállításának kezeléséért és kiszolgálásáért. Az alkalmazást alkotó modulokat az 1. táblázat foglalja össze röviden.

Modul neve	Leírás
Frontend	A felhasználói felületet megjelenítését végzi.
API Gateway	Belépési pont az ms-ek felé.
Authentication Service	A felhasználók kezelésére szolgáló modul.
Subject Service	A tantárgyak karbantartásáért felel.
Enrollment Service	A tantárgyfelvételi adatokat tárolja.
Video Service	A videók feltöltését és lejátszását végzi.
Quiz Service	A kérdéssorok kezelését és kiértékelését oldja meg.
Grade Service	A pontszámok és jegyek mentéséért felel.
Notification Service	A felhasználói értesítéseket tárolja és szolgálja ki.

1. táblázat Az alkalmazást felépítő programok

2.2 A rendszer erőforrás igénye

A szoftver összesen kilenc alkalmazásból áll, emellett a futtatáshoz szükséges az adatbázis rendszerek, üzenetsorok - MQ - futtatása, valamint opcionálisan a monitorozást segítő szoftverek erőforrás igényével is számolni kell. Az ms-ek több példányban futhatnak az aktuális terheléstől függően. A telepítő csomag két Kubernetes környezetet tartalmaz. A 2. táblázat ezen környezetek erőforrásigényét mutatja be, mely a Kubernetes alapú futtatás fejezetben részletesen kifejtésre kerül.

Környezet	Processzor	Memória	Docker memória beállítás
dev	2GHz 5 core	6.5 GB	8 GB
prod	2GHz 6 core	9 GB	12 GB

2. táblázat Hardverigény

A megadott erőforrásokkal a prod rendszerek indítási ideje - letöltött képfájlok esetén - körülbelül egy perc, míg a dev környezeté három. A telepítő tartalmaz alternatív telepítési módszereket, mint például lokális futtatás és konténerizált futtatás Docker-ben.

2.3 Telepítés

A szoftvert Kubernetes – röviden k8s - alkalmazáskezelő platformon történő futtatásra terveztem. Az alkalmazás telepítésének előfeltétele ezen platform megléte. A Kubernetes telepítése fejezet ehhez részletes útmutatót tartalmaz.

Az alkalmazás telepítése a Kubernetes parancssori programjának – kubectl - segítségével történik. A mellékelt programcsomagban található deployment könyvtárban találhatók a telepítésleíró fájlok. Ezek szolgálnak a parancssori eszköz bemeneteként, mely elvégzi a konténerek letöltését és a teljes rendszer telepítését, indítását, valamint a hálózat, tűzfalak, adatbázisok és tárterület létrehozását.

A telepítés indításához a fent említett könyvtárban ki kell adni a következő parancsot:

```
kubectl apply -k ./overlays/<környezet>
```

ahol a környezet lehetséges értékei:

- *dev* – csak a feltétlenül szükséges konténerek kerülnek telepítésre.
- *prod* – minden konténer telepítésre kerül és unix alapú tárterületet is létrehoz.
- *prod-win-wsl* – a prod környezet Windows operációs rendszerhez.

A parancs létrehozza a szükséges erőforrások leíróit, melyet a Kubernetes értelmez és a háttérben elindítja a leíró fájlok által definiált erőforrások létrehozását. Az első telepítés alkalmával a konténerek letöltése sávszélességtől függően több percet is igénybe vehet. A telepítés státusza a `kubectl get pods` paranccsal ellenőrizhető, melynek kimenetét a 2. ábra ismerteti. Ha minden szolgáltatás státusza Running és legalább 1 példány

Ready, az alkalmazás használatra kész. További hasznos parancsok a 3.8 fejezetben találhatók.

```
λ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
api-gateway-deployment-7f97dcfb68-6sjcb	1/1	Running	0	114s
auth-service-deployment-54c64bff9-zjqqq	1/1	Running	0	114s
enrollment-service-deployment-8d6bfccb4-ttshq	1/1	Running	0	114s
grade-service-deployment-b8748bd75-t4hcm	1/1	Running	0	114s
knowhere-ui-deployment-85f977575d-n4xxs	1/1	Running	0	114s
mongodb-0	1/1	Running	0	113s
notification-service-deployment-58f5f6c94d-p5x5v	1/1	Running	0	114s
postgres-0	1/1	Running	0	113s
quiz-service-deployment-8f4f5f87f-h2pww	1/1	Running	0	113s
rabbitmq-0	1/1	Running	0	113s
spring-admin-deployment-896cb59b-6r89x	1/1	Running	0	113s
subject-service-deployment-58b4ccdd54-7lm1p	1/1	Running	0	53s
subject-service-deployment-58b4ccdd54-hf45p	1/1	Running	0	113s
video-service-deployment-6778c5699d-4cfw6	1/1	Running	0	113s
zipkin-deployment-6fb9455b9d-bnm48	1/1	Running	0	113s

2. ábra Alkalmazás státusza

Az egyes komponensek az első indítás alkalmával többször újra indulhatnak, ez természetes jelenség Kubernetes alapú futtatásnál. Az alkalmazás ezután a k8s klaszter hálózati címén – helyi hálózat esetén ez localhost – érhető el, a 80-as porton.

2.4 A program használata

Az URL cím megnyitásával a landoló oldal fogadja a látogatót. Amennyiben az intézmény még nincs regisztrálva, az intézmény felelőse ezt a Regisztráció menüpont alatt megteheti. Ekkor a felhasználó bejelentkezési fiókja is létrejön adminisztrátori jogkörrel.

Intézmény azonosító	Adminisztrátor	Tanár	Diák
elte	admin@elte.hu	teacher0@elte.hu teacher1@elte.hu teacher2@elte.hu	student0@elte.hu student1@elte.hu student2@elte.hu student3@elte.hu student4@elte.hu
pte	admin@pte.hu	teacher0@pte.hu teacher1@pte.hu teacher2@pte.hu	student0@pte.hu student1@pte.hu student2@pte.hu student3@pte.hu student4@pte.hu

3. táblázat Kezdeti felhasználók

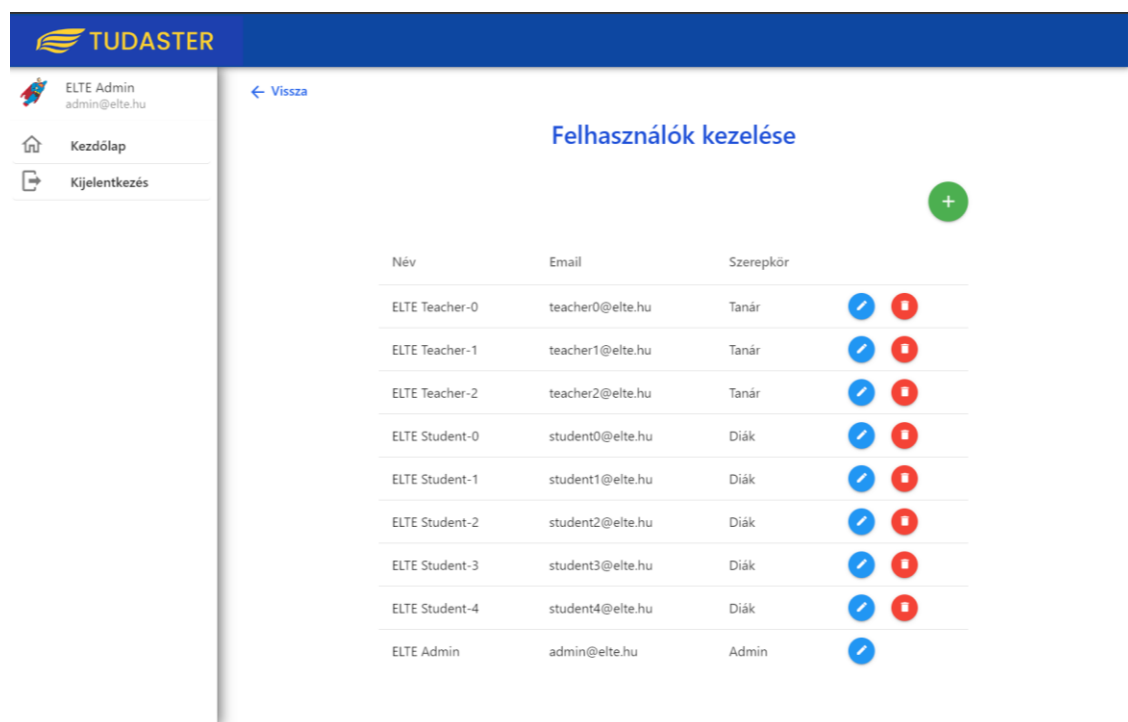
A továbbiakban az adminisztrátor vehet fel új felhasználókat a rendszerbe és ő oszthatja ki hozzájuk a megfelelő jogosultságokat. A rendszer három jogosultsági szintet különböztet meg – Adminisztrátor, Tanár, valamint Diák. Az elérhető funkcionalitás jogkörönként különböző. Az adminisztrátor által felvett felhasználók az adott intézményhez kötöttek, így más intézmények adatait nem láthatják.


















A 3. táblázat tartalmazza a rendszer indításakor automatikusan létrehozásra kerülő felhasználókat, melyek segítségével a rendszer gyorsan tesztelhető. A bejelentkezési jelszavak rendre *admin*, *teacher*, valamint *student*. Az elte intézményen belül az admin, teacher0 és student0 felhasználókhoz adatokat is előkészítettem.

Amennyiben a felhasználó rendelkezik hozzáféréssel a rendszerhez, a bejelentkezés menüponton keresztül az intézmény azonosító, email cím és jelszó megadásával bejelentkezhet.

A továbbiakban az egyes jogkörök szerint mutatom be az elérhető funkcionalitásokat.

2.4.1 Adminisztrátor

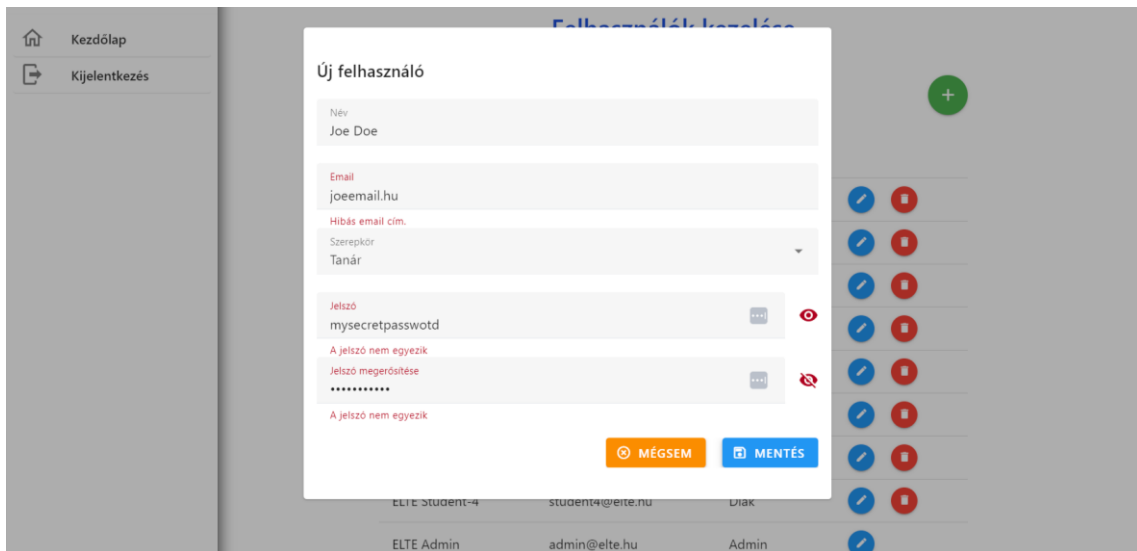


Név	Email	Szerepkör	
ELTE Teacher-0	teacher0@elte.hu	Tanár	 
ELTE Teacher-1	teacher1@elte.hu	Tanár	 
ELTE Teacher-2	teacher2@elte.hu	Tanár	 
ELTE Student-0	student0@elte.hu	Diák	 
ELTE Student-1	student1@elte.hu	Diák	 
ELTE Student-2	student2@elte.hu	Diák	 
ELTE Student-3	student3@elte.hu	Diák	 
ELTE Student-4	student4@elte.hu	Diák	 
ELTE Admin	admin@elte.hu	Admin	

3. ábra Adminisztrátor kezdőlap

Az adminisztrátor jogosult felhasználókat regisztrálni az intézményhez, törölni őket, valamint módosítani az adataikon. Ez utóbbit a felhasználók maguk is megtehetik a bal felső sarokban a saját nevükre kattintva.

A kezdőlapon található táblázatban látható a felhasználók listája, illetve minden sorban az adott felhasználó módosítására és törlésére szolgáló gombok. Az oldal tetején található Hozzáadás gomb segítségével pedig új felhasználó adható a rendszerhez. Adminisztrátori fiók nem törölhető a rendszerből.



4. ábra Adminisztrátor, új felhasználó

A 4. ábra bemutatja az Új felhasználó funkció működését. Az űrlap mezői ellenőrzésre kerülnek elküldés előtt. Az esetleges kitöltési hibákról a rendszer értesíti a felhasználót és megakadályozza a mentést. Amennyiben minden mező kitöltése megfelelő, a mentés megtörténik, majd ezt követően a felhasználó lista frissül az új-, vagy módosított felhasználó adataival.

2.4.2 Tanár



5. ábra Tanár kezdőlap

A tanári jogosultsággal rendelkező felhasználóknak - a belépést követően - a kezdőlapon megjelenítésre kerülnek az általuk összeállított aktív tantárgyak. Ezt mutatja az 5. ábra. Az egyes tantárgyak kártyái tartalmazzák annak nevét, elérhetőségének időintervallumát, melyet a kezdő-, és végdátum reprezentál. A tantárgy lehet folyamatos elérésű is, amennyiben nincs végdátum beállítva. Ez alatt található a tantárgy leírása. Hosszabb szöveg esetén csak az első 150 karakter jelenik meg. Majd a leírást követi a tantárgy kredit értéke a bal oldalon, illetve az elfoglalt és elérhető férőhelyek száma jobbra.

Lehetőség nyílik az adott tantárgy adatlapjára navigálni a tantárgy kártyáján történő kattintással, új tantárgyat felvenni, valamint az archivált tantárgyak megjelenítésével régebbi tantárgyakat böngészni. Az oldalon egyszerre legfeljebb 8 kártya jelenik meg. További tantárgyak megtekintésére lapozás segítségével van lehetőség.

Az Új tantárgy gombra, illetve egy kiválasztott tantárgy kártyájára kattintva lehet eljutni tantárgyi adatlapra. Itt található a tantárgy és a hozzá tartozó órák alapvető információi, valamint a tárgyat felvett hallgatók haladása is nyomon követhető az oldalon.

The screenshot displays the 'Java' course data page. At the top, the course name 'Java' is shown with a '(Publikálva)' status. Below this, the course description 'Programozás java nyelven' is visible. The credit value is 5, and the capacity is 34. The course start date is 2023. 01. 10. 10:00, and the end date is 'éééé. hh. nn. --:--'. The course is marked as 'Mentés' (Save) and 'Archiválás' (Archive). The 'Hallgatók' (Students) section shows progress bars for 'ELTE Student-0', 'ELTE Student-1', and 'ELTE Student-3'. On the right side, there are several informational boxes: 'A Java nyelv története' (History of the Java language), 'Alapvető vezérlési elemek' (Basic control elements), 'Osztályok' (Classes), 'Gyűjtemények' (Collections), and 'Generikus típusok' (Generic types).

6. ábra Tantárgy adatlap

A felhasználó megadhatja a tantárgy nevét, leírását, kredit értékét. Emellett beállítható a férőhelyek száma, mely maximalizálja a tantárgyat felvehető hallgatók számát. A kezdődátum és végdátum beállításával behatárolható a tantárgy elérhetőségének időintervalluma. A diákok a tantárgyat felvehetik a kezdőidőpont előtt is, de csak ebben az intervallumban tudják megnyitni és teljesíteni a tantárgyhoz tartozó tanórákat. Amennyiben valamely diák az adott intervallumon belül nem végzi el az összes órát, az érte járó krediteket sem kapja meg. Opcionálisan feltölthető borítókép a tantárgyhoz, mely a tantárgy kártyáinak díszítésére szolgál, illetve segít felkelteni a hallgatók érdeklődését.

Ezután következik a vezérlőgomb panel. Az adatlapon történt módosítások a mentés gomb segítségével véglegesíthetők. Amennyiben új tantárgyat hoz létre a felhasználó, először ki kell töltenie az összes kötelező mezőt és menteni a tantárgyat. Tanórák hozzáadására csak ezután van lehetősége. A Publikálás gomb segítségével a tantárgy a diákok számára is elérhetővé válik, akik így felvehetik azt. A tantárgy publikálásához legalább egy tanóra felvétele szükséges. A publikált tantárgy neve alatt megjelenik a Publikálva felirat. Amennyiben a tantárgy már nem aktuális, az Archiválás gomb segítségével inaktiválható. Ekkor a tantárgyat felvett hallgatók még elvégezhetik a tantárgyat – természetesen a határidők betartásával – de új diák már nem tudja felvenni. Az Archiválás visszavonása gomb segítségével ez a művelet visszavonható.


A gombsor alatt található a tantárgyat felvett hallgatók listája, valamint az általuk teljesített tanórák és az összes tanóra arányát bemutató, haladást jelző indikátor.

Az oldal jobb oldalán található a tantárgyhoz létrehozott órák listája. Az oldal tetején lévő Hozzáadás gomb segítségével felvehető új tanóra, az adott óra kártyájára kattintva pedig a meglévő módosítható.

A tanóra adatlapján kitöltendő annak neve, határideje, - mely előtt teljesíteni kell a kérdéssort - valamint leírása. Ezen felül az előzetesen feltöltött videók, illetve összeállított kérdéssorok – más néven kvízek - közül ki kell választani az órához tartozót. A Mentés gomb megnyomásával véglegesíthető az adatlap, mely siker esetén visszavigálja a felhasználót a tantárgyi adatlapra, ahol a tantárgylista frissítésre kerül. A 7. ábra ezt az oldalt szemlélteti.

[← Vissza](#)

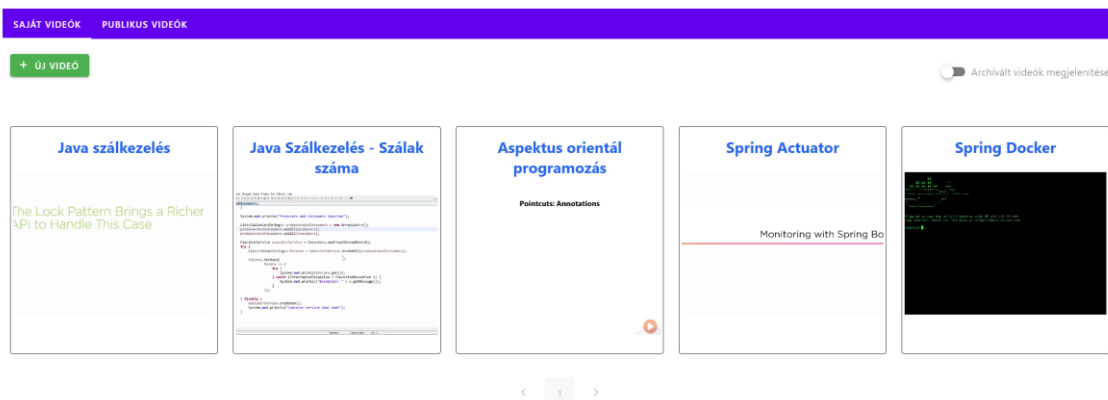
Új óra

Név Alapvető vezérlési elemek	Határidő 2023. 03. 31. 18:19
Leírás Adattípusok, ciklusok, elágazások és egyéb nyelvi elemek	
Video Java vezérlési szerkezetek	Kvíz Java - vezérlési elemek
	

7. ábra Tanóra adatlap

A főmenüből érhető el a 8. ábra által bemutatott Videók aloldal, ahol az eddig feltöltött saját, illetve publikus média anyagok találhatók, melyek között az oldal tetején helyet kapott tabfül címkéjén történő kattintással lehet váltani.

A videók feltöltési időrendben jelennek meg. Az oldal lapozás segítségével biztosítja a régebbi elemek betöltését. A kártyákra kattintva a felhasználónak lehetősége van az adott videót megtekinteni, illetve archiválni. Az Új videó gomb segítségével tölthető fel újabb média fájl.



8. ábra Videók


A 9. ábra a videó feltöltésére szolgáló oldalt mutatja be. A név megadása mellett egy videó fájl kiválasztása is kötelező. A rendszer jelenleg – környezettől függően - maximum 2 gigabájt méretig támogatja a feltöltést. A videó publikussá tehető, így az intézmény többi tanára is felhasználhatja azt tanórák összeállításához. A feltöltött videófájl

feldolgozása során a rendszer automatikusan kivág egy kulcs képkockát, mely borítóképként szolgál az oldalon. Az adatlap kitöltése és a feltöltés a Mentés gomb segítségével véglegesíthető. Nagyobb videónál és alacsony sávszélességnél ez sok időt vehet igénybe.

Új videó feltöltése

Név

Spring Applikáció monitorozása




Videó

03. Developing a Custom Health Check for the Demo Application.mp4

☐

Publikus



MENTÉS

9. ábra Videó feltöltése képernyő



A Kvízek menüpont a kérdéssorokat összegző oldalra irányítja a felhasználót, melyet a 10. ábra mutat be. A kártyák a kvíz címét, illetve leírásának első 150 karakterét tartalmazzák.

Kvízek

+ ÚJ KVÍZ



Java - vezérlési elemek

Ebben a kérdéssorban a második órán tanult vezérlési elemekre vonatkozó kérdésekre kell felelnie.



Kotlin - Lambda kifejezések

Lambda kifejezések felépítésére, inline, crossinline, noinline módosítók hatására vonatkozó kérdések.



10. ábra Kvízek

13

A bal oldalon található gombbal a kérdéssor archiválható. A jobb oldali ceruza ikonra kattintva pedig a szerkesztő képernyő nyílik meg az adott kérdéssorhoz. Az Új kvíz gomb szintén erre az oldalra navigál, értelemszerűen ilyenkor egy teljesen üres adatlap fogadja a felhasználót.

Java - vezérlési elemek

Cím

Java - vezérlési elemek

Leírás

Ebben a kérdéssorban a második órán tanult vezérlési elemekre vonatkozó kérdésekre kell felelnie.

MENTÉS

Kérdések

Kérdés

Mely vezérlési szerkezet segítségével tud végigiterálni egy tömb elemein

Pontszám

2

☐

Válasz

if

-

☒

Válasz

for

-

☐

Válasz

switch

-

+

✖

A kérdésnek legalább két válaszlehetőséget tartalmaznia kell.

Kérdés

Kötelező mező.

Pontszám

0

Kötelező mező.

+

✖

+

11. ábra Kéréssor összeállítása

A 11. ábra a Kéréssor összeállításának képernyőképet szemlélteti. A cím és leírás mezők kitöltése után az oldal közepén található zöld színű Hozzáadás gombbal lehet új kérdést felvenni. A kérdés, illetve pontszám mező kitöltése után a kék színű Új válaszlehetőség gombbal adhatók további válaszok a kérdéshez. Minden kérdésnek legalább két válaszlehetőséget kell tartalmaznia, illetve egy lehetőséget helyesnek kell jelölni. A piros Törlés gombbal az adott kérdés eltávolítható a kérdéssorból, míg a narancssárga Válaszlehetőség eltávolítása gombbal egy-egy válasz törlésére van lehetőség. A Mentés

gomb megnyomásával lehet az űrlapot elmenteni. Amennyiben nem található hiba a kitöltésben, a felhasználó visszairányításra kerül az összegző oldalra, ahol az újonnan felvett kérdéssor is láthatóvá válik. Validációs hiba esetén a mezők alatt a felhasználót a program tájékoztatja a hibáról és a mentés nem kerül végrehajtásra a hibák javításáig.

Tantárgy
Programozás

Programozás

Tanuló neve	Azonosító	Összpontszám	Érdemjegy
ELTE Student-0	student0@elte.hu	27	4
ELTE Student-1	student1@elte.hu	30	5

12. ábra Hallgatók értékelései

A 12. ábra a hallgatók értékeléseit összefoglaló képernyőképét mutatja be. Az oldal tetején a legördülő menüből kiválasztott tantárgyhoz listázásra kerülnek az azt elvégzett hallgatók. A lista tartalmazza a diák nevét, azonosítóját, valamint a kérdéssorokon elért összesített pontszámot és az ebből számított érdemjegyet.

2.4.3 Diák

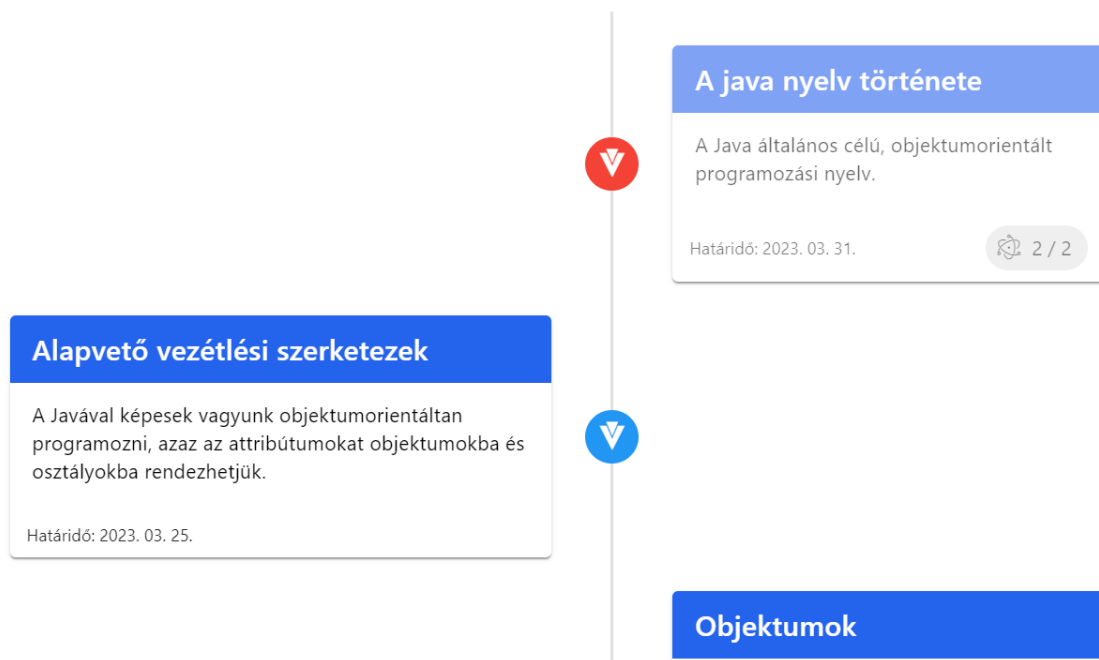
A diákok kezdőképernyőjének felépítése megegyezik a tanárokéval. Azonban náluk a felvett tantárgyak kártyái láthatók. Az oldalon két tabfül található, az elsőt az aktív tantárgyak láthatók, melyek elvégzése folyamatban van, míg a második a teljesített tantárgyak megtekintésére nyújt lehetőséget. Az aktív tantárgyak fölön lehetőség van a kártyákra kattintva a tantárgyi adatlapra navigálni.

A 13. ábra mutatja be a tárgyi adatlapot, melyen megtalálható a tantárgy szöveges leírása, valamint a hozzá tartozó órák leírással és határidővel együtt. Az oldal tetején a Tantárgy leadása gomb segítségével a hallgató leadhatja a tantárgyat. Erre az első tanóra teljesítése előtt van lehetősége. A már teljesített órák kártyájának jobb alsó sarkában található a megszerzett pontszám. Ezek a kártyák, valamint azok, amelyek határideje lejárt, letiltásra kerülnek, így a hallgató nem érheti el ezeket többé. A kártyákra kattintva az óra adatlapjára jutunk.

Java

⊖ TANTÁRGY LEADÁSA

A Java általános célú, objektumorientált programozási nyelv, amelyet a Sun Microsystems fejlesztett a '90-es évek elejétől kezdve egészen 2009-ig, amikor a céget felvásárolta az Oracle.



13. ábra Tantárgy adatlap

Az óra adatlapján a hallgató megnézheti a tanórához tartozó videót. Amennyiben nem fejezi be, az aktuális lejátszási időpont tárolásra kerül, így következő alkalommal onnan folytathatja, ahol abbahagyta. Az oldal alján lenyíló menüben található az órához tartozó leírás, illetve a tantárgy teljesítéséhez szükséges kérdéssor is. Minden kérdés mellett látható annak pontértéke, illetve a kérdés alatt a válaszlehetőségek, melyekből kattintással jelölhető be a helyesnek vélt válaszlehetőség. A kérdések megválaszolása után a Beküldés gomb segítségével véglegesíthető a kitöltés. Ezt a 14. ábra mutatja be. Ekkor a rendszer a tanulót visszairányítja a tantárgy adatlapjára. A kérdéssor kiértékelése a háttérben történik, a művelet végeztével a felhasználó értesítést kap az elért pontszámáról, mely az óra kártyáján is megjelenik a következő betöltésnél.

Java - vezérlési elemek

Ebben a kérdéssorban a második órán tanult vezérlési elemekre vonatkozó kérdésekre kell felelnie.

Mely vezérlési szerkezet segítségével tud végigiterálni egy tömb elemein

if

for

switch

Mely új funkcionalitást hozta magával a Java 8

Stream Api

Generikusok

BEKÜLDÉS

14. ábra- Kérdéssor

A főmenüből érhető el a Tantárgyak menüpont, ahol a meghirdetett, de a hallgató által még fel nem vett tantárgyakat lehet böngészni. A felület a korábban megismert kártya alapú lista segítségével jeleníti meg az adatokat. A kiválasztott tantárgy kártyára kattintva a 15. ábra által ismertetett tantárgyi összefoglaló oldal fogadja a látogatót, mely bemutatja az adott tantárgyat. Itt megtalálható a tantárgy neve és időpontja, valamint a leírása, továbbá a tárgy által tartalmazott órák, illetve látható a tantárgy kreditértéke is. Az óraleírások segítségével a tanuló képet kap, miről szól a tantárgy és mi szükséges a teljesítéséhez. A Tantárgy felvétele gombbal a diák jelentkezhet a tantárgyra. A gomb megnyomása után a tanuló a kezdőlapra kerül visszairányításra. A rendszer a háttérben megvizsgálja, felvehető-e a tantárgy a tanuló számára. Amennyiben igen, a tanuló hozzáfér a tantárgyi adatlaphoz és elvégezheti azt. Ellenkező esetben a rendszer üzenetet küld a hallgató számára az elutasítás indoklásával. Ilyen lehet például, ha a tantárgyra meghirdetett helyek időközben elfogytak.

Java

2023. 01. 10. - Korlátlan

5

A Java általános célú, objektumorientált programozási nyelv, amelyet a Sun Microsystems fejlesztett a '90-es évek elejétől kezdve egészen 2009-ig, amikor a céget felvásárolta az Oracle. A Java alkalmazásokat jellemzően bájtkód formátumra alakítják, de közvetlenül natív (gépi) kód is készíthető Java forráskódból. A bájtkód futtatása a Java virtuális géppel történik, ami vagy interpretálja a bájtkódot, vagy natív gépi kódot készít belőle, és azt futtatja az adott operációs rendszeren. Létezik közvetlenül Java bájtkódot futtató hardver is, az úgynevezett Java processzor.

Órák:

A java nyelv története

A Java általános célú, objektumorientált programozási nyelv, amelyet a Sun Microsystems fejlesztett a '90-es évek elejétől kezdve egészen 2009-ig, amikor a céget felvásárolta az Oracle.

Alapvető vezérlési szerke...

A Javával képesek vagyunk objektumorientáltan programozni, azaz az attribútumokat objektumokba és osztályokba rendezhetjük. Amint definiálod ezeket a programodban, később utalhatsz rájuk, így nem kell újra leírnod az egész kódot. A procedurális programozási nyelvekben mindig ki kell írni az egymástól függő hosszú kódsorokat. A praktikus ki objektumaink nélkül mindig újra le kell írni mindent.

Objektumok

A Javával képesek vagyunk objektumorientáltan programozni, azaz az attribútumokat objektumokba és osztályokba rendezhetjük. Amint definiálod ezeket a programodban, később utalhatsz rájuk, így nem kell újra leírnod az egész kódot.

Generikus Típusok

A generikusok megjelenése a nyelvben a Java 5 nyelvi bővítése. Ahogy láttuk már korábban, nagyon nagy szerepük lesz abban, hogy az objektumokat tároló kollekciókat és lekérdezéseket hatékonyan és biztonságosan tudjuk használni. Kicsit olyanok a Java generikusok, mint a C++-os template-ek. De csak kicsit, valójában számos technikai különbség van a kettő között, amikre most nem fogunk kitérni, célunk csak egy általános ismertetést adni, hogyan is kell ezeket a generikus dolgokat létrehozni, kezelni.

TANTÁRGY FELVÉTELE

15. ábra Tantárgyi összefoglaló

A hallgatónak lehetősége van a Jegyek menüpont alatt a tanulmányi előmenetelük megtekintésére. Az oldal tetején, bal oldalon található a megszerzett kreditek összege. Alatta pedig egy táblázat mutatja a teljesített tantárgyakat. Itt látható a teljesített tantárgy neve, a megszerzett kreditszám, mely nem teljesített tárgy – ha nem töltötte ki az összes kvízt – illetve elégtelen érdemjegy esetén nulla. Az táblázatban megtalálható még a tanuló által elért pontok száma, illetve a számított érdemjegy is.

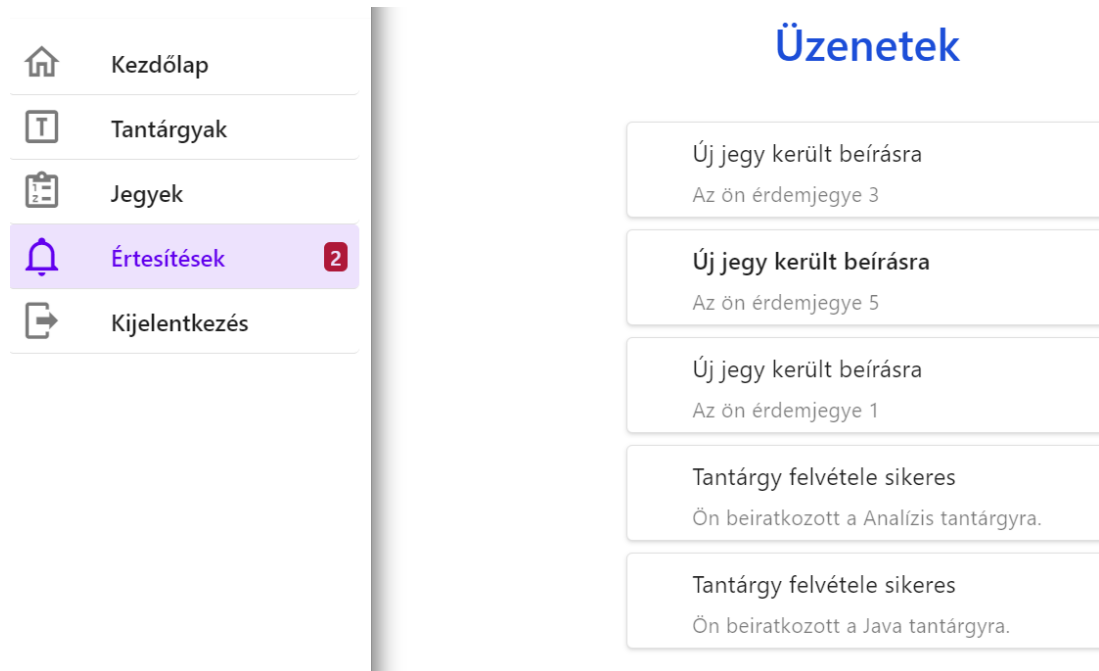
Tanulmányaim

Teljesített kredit: 12

Tantárgy neve	Kredit	Összpontszám	Érdemjegy
Programozás	3	27	4
Analízis	0	0	1
Java	5	10	5
Algoritmusok	4	6	3

16. ábra Jegyek

Az Értesítések menüpont mellett, amennyiben van olvasatlan üzenet, ezek számát a felhasználónak jelzi a rendszer. A menüpontra kattintva a felületen időrendben jelennek meg a rendszer által küldött üzenetek. A még olvasatlanok félkövér betűtípussal kerülnek kiemelésre. Az oldalon egy időben 10 értesítés kerül listázásra, a régebbi üzenetek lapozás segítségével érhetők el.



17. ábra- Értesítések

A Kijelentkezés gomb segítségével a felhasználót a rendszer kijelentkezteti és átirányítja a landoló oldalra.

3. Fejezet

Fejlesztői dokumentáció

A fejezetben bemutatásra kerülnek a program létrehozásához használt fejlesztési metódusok, keretrendszerek, adatbázisok és egyéb kiegészítő alkalmazások. Ezen felül szó lesz a fejlesztő és futtatókörnyezet kialakításáról, valamint az elkészült szoftver üzemeltetéséről és monitorozásáról.

3.1 Mikroszolgáltatások

A szoftver mikroszolgáltatás alapú architektúrát használ, ezért először ennek rövid bemutatásával kezdem a fejezetet. Megvizsgálom az előnyeit, illetve hátrányait és hogy milyen esetekben jó választás.

„A mikroszolgáltatások függetlenül telepíthető szolgáltatások, melyek egy adott üzleti területet modelleznek.” [1]

A mikroszolgáltatás architektúra a szoftvert mikroszolgáltatások gyűjteményére bontja, melyek együttesen alkotják az alkalmazást. Az ms-ek lazán kapcsoltak és ahogy az Newman idézete is írja, egy adott üzleti területet fednek le.

A monolitikus alkalmazásokkal szemben nagy előnye, hogy az egyes ms-ek fejlesztését különálló kis csapatok végezhetik szemben a sokszor több tíz-, vagy százezer soros alkalmazásokkal, ahol a fejlesztést nehezíti az azonos kódbázison dolgozó fejlesztők sokasága. Minden ms önálló alkalmazásként bármikor telepíthető, így nem szükséges kiadási ciklusokat meghatározni, illetve várni az alkalmazás többi részének elkészültére. Az ms-t fejlesztő szakemberek az adott üzleti területen mélyebb tudást szerezhetnek.

Az egyes ms-ek a követelmények megvalósításához leginkább illeszkedő technológia segítségével készülhetnek, amely egy monolitikus alkalmazásnál nem megvalósítható. A szabványos kommunikációs interfészek biztosítják a mikroszolgáltatások közötti

kapcsolatot. Például egy banki tranzakciókat kezelő ms esetén relációs adatbázisra van szükségünk, de ugyanezen alkalmazás ügyfélszerződési modulja akár dokumentum alapú adatbázist is használ. Vagy egy webes kiszolgáló esetén jó választás lehet a Java, míg a háttérben futó videók feldolgozására alkalmazott ms C++ nyelven íródhat a megfelelő sebesség elérésének érdekében.

A magasabb terheléssel rendelkező ms-ek több példányban futhatnak, - akár külön szervereken - így osztva el a terhelést. A dolgozatomban ilyen például a videókat kiszolgáló ms, mely nagy fájlokkal dolgozik. Ellenben a jegyek kiszorgálását végző ms jóval kisebb terhelésnek van kitéve.

Monolitikus alkalmazások esetén csak a teljes rendszer skálázható, jóval rosszabb határfokkal. Emiatt ezen alkalmazásoknál a vertikális skálázás – erősebb processzor, több memória hozzáadása a szerverhez – népszerűbb. Ellenben az ms-ek horizontális skálázása egyszerű – az alkalmazás több példányban fut sok, kis kapacitású szerveren.

Természetesen az architektúrának több hátránya is van. Az egy folyamaton – processzen - belül futó monolitikus alkalmazásokhoz képest a mikroszolgáltatások több folyamatból és a köztük lévő kommunikációból építik fel a funkcionalitást. Egy ms-ek közötti hívás válaszideje jóval nagyobb, mint amikor egy folyamaton belül érhető el az adott funkcionalitás. Egy monolitikus alkalmazásnál az összes adat helyben hozzáférhető, így például az összekapcsolt adatok lekérésére join művelet használható, meghívható adott műveletet elvégző metódus és sok egyéb funkció használható. Ellenben mikroszolgáltatások esetén több ms-től kell beszerezni ezeket, általában hálózaton keresztül. Az alkalmazás tesztelése is nehezebbé válik, mivel az ms-ek közötti kommunikációt, interfész réteget is tesztelni kell. A hibák kiszűrése pedig nehézkessé válik. Minden ms önálló egységet alkot, az adatokat így több adatbázis tartalmazza. Emiatt a tradicionális ACID adatbázis tranzakciók csak az adott ms-en belül működnek. Nehéz továbbá az üzleti területek megfelelő vágása. Az egyes területek határai elmosódnak, több terület használja ugyanazokat az építőelemeket.

Ezen problémák enyhítésére különböző tervezési minták léteznek, mint például a Saga minta a tranzakciók kiváltására, vagy a Domain vezérelt tervezés az üzleti modellezésre.

A mikroszolgáltatás alapú rendszerek fejlesztésének alapfeltétele a jól definiált üzleti modellek megléte. Ezért új vállalatoknál, ahol ez még kialakulóban van, illetve sokat változik, érdemes először monolitikus alkalmazásban gondolkodni és később, amennyiben az üzleti tevékenység, cégméret vagy terhelés miatt indokolt, az alkalmazás migrálható a mikroszolgáltatás alapú architektúrára. Egy ilyen rendszer fejlesztése több időbefektetést, tapasztalt fejlesztőket igényel, amely a szoftver elkészítésének költségét is növeli.

Már kiforrott üzleti modellnél, amin akár több száz fejlesztő dolgozik, a monolitikus alkalmazás fejlesztése nehézkessé válik. Amennyiben követelmény a folyamatos kiadás, skálázhatóságra van szükség, vagy a magas rendelkezésre állás fontos szempont, köszönhetően a rugalmasságának, a mikroszolgáltatás architektúra egy lehetséges megoldás.

3.2 Tudástér felépítése

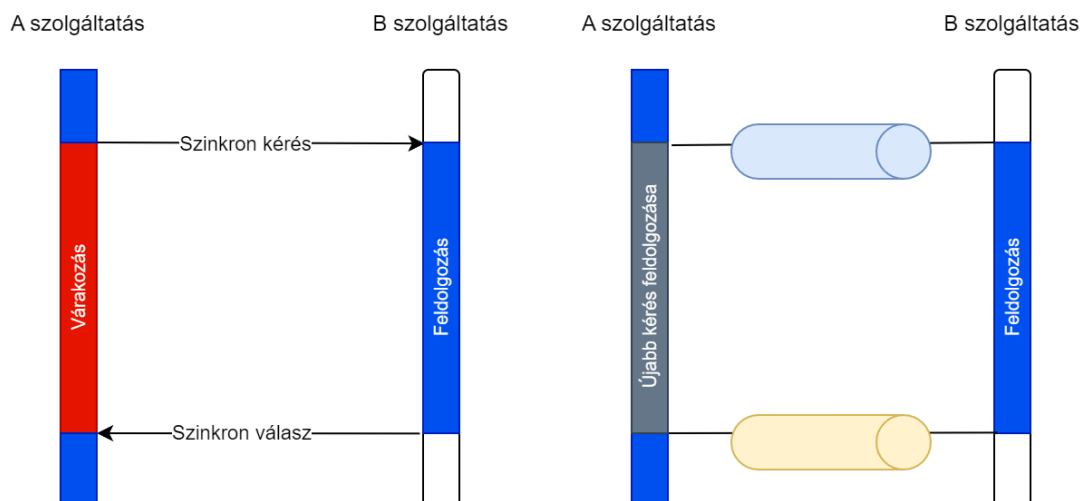
Ebben a fejezetben bemutatom a tudástér felépítését kezdve a magas szintű rendszertervekkel és az ms-ek közös kódjával, majd szolgáltatásonként kifejtem a megvalósításhoz használt módszereket. Ezután pedig a frontend tárgyalása következik.

3.2.1 Magas szintű rendszerterv

A szoftver kilenc komponensből épül fel. Ezek közül hét mikroszolgáltatás felelős az üzleti funkcionalitás megvalósításáért, a frontend a weboldal megjelenítéséért, az API Gateway pedig belépési pontként szolgál az ms-ek felé. A felhasználó az alkalmazás URL címének meglátogatásával a frontend alkalmazáson keresztül tud interakciókat végrehajtani az alkalmazással. Az adatok az ms-ekből töltődnek be, illetve a kérések kiszolgálását is ezen szolgáltatások végzik. Az ms-ekhez a kérések az API Gateway-en keresztül jutnak el, melyet az 1. ábra segítségével mutattam be.

A 19. ábra az ms-ek egymással történő kommunikációját ábrázolja, ahol az irányított egyenes az egyirányú kommunikációt jelöli, az irányítatlan pedig a kétirányút. Az kommunikáció felhasználási esettől függően lehet szinkron vagy aszinkron. Általánosan elmondható, hogy a felhasználó számára látható eredménnyel járó interakciókat szinkron kommunikációval oldottam meg, a háttérben történő feldolgozásokat pedig aszinkron módon.

Az aszinkron kommunikációnak több előnye is van. Amennyiben valamely ms nem elérhető - akár szerver, hálózati vagy alkalmazásbeli hiba miatt -, az üzenetek az üzenetsorban várokoznak, amíg az ms újra képes lesz adatok feldolgozására, ezzel növelve a rendszer hibatűrését. Az aszinkron üzenetek nem blokkolják a feldolgozó szálát, így az képes további feladatok elvégzésére, amíg a másik rendszertől válaszra vár, ezzel biztosítva a rendszer rugalmasságát. A 18. ábra ezt illusztrálja, bal oldalon a szinkron hívást, jobbra pedig az aszinkron üzenetküldést bemutatva. Nagy terhelés esetén a feldolgozó rendszerből további példányok indíthatók, melyek bekapcsolódnak az üzenetsorban található adatok feldolgozásába, így gyorsítva a rendszert. A rendszerek egymástól való függősége jelentősen csökken, ami könnyebben kezelhető, hibatűrőbb alkalmazást eredményez, mint ami szinkron kommunikáció használatával elérhető.



18. ábra Szinkron hívás és aszinkron üzenetküldés

A szinkron kommunikációt – felhasználó és gép, valamint gép és gép között is - REST interfészen keresztül valósítottam meg, mely HTTP protokollt használ. Nagy előnye az egyszerűsége és könnyű tesztelhetősége. A különböző műveleteket – olvasás, létrehozás, felülírás, törlés, vagy röviden CRUD – úgynevezett HTTP igék segítségével írja le, például a GET az adatok lekérését végzi, a POST új erőforrást hoz létre a PUT módosítja, míg a DELETE törli azt. A Tudástér a videók és képek kezelésén kívül JSON formátumú üzeneteket használ a kommunikációra mind a frontend felé, illetve az ms-ek között. A RESTful nagy hátránya, hogy lassabb számos alternatívájánál.

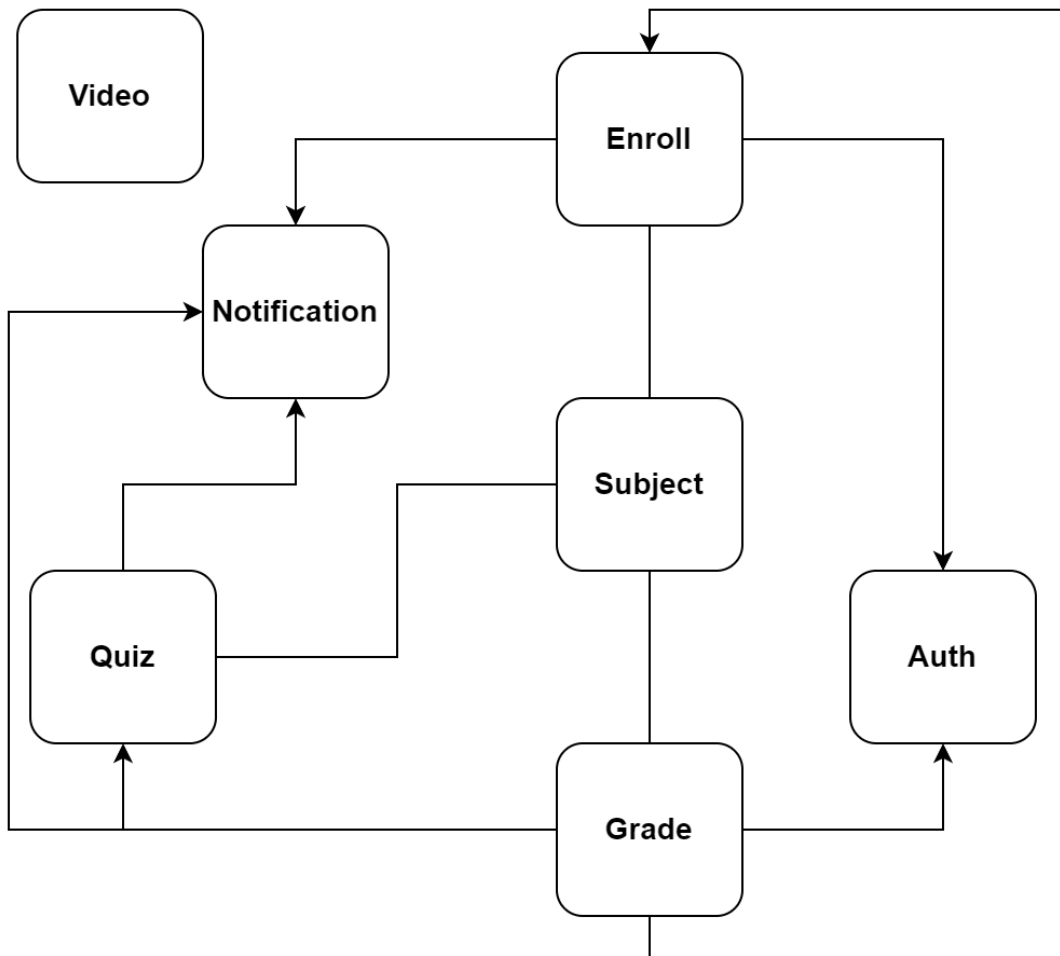
Szolgáltatások közötti kommunikációra alternatíva lehet a HTTP/2-t használó gRPC, mely bináris formátumú üzenetküldést tesz lehetővé. Előnye, hogy az üzenetformátum miatt az üzenet mérete kisebb, mint a JSON formátumú üzeneteké. Ez a protokoll gyorsabb üzenetváltást tesz lehetővé. Hátránya, hogy az üzenetek emberi szemmel nem olvashatók, így a hibák feltárása nehezebb.

Egy másik alternatíva az RSocket protokoll, mely a fent említett két kommunikációs formával szemben – melyek az OSI 7. Alkalmazási réteget használják – az OSI 4. Szállítói rétegen keresztüli üzenetküldést teszi lehetővé. Ennek köszönhetően gyorsabb a fent említett két megoldásnál. A protokoll saját keretezést használ, mely támogatja a két irányú kommunikációt a szerver és kliens között. Nem blokkoló, alkalmas adatfolyam alapú rendszerek felépítésére és visszahívások kezelésére is. Egy felépített kapcsolaton több üzenetváltás is történhet, ez egyben egyik hátránya is, hisz így a terheléelosztás megoldása nehezebb automatikusan skálázódó rendszerek esetén. További hátránya a komplexitása, illetve mivel friss megoldás, így még kiforratlan.

Nagy terhelésű rendszerek esetén a REST kommunikáció szűk keresztmetszete lehet az alkalmazásnak. Amennyiben a terhelés eléri ezt a pontot, érdemes áttérni a fent említett megoldások valamelyikére.

A frontend és backend közötti kommunikáció alternatívája lehet a GraphQL, mely erőssége, hogy egységes képet mutat a frontend alkalmazások felé, melyek kiválaszthatják, mely adatokat szeretnék megkapni a háttérrendszerektől. Segítségével kiválthatók például a BFF (Backend for Frontend tervezési minta) rendszerek, melyek adott platformra elkészített – telefon, böngésző stb. – frontend alkalmazásokat szolgálnak ki oly módon, hogy az adatokat a platform számára optimális formára hozzák. A lehetőségek közül a Tudástérhez a RESTful megvalósítást választottam, többek között az egyszerűsége, könnyű tesztelhetősége miatt. Ennek segítségével lehetőségem volt egységes kommunikációs rendszer implementálására a komponensek között, illetve a tervezett terhelés sem indokolja alternatív technológiák alkalmazását.

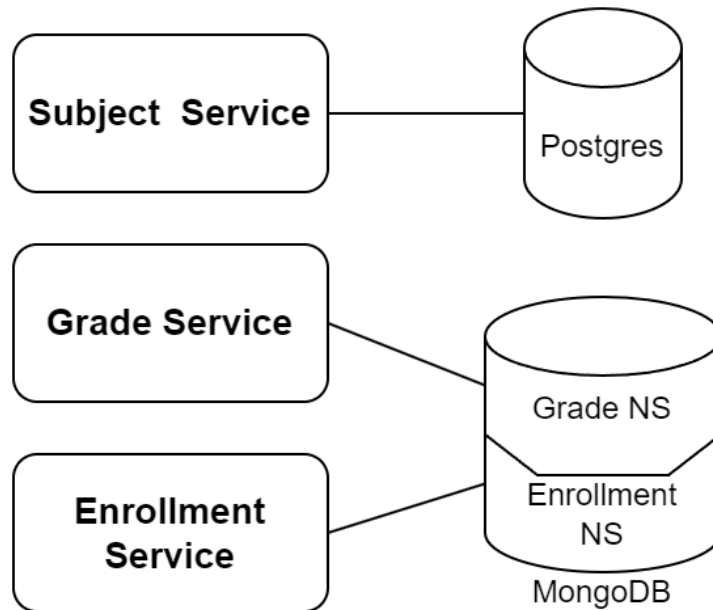
„A valódi probléma az, hogy a programozók túl sok időt töltenek el a hatékonyságon aggódva rossz helyen és rossz időben. Az idő előtti optimalizálás minden rossz forrása (vagy legalábbis a legtöbbé) a szoftverfejlesztésben” – Donald Knuth



19. ábra Magas szintű rendszerterv

Az aszinkron kommunikációt RabbitMQ üzenetsor segítségével oldottam meg, mely lehetővé teszi a pont-pont alapú üzenetváltást, valamint az üzenetszórást is. Támogatja az AMQP protokollt, mely biztosítja a biztonságos, megbízható üzenetközvetítést.

A Tudástér kétféle adatbázist használ. A legtöbb mikroszolgáltatás a dokumentum alapú MongoDB-t használja, míg a tantárgyak kiszolgálása PostgreSQL adatbázisból történik. A mikroszolgáltatás architektúra fontos eleme, hogy az ms-ek adatai egymástól elkülönítve kerüljenek tárolásra és csak az adatért felelős ms férhessen hozzá közvetlenül. Terheléstől függően használható külön adatbázis minden mikroszolgáltatáshoz, illetve kisebb terhelésnél elegendő egy adatbázis, ahol minden ms-hez külön séma tartozik. Én az utóbbit választottam az alkalmazás erőforrásigényének csökkentése érdekében.



20. ábra Adatbázisok

3.2.2 Spring Boot

Az ms-ek megvalósítására a Spring Boot [2] nyílt forráskódú, Java alapú keretrendszert használtam. A keretrendszer moduláris felépítésű és lehetővé teszi a gyors alkalmazásfejlesztést. Alapja az Inversion of Control konténer, melyet az ApplicationContext interfész deklarál. Ez biztosítja az osztálypéldányok életciklusának kezelését a létrehozástól a függőségek befecskendezésén keresztül egészen azok megszüntetéséig. A Spring az általa kezelt osztályokat Bean-eknek nevezi és annotációkat biztosít a jelölésükre. A keretrendszer ezeket a program indulásakor felolvassa és létrehozza a szükséges példányokat, majd linkeli azokat. Alapesetben Egyke példányok készülnek, de lehetőség van többféle hatáskörű Bean létrehozására, ilyen például a munkamenethez vagy kéréshez kapcsolt élettartamú egyed. A menedzselt osztályok jelölésére a @Component annotációt, illetve ennek adott réteghez tartozó változatait használja, melyet osztály szinten kell alkalmazni. Az 1. forráskód példában egy menedzselt osztályt hoztam létre, mely konstruktorában szintén a keretrendszer által kezelt osztályokat definiáltam. A keretrendszer típus alapján felismeri ezeket és automatikusan elvégzi a befecskendezést. Amennyiben valamely befecskendezni kívánt osztályból nem létezik Bean, a program hibával leáll.

```
@Component
class EnrollmmentStream(
    private val repository: EnrollmmentRepository,
    private val notifier: StudentNotifier
)
```

1. forráskód Component annotáció

A `@Bean` annotáció segítségével lehetőség van programozottan is létrehozni a menedzselt osztályokat. Ez hasznos, ha a konstruktor hívás után az osztály egyéb metódusainak hívása, esetleg beállítások megadása is szükséges. A 2. forráskód példában egy `Cache` példányt hoztam létre, mely az értékeket 5 percig tartja a memóriában.

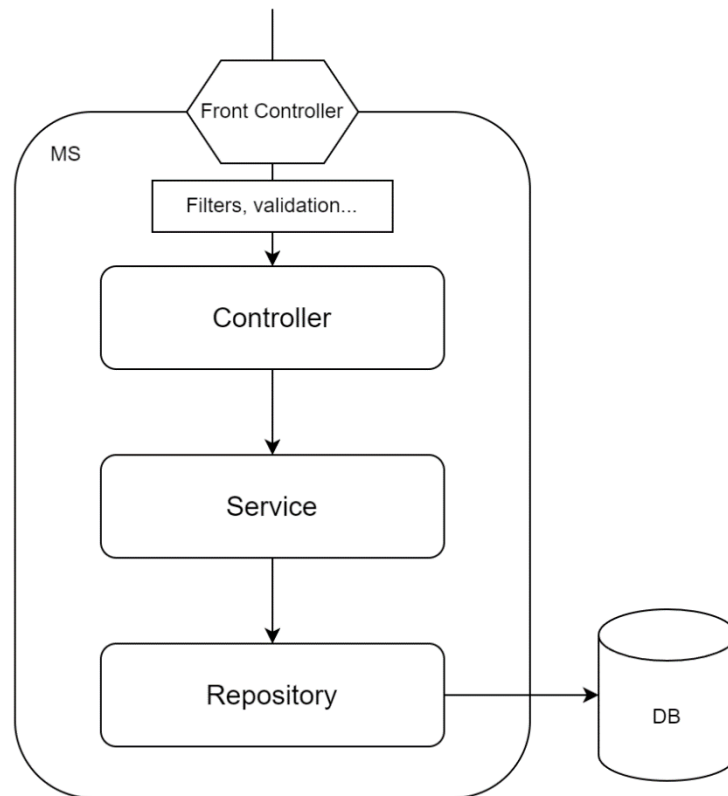
```
@Bean
fun authCache() = CaffeineCache(
    AUTH_INTEGRATION_CACHE,
    Caffeine.newBuilder()
        .expireAfterWrite(Duration.ofMinutes(5))
        .build()
)
```

2. forráskód Bean annotáció

A keretrendszer meghatározza az alkalmazás alap felépítését. Három réteget különböztet meg:

- Controller réteg. A RESTful végpontok definiálásának helye. Jelölésére a `@RestController` annotáció szolgál.
- Service réteg. Itt található az üzleti logika, adatok feldolgozása. A `@Service` annotáció segítségével hozható létre.
- Repository réteg. Az adatbázissal történő kommunikáció megvalósítására szolgál, jelölése a `@Repository` annotációval történik, mely elhagyható, amennyiben az adott osztály vagy interfész a Spring által deklarált repository interfészből származtatott.

Minden réteg csak az alatta lévő rétegeket ismeri. A Tudástér ms-ek csomagszerkezetéhez szintén ezt a három rétegű felépítést használtam, így a rétegek jól elkülönülnek. Ennek egyszerűsített változatát a 21. ábra segítségével ismertetem.



21. ábra Spring rétegek

Az alkalmazás fejlesztéséhez felhasználtam többek között a Spring Data [12] modult, mely az adatbázisok elérését, lekérdezések megfogalmazását és a tranzakciók kezelését vezérli. Ehhez adatbázis függően különböző sablon osztályokat - például JdbcTemplate, MongoTemplate - biztosít. Emellett relációs adatbázis esetén a JPA keretrendszer használatára is lehetőség van. Elérhető továbbá az úgynevezett Query by Method funkcionalitás, mely egy interfészen deklarált metódusnevekből generál lekérdezéseket. Ennek segítségével a rövidebb adatbázis kérések gyorsan megfogalmazhatók. Ehhez a keretrendszer előre meghatározott kulcsszavakat biztosít. Emellett implementálja a legalapvetőbb CRUD műveleteket is.

Az 3. forráskód példában egy tantárgyhoz tartozó órák számát kérem le.

```

interface LessonRepository : JpaRepository<Lesson, String> {
    fun countAllBySubject(subject: Subject): Long
}

```

3. forráskód Metódus alapú lekérdezés

A metódusnév feldolgozása után az alábbi SQL lekérdezés kerül végrehajtásra.

```
select count(*) from lesson l where l.subjectId= "1"
```

4. forráskód Metódusból generált SQL lekérdezés

A Spring Boot alapú alkalmazások beépített webserverral rendelkeznek, így az alkalmazásból végrehajtható fájl építhető és elindítható különálló szerver telepítése nélkül. A Tudástér beépített Tomcat szervert használ.

A keretrendszer szinte minden webfejlesztéshez szükséges esetre biztosít megoldást, ezeket az adott szolgáltatás megvalósításánál részletezem.

3.2.3 Kotlin

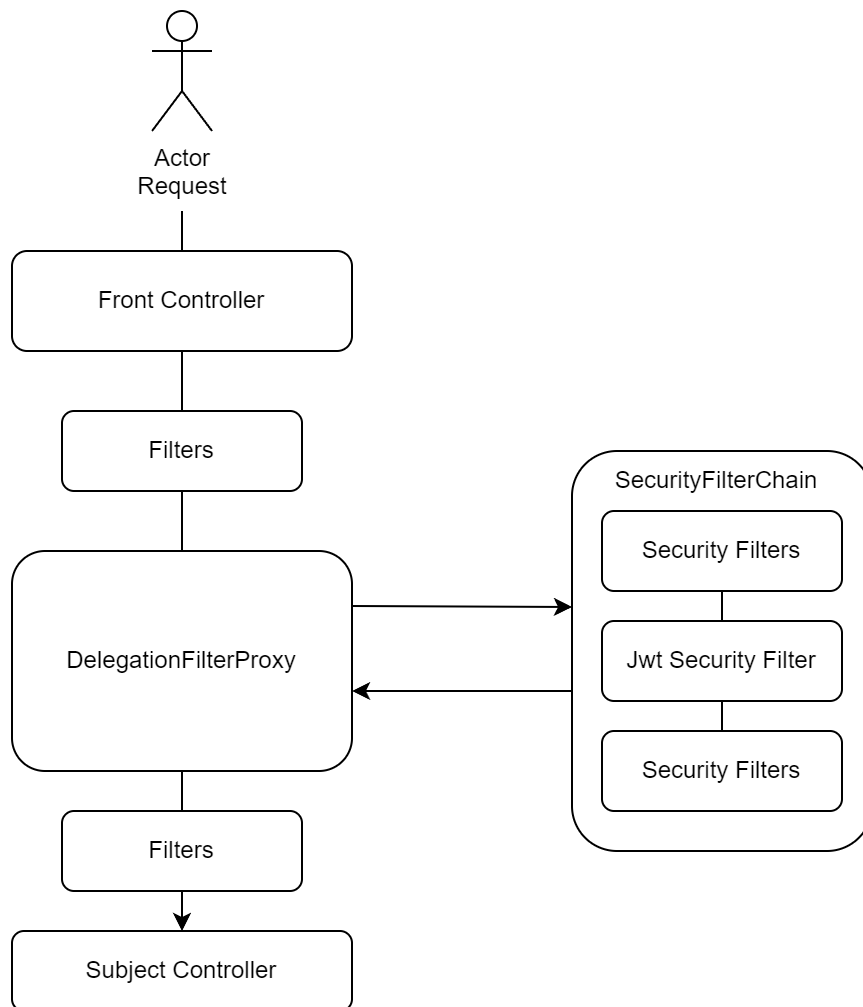
A szoftver backend részét Kotlinban [4] fejleszttem, mely egy JVM alapú, népszerű programozási nyelv. A Java nyelvvel együttműködik, könyvtárai és osztályai használhatók Kotlinban is. Az objektumorientált programozás mellett támogatja a funkcionális programozást, a null-biztos típusokat, továbbá a Java alapkönyvtárakban elérhető funkcionalitást kibővíti. Lehetőséget biztosít a típusok kiterjesztésére, operátorok felüldefiniálására és sok más funkcionalitás is helyet kapott benne.

3.2.4 Általános funkcionalitás

A ms-ek számos funkcionalitása megegyezik – például a felhasználók autorizációjának kezelése –, így a szoftver felépítésének ismertetését ezekkel a funkciókkal kezdem.

3.2.4.1 Azonosítás

Az alkalmazás használatához a felhasználónak be kell jelentkeznie a rendszerbe. Bejelentkezéskor a felhasználónév és jelszó mellé az intézmény azonosítóját – melyet az úgynevezett tenant azonosító reprezentál - is bekérem a felhasználótól. A bejelentkezési adatok ellenőrzése után egy aláírt JWT token kerül előállítására. Ezt a frontend alkalmazás minden kérés fejlécéhez csatol. A token generálás lépéseit a bejelentkezésért felelős szolgáltatásnál részletezem a 3.2.6 fejezetben.



22. ábra Security Filter

Az API Gateway ellenőrzi a token hitelességét, illetve minden ms visszaállítja a tokenből a felhasználó adatait, mely segítségével meg tudja vizsgálni a jogosultságokat az egyes funkció hívások előtt. A rendszer szál-per-kérés modellt használ, így a felhasználói adatokat az adott szálhoz kötött változóban tárolhatja a keretrendszer, amin keresztül a szükséges pontokon ez elérhető.

Ennek megvalósítására a Spring Security modult használtam. Egy ms-nél minden bejövő HTTP kérés a Front vezérlőhöz érkezik. Ez irányítja a kéréseket a megfelelő Controllerhez. A kettő között helyezkednek el a Web Filterek, illetve egy proxy, amelyen keresztül a Bean-ként definiált szűrők is meghívásra kerülnek. Többek között a Security Filter Chain, ahogy a 22. ábra ábrázolja.

```

@Bean
fun securityFilterChain(
    http: HttpSecurity
): SecurityFilterChain = http.sessionManagement {
    it.sessionCreationPolicy(
        SessionCreationPolicy.STATELESS
    )
}
    .oauth2ResourceServer {
        it.jwt().jwtAuthenticationConverter { j ->
            val converter = JwtAuthenticationConverter()
            converter.setJwtGrantedAuthoritiesConverter { jwt ->
                val c = JwtGrantedAuthoritiesConverter()
                c.setAuthorityPrefix("")
                c.setAuthoritiesClaimName(TokenField.ROLE.name)
                c.convert(jwt)
            }
            converter.convert(j)
        }
    }
    .exceptionHandling {
        it.accessDeniedHandler(BearerTokenAccessDeniedHandler())
    }
    .build()

```

5. forráskód Security filter chain

Az 5. forráskód bemutatja a SecurityFilterChain objektum példányosítását. A rendszer állapotmentes – ennek köszönhetően a felhasználó kéréseit különböző példányok szolgálhatják ki - ezért kikapcsoltam a Session létrehozását, valamint felkonfiguráltam a token validációt. A converter az Authorization fejlécben lévő JWT token ellenőrzi, feldolgozza, majd kiveszi a felhasználó szerepkörét a megfelelő mezőből, melyet a metódus szintű felhasználói felhatalmazás – azaz autorizáció - ellenőrzésére használ a program. A token aláírásának ellenőrzéséhez szükség van egy JwtDecoder példányra, amely tartalmazza az aláíráshoz használt algoritmust és a titkos kulcsot. Ehhez a már említett módon egy Bean-t hoztam létre.

```

@Bean
fun jwtDecoder(jwtProperties: JwtProperties): JwtDecoder
    = NimbusJwtDecoder.withSecretKey(
        SecretKeySpec(
            jwtProperties.secretKey
                .toByteArray(StandardCharsets.UTF_8),
            0,
            jwtProperties.secretKey.length,
            JWSAlgorithm.HS256.name
        )
    )
    .build()

```

6. forráskód JWT dekóder

A token konfigurációkat a beállításokat tartalmazó fájlból és környezeti változókból olvassa fel az ms, majd az ebből előállt objektum bemeneti paraméterként kerül átadásra a metódusnak, melyben a dekóder felparaméterezésre kerül.

Amennyiben a token lejárt, vagy a token érvénytelen, a kérés 403 Forbidden hibakóddal elutasításra kerül. Sikeres azonosítás esetén a keretrendszer elhelyezi a tokenet tartalmazó Authentication objektumot a szálhoz kötött SecurityContextHolder objektumban.

```
data class AuthenticationInfo(  
    val id: String,  
    val name: String,  
    val email: String,  
    val tenantId: String,  
    val role: Role,  
    val token: String?  
)
```

7. forráskód AuthenticationInfo osztály

Mivel az alkalmazás több pontján is hozzá kell férni a felhasználó, illetve tenant azonosítókhoz, létrehoztam egy statikus metódust, mely a tokenből összeállítja az általam létrehozott AuthenticationInfo objektumot a felhasználó adataival.

Ehhez először a context objektumot kéri el a metódus, majd amennyiben ennek típusa JwtAuthenticationToken, a deklarált enum mezők alapján elkészül az objektum.

```
fun authInfo(): AuthenticationInfo? {  
    val context = SecurityContextHolder.getContext()  
    val jwtAuth = context?.authentication as?  
        JwtAuthenticationToken  
    val token = jwtAuth?.token?.tokenValue  
  
    return if (jwtAuth != null) {  
        jwtAuth.tokenAttributes?.let {  
            return AuthenticationInfo(  
                id = it[TokenField.USER_ID.name],  
                name = it[TokenField.USER_NAME.name],  
                email = it[TokenField.EMAIL.name],  
                tenantId = it[TokenField.TENANT.name],  
                role = Role.toRole(it[TokenField.ROLE.name]),  
                token = token  
            )  
        }  
    } else (context?.authentication as?  
        SystemAuthentication)?.authInfo  
}
```

8. forráskód Felhasználói adatok lekérése

Amennyiben a token nem létezik, – kivétel rendszer szintű bejelentkezés esetén - az objektum null értékkel tér vissza. Bejelentkezve és anélkül is elérhető funkciók esetén szükség van erre. Ha a null érték tiltott, akkor az alábbi kiegészítő metódus hívásával be nem jelentkezett felhasználó esetén az ms 401 Unauthorized hibával tér vissza.

```
fun AuthenticationInfo?.orBlow() = this ?:  
throw AuthenticationCredentialsNotFoundException(  
    "Authentication not found"  
)
```

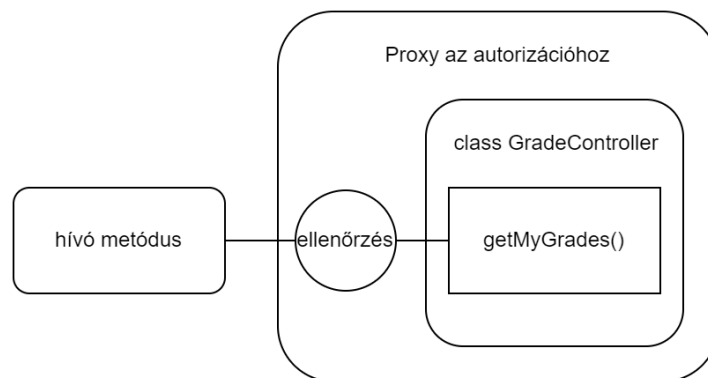
9. forráskód Szigorú bejelentkezés ellenőrzés

A keretrendszer maga is használja a felhasználói szerepköröket a metódus szintű jogosultságellenőrzésekhez. Metódusok esetén a `@PreAuthorize` annotáció segítségével adható meg, mely szerepkörök számára engedélyezett annak meghívása, ahogy az a 10. forráskód példában látható.

```
@GetMapping  
@PreAuthorize("hasRole('STUDENT')")  
fun getMyGrades()
```

10. forráskód Metódus szintű jogosultság ellenőrzés

A háttérben a keretrendszer - hasonlóan például a tranzakcióhoz vagy gyorsítótárhoz – egy proxy objektumot hoz létre az osztály köré, ahol a metódus tényleges hívása előtt végrehajtja a szükséges ellenőrzéseket.



23. ábra – Jogosultságellenőrző proxy objektum

Az adatbázis műveletekhez biztosítani kell a tenant azonosítót. Ezért a rendszer által végrehajtott műveleteket – mint például üzenetsoron érkező adatok feldolgozása, vagy ütemezett feladatok végrehajtása – becsomagoltam egy rendszerszintű jogosultságot tartalmazó funkcióba az adott tenant azonosítóval. Így a rendszer által futtatott metódusok is azonosítani tudják, mely adatokkal dolgozhatnak.

```

fun <T> withAuthContext(
    tenantId: String,
    userId: String = SYSTEM,
    body: () -> T
): T {
    SecurityContextHolder.setContext(
        SecurityContextHolder.createEmptyContext().also {
            it.authentication = SystemAuthentication(
                AuthenticationInfo(
                    userId, "System", "",
                    tenantId, Role.ROLE_SYSTEM, ""
                )
            )
        }
    )
    body()
}

```

11. forráskód Rendszer szintű azonosítás

A funkció az átadott tenant és felhasználó azonosítókkal létrehozza az Authentication objektumot és beteszi a kontextusba. Ezután meghívja az átadott metódust.

3.2.4.2 Intézmények elkülönítése

A rendszer fontos eleme, hogy egy adott iskolához tartozó adatokat csak az intézményhez tartozó felhasználók láthassák. Ennek biztosítása érdekében az adatbázis táblák tartalmazzák a korábban említett tenant azonosítót. Minden lekérdezésben szerepelni kell az erre az azonosítóra történő szűrési feltételnek. Problémát okozhat, ha beszúrásnál vagy lekérdezésnél a fejlesztő elfelejti ezt, mivel ekkor minden intézménynél megjelenik az adat. Ennek megelőzése érdekében a rendszer automatikusan hozzáilleszti a lekérdezésekhez az azonosítóra történő szűrési feltételt. Az intézményhez kötött domain modell osztályok egy TenantAware nevű ősosztályból származnak, melynek egyetlen mezője a tenant azonosító.

```

abstract class TenantAwareEntity {
    @Indexed
    lateinit var tenantId: String
}

```

12. forráskód Tenant domain modell ősosztály

Ahogy korábban említettem, a rendszer kétféle adatbázist használ, melyeknél a megoldás különböző módszerrel készült. MongoDB esetén a keretrendszer a MongoTemplate nevű osztályon keresztül kommunikál az adatbázissal, mely nem

támogatja a multitenant típusú lekérdezéseket. Az osztály csomag privát láthatóságú metódusokat használ, így egyszerű öröklődéssel nem oldható meg a metódusok felülírása. Ennek megkerülésére létrehoztam egy csomagot ugyanazon a néven, mint a `MongoTemplate` osztály csomagja és ebben került létrehozásra a `TenantAwareMongoTemplate` osztály, mely így már látja a szükséges metódusokat és hozzáadja a tenant azonosítót a lekérdezéshez, amennyiben a fent említett `TenantAware` osztályból származik.

```
if (isEntityTenantAware(entityClass, collectionName)) {
    val orgId = authInfo().orBlow().tenantId
    this.addCriteria(Criteria.where("tenantId").`is`(orgId))
}
```

13. forráskód Tenant azonosító hozzáadása a lekérdezéshez

A keretrendszer támogatja a mentés előtti és utáni callback metódusokat. Ennek segítségével a mentést is automatizáltam.

```
@Component
class BeforeSaveListener
: BeforeConvertCallback<TenantAwareEntity> {
    override fun onBeforeConvert(
        entity: TenantAwareEntity,
        collection: String
    ): TenantAwareEntity =
        entity.apply { tenantId = authInfo().orBlow().tenantId }
}
```

14. forráskód Automatikus tenant azonosító mentés

A relációs adatbázisokhoz a Hibernate JPA implementációt használja a keretrendszer, mely támogatja a multitenant lekérdezéseket. Ehhez két konfigurációs osztályt kell létrehozni, melyeket a keretrendszer automatikusan csatol a könyvtárhoz.

A 15. forráskód példában létrehoztam a `TenantIdentifierResolver` alosztályt, ahol a felülírt `resolve` metódusban a korábban bemutatott `authInfo` segédfüggvényt használtam a tenant azonosító lekérdezésére. Ezután ezt az osztályt csatoltam a Hibernate keretrendszerhez a `JpaPropertyCustomizer` osztály `customize` metódusában, mely így a HTTP fejlécből kinyert bejelentkezési információból automatikusan alkalmazza az intézmény azonosítót minden lekérdezés és beszúrás műveleten.

```

@Component
open class JpaPropertyCustomizer(
    private val tenantIdResolver: TenantIdentifierResolver
) : HibernatePropertiesCustomizer {
    override fun customize(
        hibernateProperties: MutableMap<String, Any>
    ) {
        hibernateProperties[AvailableSettings.
            MULTI_TENANT_IDENTIFIER_RESOLVER] = tenantIdResolver
    }
}

@Component
open class TenantIdentifierResolver
: CurrentTenantIdentifierResolver {

    override fun resolveCurrentTenantIdentifier(): String
    = authInfo()?.tenantId ?: "base"

    override fun validateExistingCurrentSessions(): Boolean
    = false
}

```

15. forráskód Hibernate tenant azonosító kezelés

3.2.4.3 Üzenetek lokalizációja

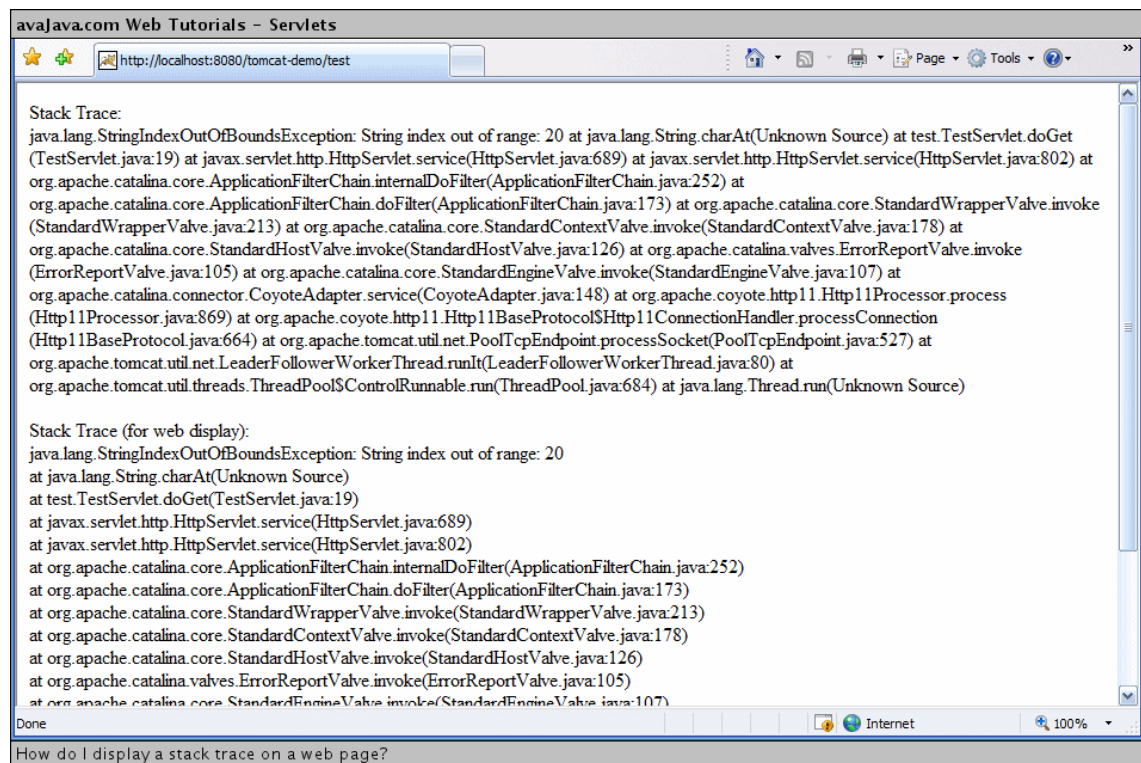
A rendszer jelenleg csak a magyar nyelvet támogatja, de a felhasználó számára küldött validációs és egyéb üzeneteket a forráskódban tárolni rossz tervezési minta. Emiatt a program nyelvi fájlokban tárolja a szöveges üzeneteket. A fájl neve kötelezően a message_ <ország kód> sémát követi. A rendszer jelenleg fixen - de igény szerint a HTTP fejlécből előállítva – megkapja a szükséges nyelvet, mely alapján a megfelelő nyelvi fájlból a kulcs alapján lekérdezi a szükséges szöveget.

3.2.4.4 Általános hibakezelés

Sok rendszernél jelent problémát, hogy nem kezelt kivételek esetén a teljes műveleti verem tartalma elküldésre kerül a kliensnek. Amellett, hogy a felhasználói élményt rontja, akár komoly biztonságtechnikai problémákat is előidézhet. Egy ilyen hibaüzenet információkat tartalmazhat a rendszer felépítéséről, szerver verzióról, adatbázis táblák szerkezetéről, illetve akár érzékeny adat is kikerülhet a rendszerből. A 24. ábra egy ilyen hibaüzenetet ábrázol.

Ennek elkerülése érdekében központi hibakezelést vezettem be, mely a Controllerek köré készít proxy objektumot. Ez elkapja az ms által kiváltott kivételeket majd megfelelő

formátumra alakítja a kliens számára. A hibákat több csoportba soroltam. Amennyiben egyik hibacsoport sem illik az adott kivételre, úgy az általános hibakezelő ad választ. A hibaüzenet megfelel az RFC-7807 szabványban [10] definiált formátumnak. A hibakezelés megvalósítása az Aspects osztályban található. Az itt deklarált RestExceptionHandler osztály a @ControllerAdvice annotációval jelölt, ez hozza létre az említett proxyt az összes Controller köré.



24. ábra Stacktrace a böngészőben

A validációs hibák a Spring BindException osztályának leszármazottjai. A 16. forráskód ezt a kivételtípust és alosztályait kezeli. Amennyiben valamely validációs feltétel nem teljesül, az ms egy kivételt vált ki. Az @ExceptionHandler annotációban megjelölt kivételt a handler metódus elkapja. Ezután a szabványnak megfelelő ProblemDetail típusú objektumot állít elő belőle. A válaszbjektum összeállításánál beállítottam a hiba kódját, fejlécét és részleteit, valamint az adott mikroszolgáltatás elérési útvonalát. Ezt követően a konkrét hibák feldolgozásánál a hibásan kitöltött mező nevét és a hiba típusát az invalid_fields mezőbe gyűjtöttem össze.

```

@ExceptionHandler(BindException::class)
fun handleValidationError(error: BindException)
: ResponseEntity<ProblemDetail> {
    val problemDetail = ProblemDetail.forStatus(
        HttpStatus.BAD_REQUEST
    )
    problemDetail.title = getMessage(VALIDATION_ERROR_TITLE)
    problemDetail.detail = getMessage(VALIDATION_ERROR_DETAIL)
    problemDetail.type = URI("/auth")
    val errors =
        error.bindingResult.allErrors.associate {
            val message = it.defaultMessage ?
                : getMessage(VALIDATION_DEFAULT_MESSAGE)

            val field =
                if (it.contains(ConstraintViolation::class.java)) {
                    val cv = it.unwrap(ConstraintViolation::class.java)
                    val f = cv.propertyPath.toString()
                    f.ifBlank { it.code }
                } else {
                    it.code
                } ?: "Unknown field"
            Pair(field, message)
        }
    problemDetail.setProperty("invalid_fields", errors)

    return ResponseEntity.badRequest().body(problemDetail)
}

```

16. forráskód Validációs hibák kezelése

Hibás kérés esetén a kliens a 17. forráskód példában látható JSON mintaválasz formátumban kapja meg a hibaüzenetet.

```

{
    "type": "/quiz",
    "title": "A kérés ellenőrzése sikertelen!",
    "status": 400,
    "detail": "Hibás kérés.",
    "instance": "/api/v1",
    "invalid_fields": {
        "name": "A cím nem lehet üres.",
        "questions[0]":
            "Minden kérdéshez kötelező a helyes válasz megjelölése.",
        "questions[0].points":
            "A pontszám csak pozitív egész szám lehet."
    }
}

```

17. forráskód Validációs hiba válasz

Ez tartalmazza a szolgáltatás és a végpont URI-ját, valamint a fent említett módon a validációs hibákat. Természetesen a frontend az űrlapok beküldés előtt is elvégzi ezeknek az ellenőrzését, így normál felhasználás mellett ezzel a hibaüzenettel nem találkozhat a felhasználó.

Az alkalmazásban az üzleti logikai hibák kiváltására egy saját hibatípust definiáltam, mely a 18. forráskód példában látható. Ez a program bármely pontjáról dobható. Tartalmazza a hibához tartozó üzenet kódját, a nyelvesített üzenetbe történő behelyettesítéshez szükséges változókat, valamint a visszaadni kívánt HTTP státuszkódot, mely alapesetben 400 Bad Request.

```
open class ApiException(  
    val msg: Messages,  
    val status: HttpStatus = HttpStatus.BAD_REQUEST,  
    vararg val params: String,  
    throwable: Throwable? = null  
) : RuntimeException(throwable)
```

18. forráskód Egyedi kivétel üzleti hibák kiváltásához

Ennek kezelését szintén a RestExceptionHandler osztály végzi egy újabb handler metódusban, ahol a megjelölt kivételtípus az ApiException. Végül bármely más, nem kezelt hiba feldolgozását az általános hibakezelő végzi, mely 500 Internal Server Error kóddal tér vissza a klienshez a már fent említett üzenet formátumot használva.

3.2.4.5 Gyorsítótár

A szolgáltatások adatokat kérnek egymástól a kérések kiszolgálásához. Ezek közül az adatok közül van, amely gyorsan változik, de olyan is, amely ritkán, vagy soha – mint például a felhasználó neve. Ezeket az adatokat a rendszerek közötti kommunikáció mennyiségének csökkentése és ezáltal a teljesítmény növelésének érdekében lokális gyorsítótárban tárolja a rendszer. Ebben segít a Spring Cache absztrakciója, mely lehetővé teszi a gyorsítótárak annotációkkal történő vezérlését. A konkrét megvalósításhoz a Caffeine memória alapú gyorsítótár megoldást választottam. Ez példányonként a lokális memóriában tárolja az adatokat. Amennyiben sok példány fut az adott ms-ből vagy nagy méretű adatot kell gyorsítótározni, érdemes elosztott gyorsítótár megoldást alkalmazni, mint például Redis-t, melyet a példányok közösen használnak.

A 19. forráskódban az Enrollment Service által megvalósított felhasználó lekérdezése funkció példáján keresztül mutatom be ezt. A felhasználók tárolására létrehoztam egy gyorsítótár példányt a keretrendszer számára. A rekordok lejáratí ideje 5 perc az adat memóriába írását követően.

```
@Bean
fun authCache() = CaffeineCache(
    AUTH_INTEGRATION_CACHE,
    Caffeine.newBuilder()
        .expireAfterWrite(Duration.ofMinutes(5))
        .build()
)
```

19. forráskód Gyorsítótár példány létrehozása

A gyorsítótárazni kívánt adat egy metódus visszatérési értéke, ezt jelezni kell a keretrendszernek. Erre szolgál a @Cacheable annotáció, mely bemeneti paraméterei a gyorsítótár neve, valamint egy kulcs. Amennyiben ugyanezzel a kulccsal érkezik a lejáratí időn belül újabb kérés, a program azt a gyorsítótárból szolgálja ki és nem hívja meg a domaint kezelő ms végpontját. A 20. forráskód a tárgyat felvett felhasználókat gyorsítótárazza, mely a tanári adatlapon megjelenik.

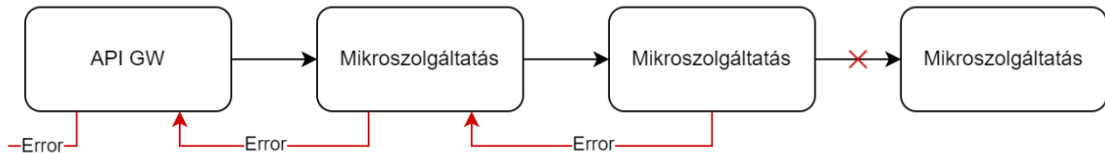
```
@Cacheable(value = [AUTH_INTEGRATION_CACHE], key = "#userIds")
fun getUsers(userIds: List<String>): List<User>{
    [auth service végpontjának hívása]
}
```

20. forráskód Felhasználók gyorsítótárazása

3.2.4.6 Szolgáltatások közötti hibakezelés

A mikroszolgáltatások közötti szinkron hívások problémákat okozhatnak. Amennyiben egyik ms nem elérhető vagy hiba történik, az őt hívó ms-ben ez kivételt vált ki. Ha több ms láncoltan hívja egymást, a kiváltott hiba egészen a felhasználóig eljut, ahogy a 25. ábra illusztrálja. Ezt a jelenséget gyűrűző hibának nevezik. Egy ilyen hiba miatt az egész rendszer elérhetetlenné válhat. Ennek megakadályozására került a rendszerbe a megszakító – eredeti nevén Circuit Breaker vagy röviden CB. Normális esetben a CB zárt, így minden kérés eljut a hívott ms-hez, mely kiszolgálja azt. Amennyiben a hívott rendszer adott határmennyiséget meghaladó százalékban hibával tér vissza vagy nem elérhető, a CB kinyílik. Ebben az állapotban az előre definiált fallback metódus eredményével tér vissza a megjelölt API hívás, mely segítségével hiba esetén egy alap funkcionalitást tud biztosítani a rendszer. A CB a beállított időintervallum után félig nyílt

állapotba áll, ilyenkor adott százalékban átengedi a kéréseket. Ha ezek a kérések hibamentesen kiszolgálásra kerülnek, akkor visszakerül zárt állapotba, mely a normál működés. Az implementációhoz a Resilience4J [11] megvalósítását használtam.



25. ábra Gyűrűző hiba

Az első lépés a CB felkonfigurálása. Beállítottam a csúszóablak típusát idő alapúra, a kérések kiszolgálásának időlimitjét egy másodpercre. Megadtam továbbá az ablak méretét, így a határértékek számításához mindig az utolsó 50 másodpercben történt hívások eredményét használja a könyvtár. Nyílt állapotban a CB 10 másodpercet vár, amíg félig nyíltba vált. Ez időt biztosít a hívott félnek, hogy a hiba elháruljon – például új példány induljon, vagy magas terhelés esetén elvezesse azt. Nem csak hiba esetén, hanem amennyiben a kérések 95 százalékának kiszolgálása fél másodpercnél lassabb, a CB kinyílik. Ez segít a hívott fél túlterhelésének elkerülésében. A 21. forráskód ezt a konfigurációt mutatja be, mely a ResilienceConfig osztályban található.

```
CircuitBreakerConfig.custom()
    .slidingWindowType(
        CircuitBreakerConfig.SlidingWindowType.TIME_BASED
    )
    .slidingWindowSize(50)
    .minimumNumberOfCalls(50)
    .failureRateThreshold(80f)
    .waitDurationInOpenState(Duration.ofSeconds(10))
    .slowCallDurationThreshold(Duration.ofMillis(500))
    .slowCallRateThreshold(95f)
    .build()
```

21. forráskód CB konfiguráció

A beállításokat követően a gyorsítótárhoz hasonlóan a végpont hívást végző metóduson annotáció segítségével engedélyeztem a CB-t. Itt állítható be a fallback metódus és a használni kívánt CB neve.

```

@CircuitBreaker(name = "auth", fallbackMethod =
"getUsersFallback")
fun getUsers(userIds: List<String>): List<User> {
    return authApi.getUsers(userIds)
}

fun getUsersFallback(userIds: List<String>, e: Exception)
: List<User> {
    logger.error { "CircuitBreaker getUsersFallback was called"
}
    return emptyList()
}

```

22. forráskód CB és fallback metódus

A fallback metódus szignatúrájának egyeznie kell az eredeti metódussal, kiegészítve azt a kivétel objektummal. A példában a getUsers metódus hívás bármely kivétel esetén a hibát a log kimenetre írja, majd egy üres listát ad vissza. A rendszer így nem hibával tér vissza, hanem kiszolgálja az ms-hez érkező kéréseket. Például a tanár lekérheti a tantárgyait és szerkeszteni is tudja őket, de a tantárgyat felvett diákok nem jelennek meg az oldalon. Ez a megoldás felhasználóbarát egy teljes leálláshoz képest.

Ha egy ms-hez túl sok hívás érkezik, a példány összeomolhat – például elfogy a memória. A terhelések csökkentésére a BulkHead és RateLimiter tervezési mintákat használtam, melyet szintén a Resilience4J csomag biztosít. Az előbbi maximalizálja az egy időben párhuzamosan történő hívások számát, míg az utóbbi adott időintervallum alatti kérések összességének számát maximalizálja.

3.2.4.7 OpenAPI dokumentáció

A ms-ek által kiajánlott végpontok OpenAPI specifikáció szerint dokumentáltak (OpenAPI, 2023). A dokumentációkhoz két végpont készült.

1. /api/[szolgáltatás neve]/v3/api-docs
2. /api/[szolgáltatás neve]/swagger-ui/index.html

Az 1. végpont JSON formátumban adja vissza a dokumentációt. Ez felhasználható például kliensek automatikus generálására. A 2. végponton grafikus felületen is elérhető a dokumentáció, melyet a 26. ábra ismertet.

GET

/api/v1/teacher/{id}

PUT

/api/v1/teacher/{id}

Parameters

Name	Description
<div>id * required</div> <div>string</div> <div>(path)</div>	<div>id</div>

Request body required

Example Value

Schema

```

{
  "name": "string",
  "description": "string",
  "credit": 0,
  "seats": 0,
  "startDt": "2023-03-27T16:58:32.518Z",
  "endDt": "2023-03-27T16:58:32.518Z",
  "coverImagePath": "string"
}

```

26. ábra Swagger UI

3.2.4.8 Modell nézetek

Az adatbázis modellek tartalmaznak érzékeny, illetve a rendszer számára fenntartott meta adatokat. Ezek kiszolgálása más rendszerek és felhasználók felé biztonsági szempontból aggályos. Ennek elkerülésére többféle megoldás létezik. Az adatbázis objektumokhoz létrehozható szűkített információt tartalmazó - DTO - osztály a kliensek felé, mely csak a szükséges mezőket tartalmazza. Ez a megoldás flexibilis, mivel az adatmodell változásai nincsenek hatással a végponton visszaadott modell formátumára. Hátránya, hogy egy plusz réteget hoz a rendszerbe – az objektumok közötti adatmásolást. Sok esetben a végpont és az adatmodell struktúrája megegyezik. Ilyen esetekben használhatók az úgynevezett objektum nézetek, ahol akár jogkörökhöz is köthetjük az egyes változók megjelenését a válaszban. Ehhez létrehoztam egy osztályt, mely az objektum JSON formátumra történő alakításába épül be, és csak azokat a mezőket engedi az eredménybe másolni, ahol az látható a bejelentkezett felhasználó jogkörével.

```

@RestControllerAdvice
class ViewFromRole : AbstractMappingJacksonResponseBodyAdvice()
{
    override fun beforeBodyWriteInternal(...) {
        val auth = authInfo()
        val role = auth.role
        bodyContainer.serializationView = View.mapping[role]
    }
}

```

23. forráskód Jogkör alapú nézet

A 23. forráskódban látható, hogy sorosítás előtt a nézet osztályok közül a rendszer kiválasztja a megfelelőt jogosultság alapján és beállítja ezt a sorosítónak. A nézetek hierarchikusan épülnek egymásra, a diákok mindent láthatnak, ami az alap nézetben van, a tanárok pedig mindent, ami a diákban. A nézeteket szerepkörhöz kötve egy Egyke objektum hasítótábla mezőjében tároltam, ahogy a 24. forráskód példában látható.

```

sealed class BasicView
open class StudentView : BasicView()
class TeacherView : StudentView()
object View {
    val mapping = mapOf(
        Role.ROLE_STUDENT to StudentView::class.java,
        Role.ROLE_TEACHER to TeacherView::class.java,
        "UNAUTHORIZED" to BasicView::class.java,
    )
}

```

24. forráskód Nézet osztályok

Ezután az adatbázis modell mezőit megjelöltem azok láthatóságával. A 25. forráskód példában a tantárgy nevét a diákok és az öröklődés miatt a tanárok is láthatják, de az archivált státuszt csak a tanárok.

```

@JsonView(StudentView::class)
var name: String? = null

@JsonView(TeacherView::class)
var archive: Boolean = false

```

25. forráskód Nézetek alkalmazása

3.2.4.9 Adatmodell meta adatok

Az domain modellek auditálásához és a tranzakciók konfliktuskezeléséhez meta adatokat tartalmaznak. Ezeket a rendszer automatikusan frissíti, mielőtt az objektum adatbázisba írása megtörténik.

- createdAt – az adatbázis bejegyzés létrejöttének időbélyege
- modifiedAt – az utolsó módosítás dátuma
- createdBy – a bejegyzést létrehozó felhasználó azonosítója
- modifiedBy – a bejegyzést utoljára módosító felhasználó azonosítója
- version – módosítások száma.

A rendszer optimista zárolási technikát használ ugyanazon rekord adatbázisba történő írása során. Ez a version mező alapján történik. Amikor a rendszer egy rekordot kiolvas és módosít, a visszaírás előtt ezt az értéket megnöveli. Amennyiben az új érték kisebb vagy egyenlő, mint az adatbázisban a mező jelenlegi értéke, az jelzi, hogy más szál időközben módosította a bejegyzést. Ilyenkor a rendszer kivételt vált ki elkerülve a módosított bejegyzés felülírását.

3.2.4.10 Adatbázis migráció

A mikroszolgáltatások az adatbázis felépítéséhez, módosításához, teszt adatok beszúrásához adatmigrációs scripteket használnak. Ennek nagy előny, hogy a programhoz hasonlóan az adatbázis séma is bekerül a verziókezelőbe és a program minden verziójához a megfelelő séma elérhetővé válik. A PostgreSQL-t használó Subject Service a relációs adatbázisokhoz készült Liquibase-t, míg a MongoDB alapú ms-ek a Mongock nevű migrációs eszközt használják. A migráció az alkalmazás indításakor történik automatikusan, melyről a migrációs eszköz az adatbázisban nyilvántartást vezet, így csak azok a scriptek kerülnek végrehajtásra, melyek még nem kerültek be oda.

3.2.5 Api Gateway

Az Api Gateway a rendszer központi belépési pontja, mely egységes képet mutat a frontend alkalmazásnak a háttéralkalmazásokról. A Kubernetesre [3] telepített alkalmazás esetében csak az Api Gateway végpontja elérhető a klaszteren kívülről. Kettős szerepet tölt be. Terminálja a védett végpontokra érvényes token nélkül érkező kéréseket. A konfigurációs fájl tartalmazza a publikusan elérhető végpontok listáját,

melyet az alkalmazás induláskor felolvas és tárol. Beérkező kérésnél megvizsgálja, hogy az aktuális kérés URL-je megtalálható-e a listában, amennyiben igen, a kérést tovább engedi az ms-ek felé, egyébként megvizsgálja a tokenet és érvénytelen esetén visszautasítja a kérést 401 Unauthorized hibakóddal. A vizsgálathoz egy webfiltert és egy lambda kifejezést hoztam létre.

```
val secured = { request: ServerHttpRequest ->
    publicRoutes.urls.none { request.uri.path.contains(it) }
}
```

26. forráskód Publikus végpontot vizsgáló lambda kifejezés

A kifejezés bemenete a HTTP kérés. Ebből kértem le a REST kérés URL útvonalát. Amennyiben ez egyik publikus URL-re sem illeszkedik, a végpont védett. A visszatérési érték boolean, mely igaz, ha a végpont védett. Ezt a lambda kifejezést használja fel a webfilter annak eldöntésére, meg kell-e vizsgálni a JWT tokenet.

Igaz érték esetén a kérés fejlécéből kiveszem az Authorization mező értékét és levágom az első 6 karaktert, mely a token típusa – a Tudástér esetében Bearer. Ezután az aláírás és lejárat dátum szerinti ellenőrzés következik, majd az eredmény alapján kerül a kérés továbbításra vagy visszautasításra.

```
override fun filter(
    exchange: ServerWebExchange,
    chain: GatewayFilterChain
): Mono<Void> {
    if(secured(exchange.request)) {
        val authToken =
            exchange.request.headers[HttpHeaders.AUTHORIZATION]?
                .first() ?: return unauthorized(exchange)

        val token = authToken.substring(7)

        if (!tokenManager.validate(token)) {
            return unauthorized(exchange)
        }
    }
    return chain.filter(exchange)
}
```

27. forráskód Token ellenőrzés

Az API Gateway másik feladata a kérések eljuttatása a megfelelő ms-hez. Minden kérés a frontend felől tartalmazza a szolgáltatás nevét. Például /subject/api/v1/1 erőforrás az 1-es azonosítóval rendelkező tantárgy erőforrást azonosítja. Az Api Gateway elemzi a kérés URL-jét és a beállított szabályoknak megfelelő szolgáltatás felé irányítja azt,

eltávolítva a kérésből a szolgáltatás nevét. Az alábbi beállítás a /subject kezdetű kéréseket továbbítja az URI mezőben meghatározott címre levágva a /subject előtagot. Az URI egy Kubernetes terheléelosztó címe, melyről a 3.4.2 fejezetben írok részletesen.

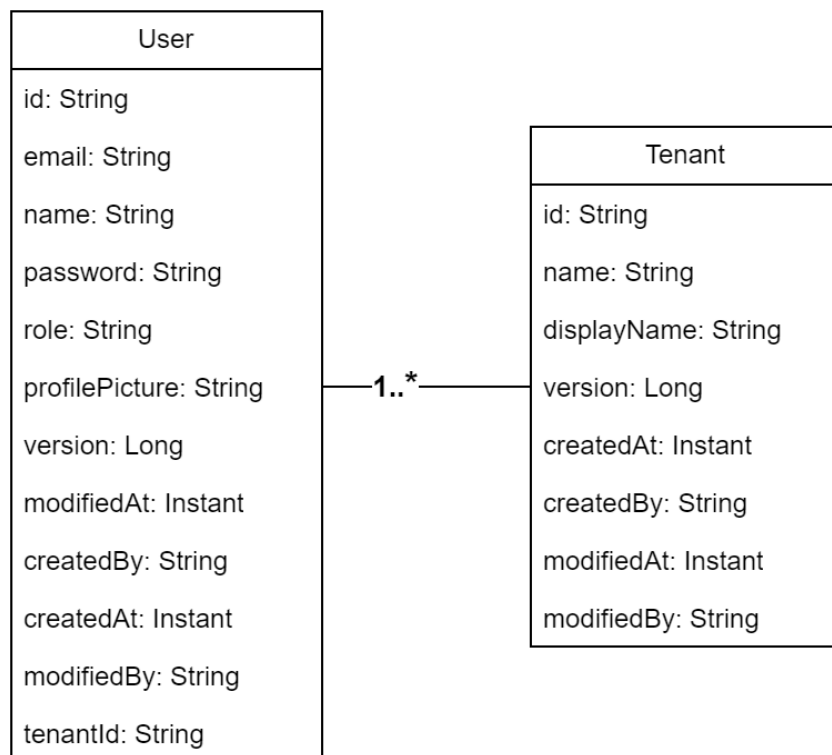
```
- id: subject-route
  uri: http://subject-service-url:8080
  predicates:
    - Path=/subject/**
  filters:
    - RewritePath=/subject/(?<segment>.*), /$\{segment}
```

28. forráskód Kérés átirányítása

3.2.6 Felhasználók kezelése

A rendszerbe új felhasználó kétféleképpen kerülhet. Az egyik az intézmény regisztrálásakor létrejövő adminisztrátor. Ezután az adminisztrátor vehet fel különböző jogkörrel új felhasználókat ehhez az intézményhez.

Az Auth Service két domain modellt kezel, az intézményeket és a felhasználókat.



27. ábra Felhasználók és Intézmények modell

A regisztrációs űrlap kitöltése és elküldése után az adatok validációja következik. A keretrendszer biztosítja az ehhez szükséges annotációkat, azonban ezek csak egy-egy mező kitöltöttségét, hosszát tudják vizsgálni. A jelszó validáció esetén szükségem volt a

jelszó és jelszómegegyezés összevetésére. Ehhez készítettem egy saját annotációt, amelyet osztály szinten lehet alkalmazni.

```
@Constraint(validatedBy = [PasswordMatchValidator::class])
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class PasswordMatch(
    val message: String = "{validation.password.not.match}",
    val groups: Array<KClass<*>> = [],
    val payload: Array<KClass<out Payload>> = [],
)
```

29. forráskód Jelszó ellenőrző annotáció

A hozzá kapcsolt validátor osztály megvizsgálja a jelszavakat. Ez beépül a keretrendszer validációs fázisába, mely a Controller metódus meghívása előtt fut le.

A keretrendszer feldolgozza az annotációt és meghívja az annotációban beállított osztály `isValid` metódusát, melynek átadja a kérés payload objektumát. Itt történik a tényleges vizsgálat. A metódus igaz értéket ad vissza, ha a jelszó nem null érték és a két jelszó megegyezik.

```
class PasswordMatchValidator :
    ConstraintValidator<PasswordMatch, RegisterRequest> {
    override fun isValid(
        value: RegisterRequest?,
        context: ConstraintValidatorContext?
    ): Boolean {
        return value != null &&
            value.password == value.confirmPassword
    }
}
```

30. forráskód Jelszavak összevetése

A jelszó helyességét szintén vizsgálja a rendszer. Az egyszerűség kedvéért csak a jelszó hosszára adtam megkötést, melynek legalább 8 karakternek kell lennie.

Amennyiben sikeres az ellenőrzés, megvizsgálom, hogy az intézmény regisztrálva lett-e korábban. Ha igen, akkor hibaüzenettel tér vissza a rendszer. Amennyiben még nem létezik, létrehozásra kerül az adatbázisban. Ezt követően a felhasználó tárolása következik. Biztonsági okokból a rendszer nem tárolja a felhasználó jelszavát, csak az ebből készült hash értéket. Ehhez az Argon2 hasító algoritmust használtam az OWASP [9] ajánlás szerint. Végül a felhasználót sikeres regisztráció után a bejelentkezési oldalra irányítja a rendszer, ahol automatikusan kitöltésre kerül az intézmény azonosító.

A bejelentkezésnél a felhasználó email címe és jelszava mellett az intézmény azonosítóját is meg kell adnia. A rendszer először ellenőrzi, az intézmény létezik-e és hozzá megtalálható-e a felhasználói fiók. Bármelyik hiányzik, a bejelentkezés elutasításra kerül. Siker esetén a jelszó ellenőrzése következik. Mivel a rendszer csak a jelszó hash-ét tartalmazza, így a bejelentkezéshez beírt jelszót is hash-elni kell, majd a két hash-t összehasonlítva dönthető el, a felhasználó jó jelszót írt-e be. Ha sikeres a vizsgálat, a felhasználó egy JWT tokenet kap, melyet a frontend minden kéréshez csatol. A tokenbe bekerülnek a felhasználó főbb adatai, valamint a token lejárat dátuma, ahogy a 31. forráskód példában látható. Így adatbázis kérés nélkül visszaállítható a felhasználó bármely szolgáltatásban. A token aláírása egy titkos kulccsal történik. Ennek biztonságos tárolásáról a rendszert üzemeltetőnek gondoskodnia kell.

```
key = Keys.hmacShaKeyFor(secret_key)

fun generateToken(user: User): String {
    val expiry = jwtProperties.validityInMinutes.toLong()
    return Jwts.builder()
        .setClaims(
            mapOf(
                TokenField.ROLE.name to user.role,
                TokenField.TENANT.name to user.tenantId,
                TokenField.USER_ID.name to user.id
            )
        )
        .setSubject(user.email)
        .setIssuedAt(Date())
        .setExpiration(Date(Date().time + (expiry * 60 * 1000)))
        .signWith(key)
        .compact()
}
```

31. forráskód JWT token generálás

Az adminisztrátorok a felhasználókat törölhetik, illetve módosíthatják az adataikat. A szolgáltatás ezen kívül lehetőséget biztosít a felhasználó adatainak lekérésére, a profilkép feltöltésére és a jelszó megváltoztatására is. Jelszóváltoztatás esetén először ellenőrzöm a meglévő jelszó helyességét, majd a regisztrációnál leírt módon az új jelszó mentésre kerül, amennyiben helyes, ahogy a 32. forráskód példában látható.

```

fun updatePassword(passwordChangeRequest: PasswordChangeRequest)
{
    val user = repository.findById(authInfo().orBlow().id).get()
    val isCurrentPasswordValid = passwordEncoder.matches(
        passwordChangeRequest.currentPassword
        ,
        user.password
    )

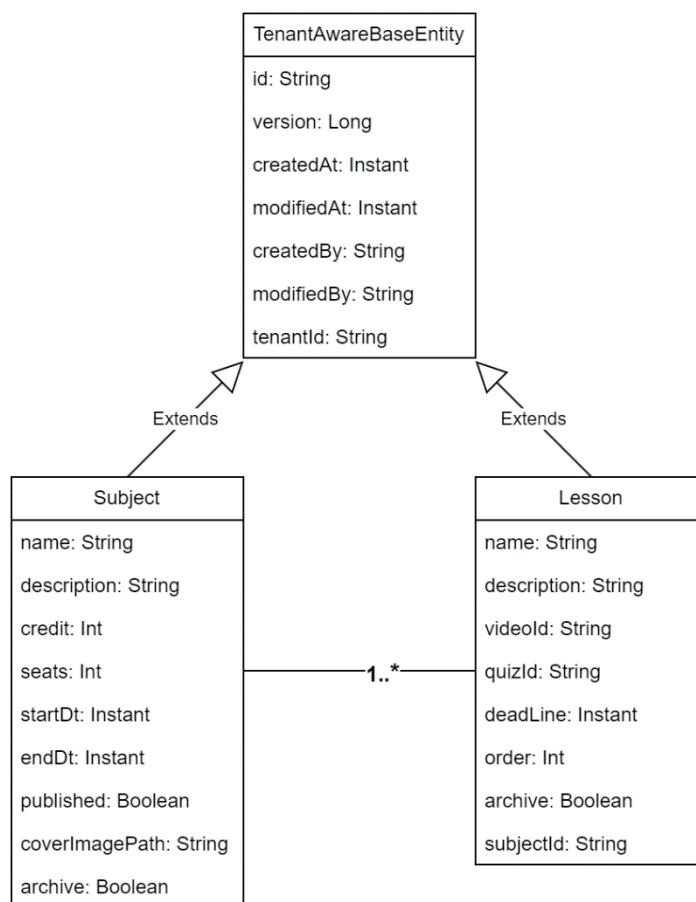
    if (!isCurrentPasswordValid) {
        throw ApiException(INVALID_CURRENT_PASSWORD)
    }
    val hash = passwordEncoder.encode(
        passwordChangeRequest.password
    )

    repository.save(user.copy(password = hash))
}

```

32. forráskód Jelszó módosítása

3.2.7 Tantárgy szolgáltatás

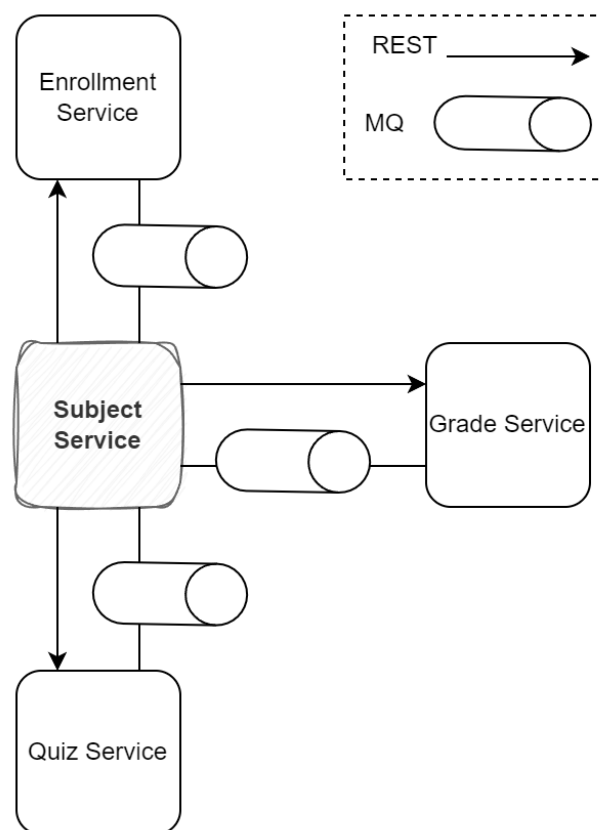


28. ábra Subject és Lesson modellek

A Subject Service felelős a tantárgyak és a hozzá tartozó tanórák kezeléséért. Ezeket a tanár szerepkörű felhasználók hozzák létre, akik szerkeszthetik, archiválhatják, valamint publikálhatják őket. A diákok pedig felvehetik és elvégezhetik a tárgyakat.

Két adatmodellt - Subject és Lesson, 28. ábra – kezel, melyek egy-a-többhöz viszonyban állnak egymással. A közös, automatikusan kezelt mezők egy őosztályba kerültek kiszervezésre.

A szolgáltatás több ms-sel is kommunikál mind REST végpontokon, valamint RabbitMQ-n keresztül, ahogy a 29. ábra mutatja.



29. ábra Subject szolgáltatás kommunikáció

A következő bekezdésekben a szolgáltatás által kínált fontosabb funkciókat mutatom be. Először a tanár, majd diák szemszögből.

Mivel a tanár jogkörű felhasználó csak a saját tantárgyait láthatja, ezért a bejelentkezési adatok alapján szükséges a felhasználó azonosítójára is szűrni az adatbázis lekérdezésekben. Ehhez a domain modellen definiáltam egy filtert, melyet szükség

esetén be lehet kapcsolni és ekkor automatikusan hozzáadja a feltételt a létrehozott SQL-hez.

```
@Filter(name = "creatorFilter")
@Entity
class Subject : TenantAwareBaseEntity()
```

33. forráskód Szűrés a rekord létrehozójára

Ennek bekapcsolásra több helyen is szükségem volt, így létrehoztam a Service osztályok fölé egy közös őssztályt. Itt kapott helyet a 34. forráskódban látható magasabb rendű függvény, – High Ordered Function, vagy HOF – mely az átadott metódus meghívása előtt bekapcsolja a szűrőfeltételt, illetve beállítja az aktuális felhasználó azonosítóját a `userId` paraméter értékeként.

```
fun <T> withUserFilter(query: () -> T): T {
    val session = entityManager.unwrap(Session::class.java)
    val filter = session.enableFilter("creatorFilter")
    filter.setParameter("userId", getUserId())
    val result = query()
    session.disableFilter("creatorFilter")
    return result
}
```

34. forráskód Felhasználó bekapcsolása

Ezután meghívja a paraméterként kapott funkciót, majd kikapcsolja a filtert és visszatér az eredménnyel. A szűréshez a modellen is definiálni kell a filtert, mely összekapcsolja az átadott változó értékét az adatbázis mezővel, ez a 35. forráskód példában látható.

```
@FilterDef(
    name = "creatorFilter",
    defaultCondition = "created_by = :userId",
    parameters = [
        ParamDef(name = "userId", type = String::class)
    ]
)
```

35. forráskód Filter létrehozása

A felhasználó lekérése történhet az `authInfo` metóduson keresztül, de a statikus metódust sok helyen hívva a tesztelés és módosítás is nehézkessé válik, ezért ezt kiemelttem egy interfészre, így a tesztekben könnyedén létrehozható a valós felhasználó lekérdezés helyett a szükséges utánpótlás - mock – objektum.

```
interface UserProvider {
    fun getUser() = authInfo().orBlow()
    fun getUserOrNull() = authInfo()
    fun getTenantId() = authInfo().orBlow().tenantId
    fun getUserId() = authInfo().orBlow().id
}
```

36. forráskód Felhasználó adatait biztosító interfész

A Kotlin nyelvben lehetőség van metódusok automatikus delegálására, a fordító létrehozza a származtatott osztályban a felülírt metódusokat és meghívja benne a delegált osztály azonos metódusát. A szolgáltatások ősosztálya – BaseService – delegálja a UserProvider interfész metódusait az interfészt implementáló objektumon keresztül. Így könnyen és cserélhető módon kezelhetők a bejelentkezési adatok az üzleti logikát megvalósító Service rétegben. Ez látható a 37. forráskód példában.

```
object UserProviderImpl : UserProvider

abstract class BaseService(
    private val userProvider: UserProvider = UserProviderImpl
) : UserProvider by userProvider {...}
```

37. forráskód - Delegált

A tanári jogosultsághoz kapcsolódó kérések belépési pontja a TeacherController osztály. Ezen kéréseket az /api/v1/teacher kezdetű végponton várja a rendszer, ahol a v1 az aktuális API verziót jelöli.

```
@RestController
@PreAuthorize("hasRole('TEACHER')")
@RequestMapping("/api/v1/teacher")
class TeacherController(
    private val subjectService: TeacherSubjectService,
    private val lessonService: TeacherLessonService,
)
```

38. forráskód Tanári kéréseket kiszolgáló végpont

A @RestController annotáció jelzi a keretrendszernek, hogy egy olyan Bean-t szeretnék létrehozni, amely végpontokat tartalmaz és alapesetben az üzenetváltáshoz JSON formátumot használja. Az osztályba két üzleti logikát kezelő Service példányt injektáltam, ezek végzik az adatok feldolgozását.

```

@GetMapping
fun getMySubjects(
    @RequestParam(name = "archive", defaultValue = "false")
    includeArchive: Boolean,
    @RequestParam(name = "page", defaultValue = "0")
    pageNumber: Int
): ResponseEntity<Page<Subject>> {
    val page = PageRequest.of(pageNumber, 8)
    val result = subjectService.findAll(includeArchive, page)
    return ResponseEntity.ok(result)
}

```

39. forráskód Lapozható tantárgyaim végpont

A saját tárgyak lekérésénél lehetőség van lapozásra, illetve az archivált tantárgyak listázására is. Ezt URL query paraméter segítségével oldottam meg, amely egy logikai és egy lapszám értéket vár bemenő paraméterként. Amennyiben valamelyik hiányzik, az alapértelmezett értékek kerülnek alkalmazásra. Egy lapon fixen nyolc tantárgy kapott helyet.

A Controller metódus ezután az üzleti logikai réteg felé továbbítja a kérést, amely a bejövő boolean érték alapján eldönti, melyik lekérdezést kell végrehajtani.

```

fun findAll(includeArchive: Boolean, page: PageRequest):
Page<Subject> {
    return if (includeArchive) {
        val subjectsPage = withUserFilter {
            repository.findAllBy(page)
        }
        enhanceWithOccupiedSeats(subjectsPage)
    } else {
        val subjectsPage = withUserFilter {
            repository.findAllByArchiveFalse(page)
        }
        enhanceWithOccupiedSeats(subjectsPage)
    }
}

```

40. forráskód Lapozható tantárgyaim, szolgáltatás réteg

A lekérdezés végrehajtását az adatbázisban a Repository réteg végzi. Amennyiben csak az aktív tárgyakat szeretné listázni a tanár, a beépített *findAll* metódust használtam, ha azonban az archivált tantárgyakra is szüksége van, a már említett Query-by-Method funkcionalitás segítségével történik a lekérdezés.

```

@Transactional(propagation = Propagation.MANDATORY)
interface SubjectRepository : JpaRepository<Subject, String>,
CustomSubjectRepository {

    @EntityGraph(value = "Subject.noJoin")
    fun findAllBy(page: PageRequest): Page<Subject>
}

```

Problémát jelent a kapcsolt adatbázis táblák – esetemben a tanórák – laza betöltése. A JPA alapesetben nem tölti be az lista típusú kapcsolatokat, de amikor a sorosító algoritmus megkezdje ennek feldolgozását, a JPA automatikusan minden tantárgyhoz egyesével, az azonosító alapján lekéri a hozzá tartozó tanórákat az adatbázisból. Ez a jelenség úgy is ismert, mint N+1 lekérdezés probléma [14]. Tegyük fel, hogy az eredménylista 100 tantárgyat tartalmaz, ilyenkor mindegyikhez egy kérést küld a rendszer az adatbázis felé a tantárgyhoz tartozó tanórák betöltésére, összesen tehát 100 lekérést és egyet a tantárgyak betöltésére. Ez teljesítménygondokat, illetve az adatbázis túlterhelését eredményezheti, valamint több kapcsolattal rendelkező entitás esetén az ms memóriája is elfogyhat a hatalmas betöltött adatmennyiség miatt.

Erre kínál megoldást a JPA Entity Graph funkcionalitás, mely segítségével megfogalmazható, mi történjen kollektciók betöltése esetén. Két ilyen gráfot hoztam létre, a Subject.noJoin esetén a JPA egyáltalán nem tölti be a tanórák listáját a tantárgyakhoz, a Subject.lesson esetén pedig egyetlen lekérést használ, melyben join művelet segítségével kapcsolja össze a táblákat.

```

@NamedEntityGraphs(
    value = [
        NamedEntityGraph(
            name = "Subject.lessons",
            attributeNodes = [
                NamedAttributeNode("lessons")
            ],
        ),
        NamedEntityGraph(name = "Subject.noJoin")
    ]
)
class Subject : TenantAwareBaseEntity()

```

41. forráskód Entity Graph a tantárgyak modellhez

Miután az adatok lekérése befejeződött, szükség van a tantárgyra beiratkozott tanulók adataira is. Ez az Enrollment Service-től kérhető el, mely egy szinkron REST kérést küld a

szolgáltatásnak, majd a kapott számértékkel frissíti a válasz objektumot. Ebben segítségemre volt a Feign kliens, mely a Spring Data könyvtárhoz hasonlóan interfész alapú lekérdezés megfogalmazását teszi lehetővé, de ebben az esetben REST végponton keresztül. A lekérdezések végrehajtásához a meghívni kívánt ms-ekhez létrehoztam egy-egy ilyen interfészt, ahol megtalálható az összes, adott ms felé induló kérés. Ennek egy részlete látható a 42. forráskód Enrollment Feign kliens példában.

```
@FeignClient(  
    name = "enrollment-api",  
    url = "\\${knowhere.service.enrollment}/internal/api/v1",  
    configuration = [FeignConfig::class]  
)  
interface EnrollmentApi {  
  
    @RequestMapping(  
        method = [RequestMethod.GET],  
        value = ["/student-count/{subjectId}"]  
    )  
    fun studentsCountForSubject(  
        @PathVariable("subjectId") subjectId: String  
    ): CountResponse
```

42. forráskód Enrollment Feign kliens

A meghívni kívánt szolgáltatás URL címét környezeti változóból olvassa fel a program. Az interfész metódusok a Controller rétegben is használt annotációkkal paraméterezhetők, így a két réteg fejlesztése nagyon hasonló. A háttérben a keretrendszer ez alapján RestTemplate segítségével kérést küld a másik rendszernek és a válaszból legyártja a visszatérési érték típusának megfelelő objektumot. Az interfészek a 3.2.4.7 fejezetben bemutatott OpenApi leíróból automatikusan generálhatók.

A mentést, archiválást, publikálást egyszerű HTTP igék használatával oldottam meg, ahol az adatok ellenőrzés után az adatbázisban kerülnek mentésre.

A diákok lekérhetik a felvett és elérhető tantárgyak listáját, beiratkozhatnak a kiválasztott tantárgyakra, illetve leadhatják azokat. A tantárgy és tanóra adatlapját is megtekinthetik. Ezen funkcionális megvalósításához több szolgáltatás együttműködésére van szükség. A diákokhoz tartozó funkcionális megvalósításáért a StudentController, StudentSubjectService és a StudentLessonService osztály a felelős. Az alábbi példában egy tantárgy betöltését mutatom be, ahol a diák már beiratkozott rá. A végpont egy GET kérést fogad, ahol az azonosítót URL paraméterként kapja az ms.

A válasz pedig a tantárgy objektum kiegészítve a tanórák és a tanuló által megszerzett pontszámok adataival.

```
@GetMapping("/enrolled/{id}")
fun getEnrolledSubject(
    @PathVariable id: String
): ResponseEntity<ActiveSubjectLessons> {
    val response = subjectService.findEnrolledWithGrades(id)
    return ResponseEntity.ok(response)
}
```

43. forráskód Tantárgy lekérése, melyre a tanuló beiratkozott

Az azonosítót használva az adatbázisból betöltöm a tantárgyat. Amennyiben ilyen azonosítóval nem található bejegyzés, a rendszer 404 Not Found hibakóddal tér vissza. Ha sikeres a betöltés, a Grade Service-től lekérem a tantárgyhoz tartozó órákon a tanuló által elért pontszámokat. A Quiz Service-től pedig a megoldott kérdéssorok listáját. Ez a két szolgáltatás egy-egy listát ad vissza. A listák elemeit párosítani kell a tantárgy tanóra listájában található rekordokkal. Ez megoldható lenne egy kiválasztással minden tanórához, így viszont N tanóra és a hozzá tartozó N bejegyzés esetén $O(N^2)$ komplexitást eredményezne. Ezért az eredmények és kérdéssor megoldások esetén is egy hasítótáblát hoztam létre – ez könnyen megtehető a beépített `associateBy` függvénnyel -, ahol a kulcs a tanóra azonosítója az érték pedig az adott listaelem. Ebből már $O(1)$ alatt elkérhető egy elem (amennyiben nincs hash ütközés). Így minden listán csak egyszer szükséges végigiterálni.

```
val gradeData =
    gradeIntegration.getStudentAssessmentsForSubject(
        subject.id, getUserId()
    )
    .associateBy { it.lessonId }

[...]

ActiveLesson(
    ...,
    points = gradeData[it.id]?.points
)
```

44. forráskód Értékelések lekérése és a lista hasítótáblává alakítása

A tantárgyi adatlapon a tanórára kattintva megtekinthető a hozzá tartozó videó és kitölthető a kérdéssor. A tanóra hozzáférhetőségét a megnyitás előtt meg kell vizsgálni. Amennyiben a felhasználó nem jogosult a megtekintésre, hibaüzenetet kap. A felületen

a tanóra ilyen esetben le van tiltva, ahogy a 13. ábra mutatja, de ez nem véd az illetéktelen hozzáféréstől – mint például az URL alapú lekérdezés vagy felhasználó felület lokális módosítása. Ezért amikor a tanuló az adott óra adatlapját lekéri, ellenőrzöm ezen feltételeket, illetve a tanóra határidejét, amennyiben ez be van állítva.

```
lesson.deadLine != null &&  
lesson.deadLine?.isBefore(Instant.now()) != false
```

45. forráskód Tanóra határidejének ellenőrzése

Ezután a tantárgy vizsgálata következik. Csak azon tárgyhoz tartozó órák érhetőek el, amelyek publikáltak és az aktuális időpont a kezdő és végdátum között van. Ezt a vizsgálatot SQL segítségével hajtottam végre a 46. forráskódban látható módon.

Ha nem található tantárgy, akkor a kérés elutasításra kerül. Ezután a külső rendszerekhez kötött vizsgálatok következnek. Meg kell vizsgálni, hogy a tanuló felvette-e a tantárgyat, mivel csak azok a hallgatók érhetik el az órát, akik ezt sikeresen megtették. Majd a következő vizsgálat ellenőrzi, hogy megoldotta-e már a kérdéssort a felhasználó, illetve szerzett-e jegyet. Bármelyik igaz, a kérés elutasításra kerül, amely jelen esetben egy `ApiException` típusú kivétel dobását jelenti a megfelelő hibaüzenettel.

```
override fun findAvailableForEnrollById(subjectId: String):  
Subject? {  
    val now = Instant.now().toString()  
    val tenant = authInfo().orBlow().tenantId  
    val sql = ""  
        select * from subject s  
        where s.published = true  
            and s.archive = false  
            and (s.end_dt >= cast(:now as timestamp)  
                or s.end_dt is null)  
            and s.tenant_id=:tenant  
            and s.id=:subjectId  
        """.trimIndent()  
  
    val nativeQuery = entityManager.createNativeQuery(  
        sql, Subject::class.java)  
    nativeQuery.setParameter("now", now)  
    nativeQuery.setParameter("tenant", tenant)  
    nativeQuery.setParameter("subjectId", subjectId)  
    return nativeQuery.singleResult as Subject  
}
```

46. forráskód Elérhető tantárgy lekérése az adatbázisból

A tantárgyak rendelkezhetnek Kurzus Vége dátummal. A lejárt tantárgyakról RabbitMQ üzenetet küldök, melyet adott ms-ek figyelnek és módosítják a saját adataikat az eseménytől függően. Ehhez ütemezett feladatot -más néven Job-ot - használtam.

A Job lekérdezi az adatbázisból a lejárt tantárgyakat. Ehhez SQL lekérdezést használtam, mely összegyűjti az elmúlt órában lejárt tantárgyakat.

```
select * from subject s
  where s.end_dt < cast(:now as timestamp)
 and s.end_dt > cast(:pastOneHour as timestamp)
```

47. forráskód Lejárt tantárgyak lekérdezése

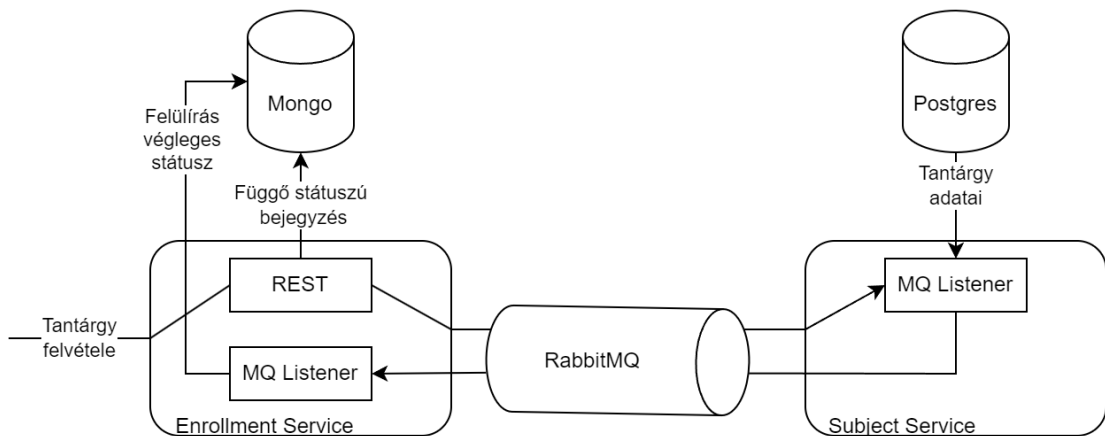
Majd a tantárgyakat a RabbitMQ üzenetsorba teszi. Innen tudja kivenni például az Enrollment Service, ami lezárt státuszúra állítja a diákok jelentkezéseit a tantárgyon.

Ahhoz, hogy a rendszer több futó példány esetén is csak egy futtassa a Jobot, elosztott zár mechanizmust használtam. A zárolás adatbázison keresztül történik, így csak az a példány kap jogot a futtatásra, amelyik megszerzi és birtokolja ezt.

```
@SchedulerLock(
    name = "subject_expiry",
    lockAtLeastFor = "10m",
    lockAtMostFor = "20m"
)
@Scheduled(fixedDelay = 30, timeUnit = TimeUnit.MINUTES)
fun watchSubjectExpiry() {
    val subjects = subjectRepository.findExpiredInLastHour()
    subjects.forEach { subject ->
        logger.info { "Subject ${subject.id} expired." }
        rabbitTemplate.convertAndSend(
            SUBJECT_CLOSED_EVENT_NAME,
            subject
        )
    }
}
```

48. forráskód Lejárt tantárgy üzenetsorra helyezése

A rendszer a tantárgyfelvételt és a jegybeírást aszinkron módon kezeli. Amikor egy diák felvesz egy tantárgyat, az Enrollment Service üzenetet küld erről az eseményről. A Subject Service ezt a csatornát hallgatja. A kapott üzeneteket kiegészíti a tantárgy adataival és az üzenetsorra helyezi. Ennek megvalósításához használt architektúrát a 30. ábra illusztrálja.



30. ábra Enrollmment üzenet feldolgozásának folyamata

A megvalósítás az EnrollementQueueListener osztályban található. Itt hoztam létre a RabbitMQ üzeneteket fogadó metódust.

A `@RabbitListener` annotációban adható meg a figyelni kívánt üzenetsor neve. A `RabbitTemplate` segítségével pedig a megadott nevű üzenetsorra helyezhető a válasz. A csatornák beállítása és létrehozása a `RabbitConfiguration` osztályban található.

```

@RabbitListener(queues = [ENROLLMENT_IN_QUEUE_NAME])
fun listener(message: EnrollmmentMessage) {
    withAuthContext(message.tenantId) {
        val subject =
            subjectService.findAvailable(message.subjectId)

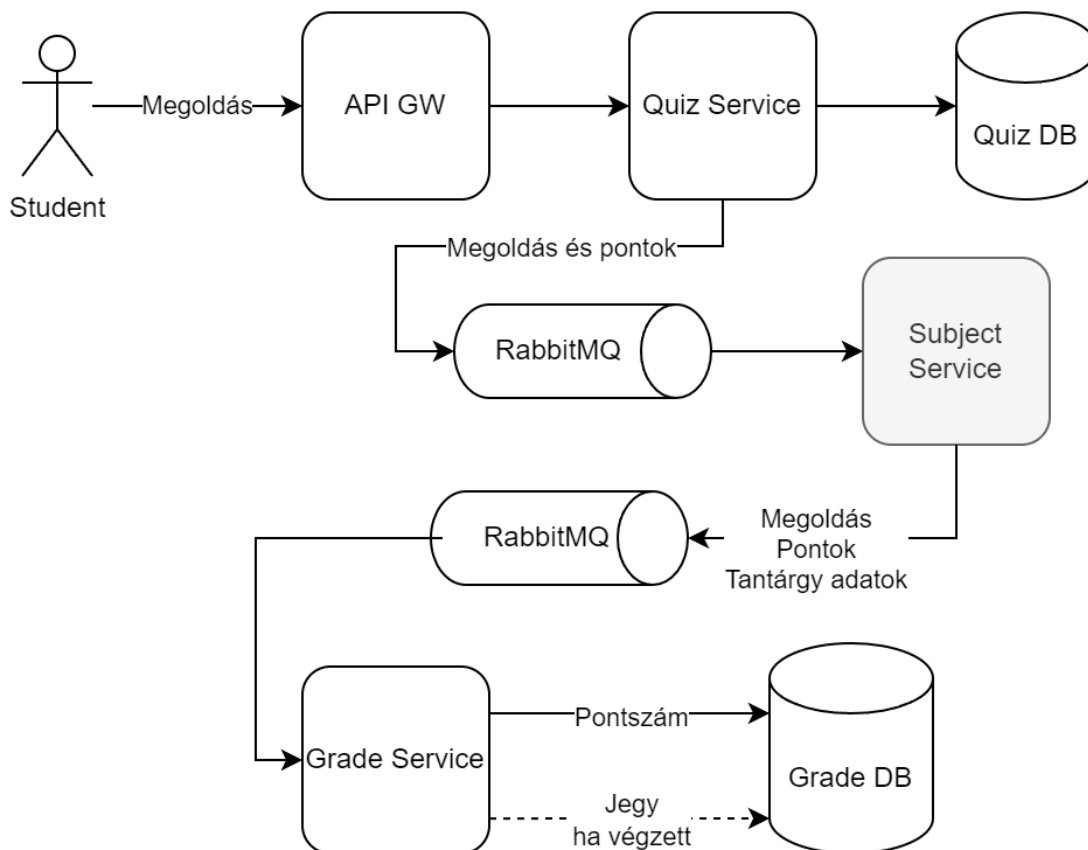
        val response =
            if (subject == null) {
                getSubjectMessage(message, 0, false)
            } else {
                getSubjectMessage(
                    message, subject.seats!!, true,
                    subject.name
                )
            }

        rabbitTemplate.convertAndSend(
            ENROLLMENT_OUT_QUEUE_NAME, response
        )
    }
}

```

49. forráskód Enrollmment üzenet feldolgozása

Amikor a tanuló beküldi a kérdéssor megoldását, a Quiz Service publikálja ez a megoldást a kiértékelés eredményével együtt.



31. ábra Megoldás kiértékelése

A Subject Service ezt az üzenetet is figyeli és kiegészíti a tantárgy adataival, majd az üzenetsorra helyezi, amelyet a Grade Service hallgat és beírja a pontszámokat az adatbázisba, illetve szükség esetén kiszámolja a végső jegyet is. A teljes folyamatot a 31. ábra mutatja be. A megvalósítás az előző példában bemutatott felépítést használja és a `SolutionQueueListener` osztály valósítja meg.

3.2.8 Tantárgy felvétele szolgáltatás

Az Enrollment Service biztosítja a tantárgyak felvételéhez, illetve leadásához szükséges funkcionalitást, illetve kezeli az adatmodellt, melyet a 32. ábra ismertet.

A modell tárolja, mely diák mely tantárgyakat vette fel. A státusz mező pedig ennek állapotát írja le. A tantárgy felvétele aszinkron módon történik, amikor a diák rákattint a felvétel gombra, egy új rekord jön létre az adatbázisban *feldolgozás alatt* státusszal. A szolgáltatás ezután üzenetet küld a tantárgyakat kezelő szolgáltatásnak, amely a 3.2.7 fejezetben leírt módon ellenőrzi, felvehető-e a tantárgy a tanuló számára. Erről értesítést küld vissza, mely alapján új státusz kerül beállításra. Ha minden hely betelt,

akkor *nincs hely*, ha már nem lehet jelentkezni a tárgyra, akkor *hiba*, ha pedig minden ellenőrzés sikeres volt, akkor a *beiratkozva* státusz. Ez utóbbi esetben a tantárgy megjelenik a diák által felvett tantárgyak listájában a felületen. A tanuló minden esetben üzenetet kap az állapotváltozásról, melyben megtalálható annak oka is.

Enrollment
id: String
subjectId: String
studentId: String
status: String
version: Long
modifiedAt: Instant
createdAt: Instant
tenantId: String

32. ábra- Tantárgyfelvétel adatmodell

A tantárgyat elvégezve a Grade Service, a tantárgy lezárása esetén pedig a Subject Service küld üzenetet, mely alapján tantárgyfelvétel státusza *befejezett*, illetve *lezárt* állapotra vált. A tantárgy leadása esetén pedig *leadva* lesz.

Emellett a szolgáltatás végpontokat kínál a többi szolgáltatás számára a tantárgyfelvételi adatok lekéréséhez. Ezek a végpontok a felhasználók számára nem érhetők el. Lekérhető többek között a tantárgyat felvett hallgatók száma és adatai, illetve egy tanuló összes felvett tárgya.

3.2.9 Videók lejátszása

A Video Service felelős a videók feltöltésének és lejátszásának kezeléséért. A videófájlok mentése jelenleg lokális tárhelyre történik, melyhez a virtuális meghajtókat a Kubernetes kezeli. Ennek részleteit a 3.4.2 fejezetben tárgyalom. Igény esetén lehetőség van hálózati meghajtó csatolására is. Ez lehet AWS S3, Azure File Storage vagy más egyéb megoldás. Videót csak a tanári jogkörrel rendelkező felhasználók tölthetnek fel. Ehhez a felületen ki kell tallózni a megfelelő formátumú videó fájlt, illetve megadni egy

felhasználóbarát elnevezést. Lehetőség van a videó elérését publikusra állítani, ekkor nem csak a feltöltő, hanem az intézmény összes tanára láthatja és felhasználhatja a videót a tanórák összeállításához. A feltöltéshez a frontend multipart/formdata típusú POST kérést küld, mely átadja a fájl folyamat az ms-nek. Ez látható az 50. forráskód példában.

```
@PostMapping
@PreAuthorize("hasRole('TEACHER')")
fun upload(
    @RequestPart("file", required = true) file: MultipartFile,
    @RequestPart("title", required = true) title: String,
    @RequestPart("public") public: String?
): ResponseEntity<VideoMetadata> {
    val response = service.saveVideo(
        file, title, public.toBoolean()
    )
    return ResponseEntity.status(HttpStatus.CREATED)
        .body(response)
}
```

50. forráskód Controller fájl feltöltéséhez

A VideoService osztály saveVideo metódusban létrehozok egy adatbázis bejegyzést, amelyben tárolom a fájl nevét és egyéb metaadatait. A fájl az ütközések elkerülésének érdekében prefixként megkapja a mentett rekord egyedi azonosítóját. Ezután ellenőrzöm, hogy a mentéshez használt könyvtár létezik-e. Ha nem, akkor ez létrehozásra kerül, majd a fájl lemezre írása következik.

```
private fun writeFile(file: MultipartFile, fileName: String) {
    checkAndCreateDir()
    file.transferTo(Path.of("$VIDEO_DIR/$fileName"))
}
```

51. forráskód Fájl lemezre írása

Ezután belső adatbuszra üzenetet küldök a feltöltési eseményről és végül a metódus visszatér a létrehozott adatbázis bejegyzéssel. A felhasználó így már lejátszhatja a videót. Az adatbuszt hallgató eseménykezelő a háttérben aszinkron módon megkezd a borítókép kivágását a videóból. A fájlhoz tartozó adatfolyam elérése után az első kulcs képkockát próbálom meg kivágni. Amennyiben a videó nem tartalmaz ilyet, akkor a 101. képkockát. Ezt a képkockát ezután képpé alakítom, majd elmentem a bélyegképeket tartalmazó mappába és beírom a nevét a videó metaadatait tartalmazó rekordba. Az ezt követő kéréseknél a frontenden megjelenik a videóból származó kép az ideiglenes

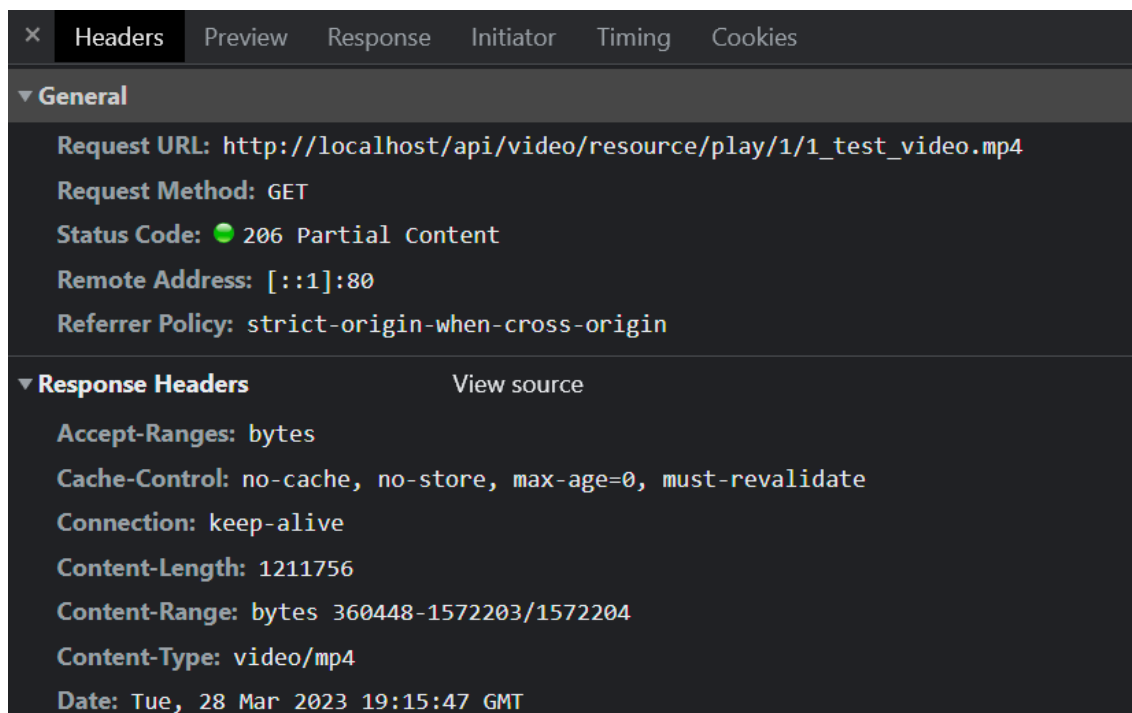
helyett. A kivágáshoz az ffmpeg csomagot használtam, a teljes feldolgozás a 52. forráskód példában látható.

```
private fun createThumbnail(video: VideoMetadata) {
    try {
        val grabber =
            FFmpegFrameGrabber("$VIDEO_DIR/${video.fileName}")
        try {
            grabber.start()
            val thumbnailName = "${newId()}.jpeg"
            val converter = Java2DFrameConverter()

            var frame = grabber.grabKeyFrame()
            if (frame == null) {
                for (i in 0..99)
                    grabber.grabImage()
                frame = grabber.grabImage()
            }
            val bufferedImage = converter.convert(frame)
            ImageIO.write(
                bufferedImage,
                "jpeg",
                File("$THUMBNAIL_DIR/$thumbnailName"))
            val updated = video.copy(thumbnail = thumbnailName)
            repository.save(updated)
        } finally {
            grabber.stop()
        }
    } catch (e: Throwable) {
        logger.error { "Cannot create thumbnail for ${video.id}" }
    }
}
```

52. forráskód Bélyegkép kivágása videóból

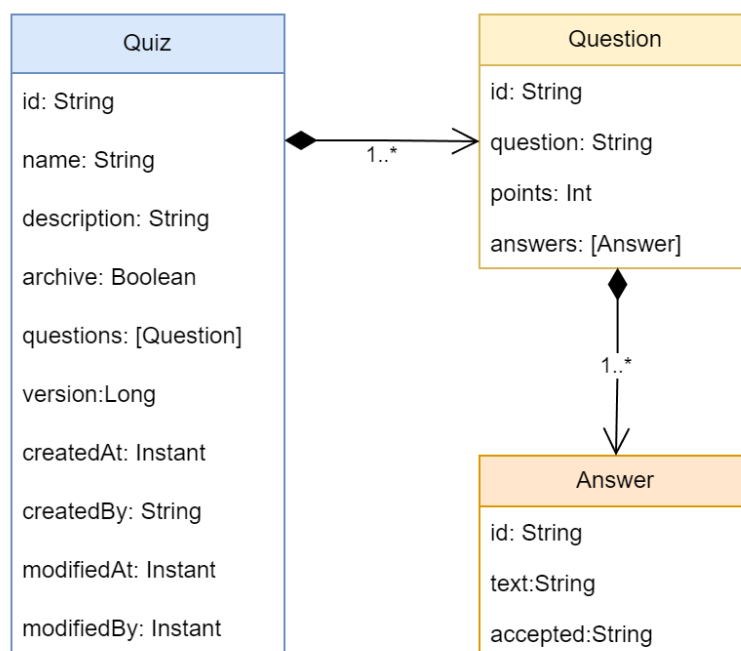
A keretrendszer automatikusan támogatja a fájlok részleges lekérését. Videó lejátszásnál ezt kihasználva a frontend a kérés fejlécében elküldi az elérni kívánt erőforrás tartományt a Content-Range mezőben, mely négy részből áll. Az első a mértékegység, amelyben a tartomány megadása történik, általában bájt. Ezt követi a tartomány kezdő és utolsó bájtjának pozíciója kötőjellel elválasztva, végül a teljes fájl méret. Az ms a videó ezen részét küldi vissza, így a kliens lejátszás közben folyamatosan tudja betölteni az adatokat, nem kell megvárni a lejátszással, amíg az egész videó letöltődik. A válasz státuszkódja ebben az esetben 206 Partial Content. Ennek a kérésnek és válasznak a fejlécéről készült képernyőfotót mutat a 33. ábra.



33. ábra Részleges erőforrás lekérés

3.2.10 Kérdéssor szolgáltatás

A Quiz Service tárolja és szolgálja ki a kérdéssorok adatait, illetve végzi a megoldások kiértékelését. Az adatmodellt a 34. ábra mutatja be. A tanároknak lehetősége van új kérdéssort létrehozni, meglévőt módosítani, illetve archiválni azt. A funkcionalitás a korábban tárgyalt REST megoldáson alapul.



34. ábra Quiz adatmodell

A feladatsor megoldásának beküldése POST metóduson keresztül történik, az üzenettörzs tartalmazza a tanóra, valamint a kérdéssor azonosítóját, mivel egy kérdéssor több órához is tartozhat. Emellett kulcs-érték párokként a kérdés azonosítóját és a megoldás azonosítóját is.

A rendszer validáció után elmenti a válaszokat, majd megkezdi a kiértékelést. Ehhez betölti adatbázisból a Quiz rekordot, amely kérdéseiből egy hasítótáblát készít. Ennek kulcsa az egyedi azonosító, értéke pedig maga a Question objektum.

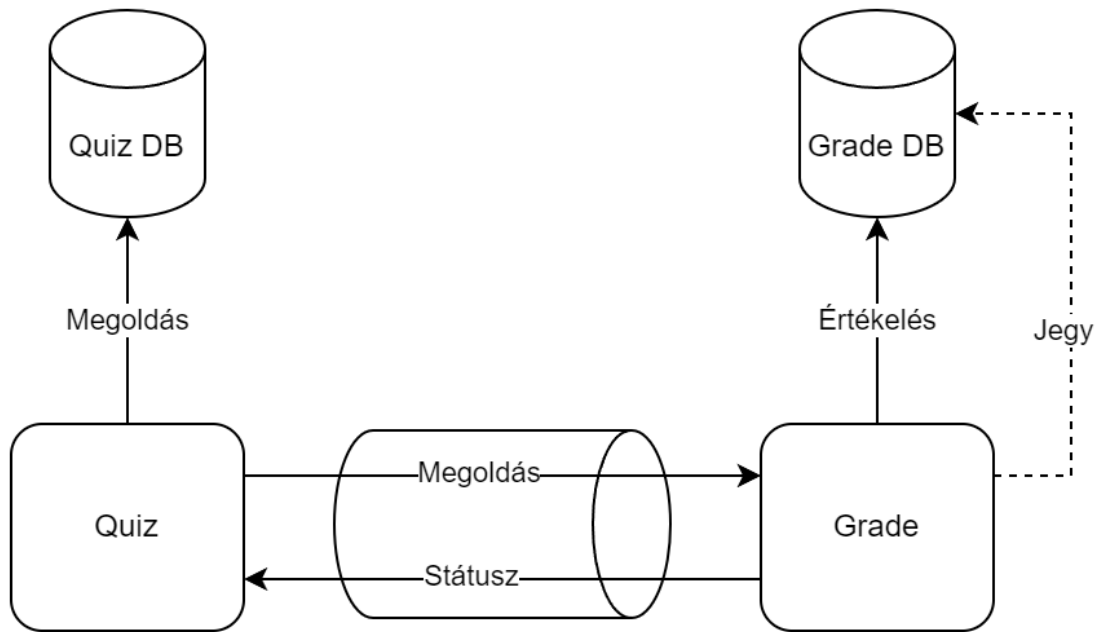
Ezután a diák válaszain végigiterálva minden válaszhoz megkeresi a hozzá tartozó kérdést, majd a kérdéshez tartozó helyes választ. Amennyiben ennek az azonosítója megegyezik a válasz azonosítójával, a kitöltés helyes, így jár érte a pont, egyébként 0 pontot kap a hallgató az adott válasza. Ezeket a részpontokat összegezve előáll a diák által elért pontszám. A kalkulációhoz használt kódrészlet a 53. forráskód példában található.

```
private fun calculatePoints(quiz: Quiz, answers:
List<QuestionAnswer>): Int {
    val questionsMap = quiz.questions.associateBy { it.id }
    return answers.sumOf {
        val question = questionsMap[it.questionId]!!
        val rightAnswer = question.answers.first { a ->
            a.accepted
        }
        if (it.answerId == rightAnswer.id) {
            question.points
        } else {
            0
        }
    }
}
```

53. forráskód Megoldás pontértékének kiszámítása

A kiértékelés után a megoldás adataival RabbitMQ üzenetet küldök a Grade Service-nek, majd a rekordot *elküldve* státuszúra állítom. Ezt a folyamatot illusztrálja a 35. ábra.

A feldolgozás innentől aszinkron módon történik. A Grade Service megkapja az üzenetet és feldolgozza azt, majd válaszüzenetet küld, amint végzett. Ezt az üzenetet a Quiz Service megkapja és a rekord státuszát *befejezett* állapotúra állítja, mely jelzi, hogy sikeresen fel lett dolgozva a kiértékelte megoldás és az érte járó pontszám beírásra került.



35. ábra Megoldás kiértékelése és értékelés mentése

Előfordulhat azonban, hogy nem érkezik válaszüzenet, mely bármilyen rendszerhibából, de akár az üzenetsor hibájából is adódhat. Ekkor nem tudni biztosan, a másik rendszer feldolgozta-e az üzenetet és beírta-e a jegyet az adatbázisba. Erre nyújt megoldást a Kimenő üzenet – Outbox - tervezési minta. Az ms egy időzített Job-ot futtat, mely összegyűjti azokat a megoldás rekordokat, amelyek több mint tíz perce lettek kiküldve, a státuszuk még nem *befejezett* és maximum háromszor voltak újrapróbálva. A lekérdezés megfogalmazásához MongoDB Criteria leírót használtam, mely a 54. forráskód példában található.

```

override fun findForMqRetry(): List<Solution> {
    val before10Minutes = Instant.now().minus(10,
ChronoUnit.MINUTES)

    val criteria = Criteria.where("attempts").lte(3)
        .and("status")
        .`in`(listOf(ProcessStatus.NEW, ProcessStatus.SENT))
        .and("lastAttempt").lte(before10Minutes)

    return mongoTemplate.find(MongoQuery.query(criteria),
Solution::class.java)
}
  
```

54. forráskód Befejezetlen kiértékelések összegyűjtése

A kapott rekordok közül azokat, amik háromnál kevesebbszer voltak elküldve, újra az üzenetsorra rakom. Amelyek már többször, azok státuszát *hibás*-ra állítom és üzenetet küldök a felhasználónak, hogy a probléma megoldása érdekében keresse fel az

oktatóját. Ezután a próbák számát megnövelve, az új státusszal együtt a rekordot frissítem az adatbázisban.

```
@SchedulerLock(name = "solution_send")
@Scheduled(fixedDelay = 5, timeUnit = TimeUnit.MINUTES)
fun sendFailedSolutions() {
    val failedSolutions = repository.findForMqRetry()

    failedSolutions.forEach {

        withAuthContext(it.tenantId) {
            val status = if (it.attempts < 3) {
                service.evaluateAndSend(it)
                it.status
            } else {
                val quiz = quizRepository.findOneById(it.quizId)
                sendNotification(quiz.name, it.studentId)
                ProcessStatus.FAILED
            }
            repository.save(
                it.copy(
                    lastAttempt = Instant.now(),
                    attempts = it.attempts + 1,
                    status = status
                )
            )
        }
    }
}
```

55. forráskód Megoldás újraküldése

3.2.11 Jegyek szolgáltatás

A Grade Service kezeli az órákhoz tartozó kérdéssorokon szerzett pontokat, valamint az ezekből számított végső érdemjegyeket. A domain modellt a 36. ábra mutatja be. A szolgáltatás csak egy publikus végponttal rendelkezik, melyen keresztül a diákok lekérhetik az érdemjegyeiket.

A végpont megvalósítása a GradeController osztályban található, mely GET kérést fogad, visszatérési értéke pedig a tanuló által elvégzett tantárgyak listája, mely tartalmazza a megszerzett jegyeket, pontszámokat és krediteket listája.

A szolgáltatás a bejelentkezett diákhöz tartozó összes jegyet lekéri az adatbázisból. A rekordok tartalmazzák a tantárgy azonosítóját, mely alapján a Subject Service-ből lekérem a tantárgyak nevét és kiegészítem vele a válasz objektumot.

Az Grade Service privát végpontokon a többi ms felé is kiszolgálja ezeket az adatokat. Lehetőség van lekérni adott tantárgyhoz tartozó jegyeket és pontszámokat.

Grade	Assessment
id: String	id: String
subjectId: String	subjectId: String
userId: String	lessonId: String
mark: Int	studentId: String
points: Int	points: Int
credit: Int	maxPoints: Int
version: Long	solutionId: String
createdAt: Instant	version: Long
modifiedAt: Instant	createdAt: Instant
tenantId: String	modifiedAt: Instant
	tenantId: String

36. ábra- Jegy és Értékelés adatmodell

A jegyek és értékelések kalkulációja az üzenetsoron kapott megoldások fogadásával kezdődik. Ennek architekturális felépítését a korábbi fejezetekben már bemutattam.

Az üzenetet a GradeQueueListener osztályban található listen metódus fogadja.

```
@RabbitListener(queues = [GRADE_QUEUE_NAME])
fun listen(enhanced: EnhancedSolutionModel)
```

56. forráskód Megoldás üzenetsort hallgató metódus

A beérkező üzenetből a tanóra és tanuló azonosítója segítségével ellenőrzöm, hogy a diáknak az adott pontszám be lett-e írva korábban. Ilyen eset lehet, amennyiben a jegybeírás sikeres volt, de erről az értesítést a Quiz Service nem kapta meg. Ilyenkor az ott tárgyalt újraküldés miatt egy üzenet többször is megérkezhet. Emiatt szükséges az ellenőrzés.

```
val assessment =
    assessmentService.getAssessmentForStudentByLessonId(lessonId,
        studentId)

if (assessment == null) {
    saveAssessment(enhanced.subjectId, solution)
    sendAssessmentNotification(enhanced.subjectName, solution)
    gradeStudent(enhanced)
}
```

57. forráskód Értékelés beírása a megoldáshoz

Emellett az adatbázis rekordban a megoldás azonosítóját egyedi indexszel láttam el, így semmiképp nem kerülhet be az értékelés kétszer az adatbázisba. Ha még nem lett beírva, akkor elmentem a megoldás alapján az értékelést, melyről a diák értesítést is kap.

Ezután a jegy generálás vizsgálata következik. Ehhez össze kell gyűjteni a diák által eddig megoldott kérdéssorok adatbázis rekordjait. Amennyiben a tantárgy összes órájához tartozó kvízt kitöltötte, kiszámítom a kérdéssorokon elért pontszámok összegét, illetve a lehetséges maximális pontszámot. Az értékelés – Assessment – objektumok minden ehhez szükséges információt tartalmaznak. A számítás a Kotlin által beépített `sumOf` metódus segítségével könnyen megoldható, ez a listán végigiterálva a megadott mezőben található értékeket összeadja.

```
private fun gradeStudent(enhanced: EnhancedSolutionModel) {
    val studentId = enhanced.solution.studentId
    val assessments =
assessmentService.getStudentAssessmentsForSubject(
    enhanced.subjectId, studentId )
    if (assessments.size == enhanced.lessonCount) {
        val sumPoints = assessments.sumOf { it.points }
        val sumMaxPoints = assessments.sumOf { it.maxPoints }
        gradeService.gradeStudent(
            studentId, enhanced.subjectId,
            enhanced.credit, sumPoints, sumMaxPoints)
    }
}
```

58. forráskód Összpontszám kiszámítása és elküldése jegybeírásra

A rendszer jelenleg egy egyszerű számítási modellt alkalmaz az érdemjegy meghatározásához, mely a 4. táblázatban látható.

Elért százalékos eredmény	Jegy
0-44	1
45-59	2
60-69	3
70-84	4
85+	5

4. táblázat Jegyek számítása

Amennyiben a jegy elégtelen, a tanuló a tantárgyért nem kap kreditet, ellenkező esetben megkapja azt.

A jegybeírásról a többi ms RabbitMQ-n üzenet kap, valamint a diáknak is kiküldésre kerül a kapott jegyet és a tantárgy nevét tartalmazó értesítés.

3.2.12 Értesítések szolgáltatás

A Notification Service üzenetsoron fogadja a többi ms-től a felhasználónak szánt értesítéseket, melyeket adatbázisba ment. Az ehhez használt domain modell a 59. forráskódban látható.

```
@Document(collection = "notification")
data class NotificationEntity(
    val id: String,
    val userId: String,
    val title: String,
    val text: String,
    val status: Status,

    @Version
    var version: Long = 0,

    @CreatedDate
    var createdAt: Instant? = null
)
```

59. forráskód Notification domain modell

A dokumentum tartalmazza a felhasználó azonosítóját, az értesítés címét és az üzenet szövegét, valamint státuszát, amely *új*, illetve *olvasva* lehet.

A rendszer lapozható végpontot biztosít a felhasználónak az üzenetek megtekintésére. Egyszerre 10 üzenetet lehet lekérni. Az oldalszámot a végpont query paraméter formájában fogadja. Ennek hiányában az első oldal jelenik meg – nullától indexelve.

```
@GetMapping
fun myNotifications(
    @RequestParam(name = "page", defaultValue = "0")
    pageNumber: Int
): Page<NotificationEntity> {
    val user = authInfo().orBlow()
    val page = PageRequest.of(pageNumber, 10)
    return repository.findByUserIdOrderByCreatedAtDesc(user.id,
page)
}
```

60. forráskód Értesítések lekérése

A lapozást a keretrendszer automatikusan kezeli a Query-by-Method funkcionalitás segítségével, melyhez a PageRequest objektum átadása szükséges. Ez tartalmazza az oldalszámot és lapméretet. A lekérdezésben MongoDB esetén a megfelelő adatok elérését a skip és limit függvények használatával valósítja meg a keretrendszer. A visszatérési érték egy Page objektum, mely tartalmaz egy, a típusparaméternek megfelelő listát, illetve a lap indexét, méretét és az összes rekord számát. Ez látható a 61. forráskód példában. A frontend ezt felhasználva képes implementálni a lapozást.

```
interface NotificationRepository :  
MongoRepository<NotificationEntity, String> {  
  
    fun findByIdOrderByCreatedAtDesc(  
        userId: String,  
        page: PageRequest  
    ): Page<NotificationEntity>  
  
    fun countByUserIdAndStatus(userId: String, status: Status):  
Long  
}
```

61. forráskód Lapozható értesítés adatbázis lekérés

Amikor a felhasználó a felületen rákattint egy olvasatlan üzenetre, a rendszer azt olvasottnak jelöli. A frontend ilyenkor PATCH kérést küld az értesítés azonosítójával. Az ms megkeresi a rekordot, átállítja a státuszát olvasottra és elmenti az adatbázisba.

```
@PatchMapping("/{id}")  
fun read(  
    @PathVariable id: String  
): NotificationEntity {  
    val notification = repository.findByIdOrNull(id)  
    if (notification != null) {  
        return repository.save(  
            notification.copy(status = Status.READ))  
    }  
    throw ApiException(Messages.NOT_FOUND, HttpStatus.NOT_FOUND,  
id)  
}
```

62. forráskód Értesítés olvasottnak jelölése

3.2.13 Felhasználói felület

A frontend alkalmazástól – bár jelentős mennyiségű kódot tartalmaz – csak röviden írok. A szakdolgozatom témája a háttérrendszerek felépítése, de ennek segítségével válik a rendszer használhatóvá.

A felhasználói felületet - mely a knowhere-vui könyvtárban található - VueJS [5] keretrendszer segítségével valósítottam meg JavaScript nyelven. A keretrendszer könnyen tanulható, mivel HTML, CSS és JS felhasználásával készülnek a komponensek, míg például a React könyvtárnak saját leíró nyelve van. Emellett a VueJS gyorsabb, mivel előre feldolgozott sablonokat használ. Lehetőséget biztosít a komponensek közötti egy- és két irányú kötések megvalósítására. Hátránya, hogy kevésbé rugalmas.

A felületi elemek megvalósításához ezenkívül a Vuetify keretrendszert használtam, mely előre gyártott komponenskészleteket biztosít az általános problémákra (például táblázatok, lenyíló menük). A stíluslapok kezeléséhez Tailwind CSS keretrendszert integráltam, mely előre elkészített stílusosztályokat kínál. Ezek könnyen személyre szabhatók.

```
axios.interceptors.request.use((config) => {
  const token = localStorage.getItem('jwtToken')
  if (token) {
    config.headers.Authorization = `Bearer ${token}`
  }
  if (!config.headers.Accept) {
    config.headers.Accept = 'application/json'
  }
  return config
})
```

63. forráskód - Axios interceptor

Az Api Gateway-jel történő kommunikációt az Axios könyvtár végzi, ehhez készítettem egy interceptort, amely automatikusan hozzáadja az Authorization fejléct a kérésekhez. Ezenkívül egy modult is létrehoztam a REST hívások kezelésére, ahol a lekérdezések definiálhatók. Például a felhasználók lekérdezése a 64. forráskódban.

```
_rest.loadAllUser = (callback) => {
  axios.get(`${authApi}/user`).then((response) => {
    safeCallback(callback, response)
  }).catch()
}
```

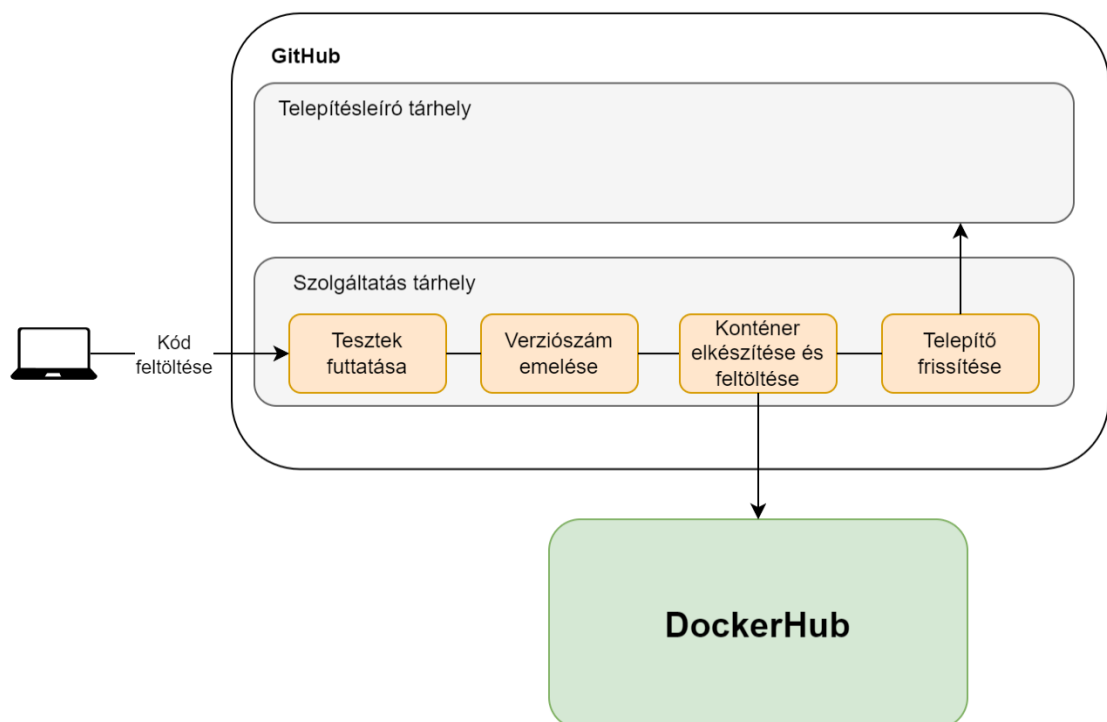
64. forráskód Felhasználók lekérdezése

A modul globális változóként beállításra kerül a keretrendszernek, így minden komponensből elérhető.

Emellett központi funkcionalitásként minden kérés esetén automatikusan elindul a folyamatjelző, mely a kérés befejezése után eltűnik. Ugyanígy központi hibakezelést is készítettem a lekérésekhez, ez bármely hiba esetén értesíti a felhasználót a hiba okáról. Az egyes oldalak egyedileg készültek a fent említett technológiák segítségével.

3.3 Futtatható állomány készítése

Az alkalmazás konténerizált környezetben fut, ezen konténerek előállításához több lépésből áll. Elsőként a Kotlin és Java osztályok fordítása és becsomagolása történik futtatható JAR állományba. Ehhez a Gradle [6] függőségkezelő és projektépítő könyvtárat használtam, mely az egyik legelterjedtebb eszköz JVM alapú alkalmazások esetén. Az alkalmazás függőségei a `build.gradle.kts` leíró fájlban vannak definiálva, ezeket az eszköz automatikusan letölti. Az építés a `gradle build` paranccsal indítható, mely lefordítja az elkészült osztályokat és a függőségekkel közösen futtatható állományba csomagolja az adott ms-t. A konténer létrehozásához a Google által fejlesztett JIB konténer építő eszközt használtam, mely a futtatható állományból elkészíti a konténert. Nagy előnye, hogy futó Docker daemon nélkül is képes OCI képfájlok létrehozására, így olyan szerveren is felépíthető a projekt, ahol nincs telepítve Docker. Emellett beépül a Gradle építési folyamatba. A konténer létrehozása a `gradle jib` paranccsal indítható.



37. ábra- Projekt építés

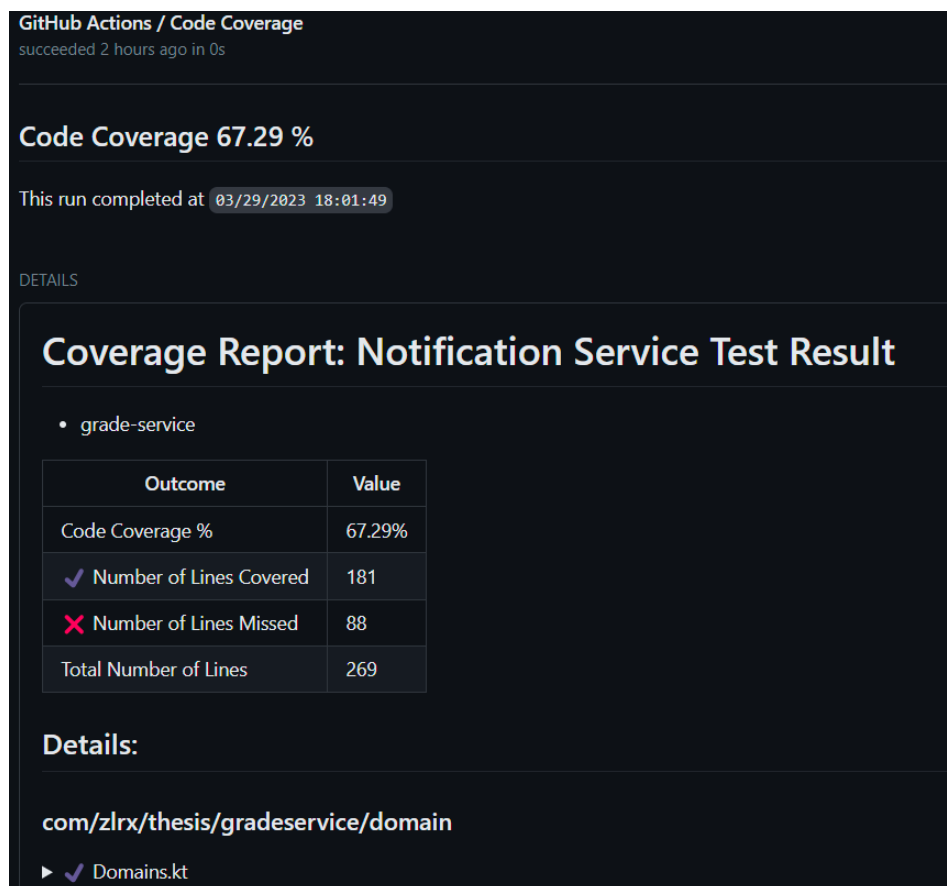
Mivel az összes ms és frontend építése kézzel sok időt venne igénybe, ezt a folyamatot automatizáltam. Az építés GitHub Action segítségével történik. Minden projekt tartalmaz ehhez egy `build-image.yml` nevű leíró fájlt. Az ebben leírt lépéseket 37. ábra

mutatja be. A forráskód a GitHub által nyújtott online git tárhelyre kerül feltöltésre. Az építés akkor kezdődik, ha új kód érkezik az adott projekt master ágára.

Ennek az első lépése a tesztek futtatása és az eredmények összegzése, majd feltöltése az aktuális építéshez, így minden építésnél látható, mely tesztek voltak sikeresek és melyek buktak el, valamint a lefedettség is megtekinthető. Egy ilyen riportot ismertet a 38. ábra.

Konténerizált rendszereknél fontos, hogy a konténerek verziókezeltet kerüljenek feltöltésre. A Kubernetes és Docker rendszerek a telepítésre átadott verziót indítják el, ám ha ugyanazzal a verziószámmal készül egy újabb konténer, – alapesetben ez latest – ami felülírja a régi verziót a képájlók tárhelyén, a vezérlő nem fogja az újabbat letölteni és telepíteni, mivel először a lokális tárból ellenőrzi a rendszer a verziószámot. Ezért az alkalmazás automatikus szemantikus verziózást [13] alkalmaz.

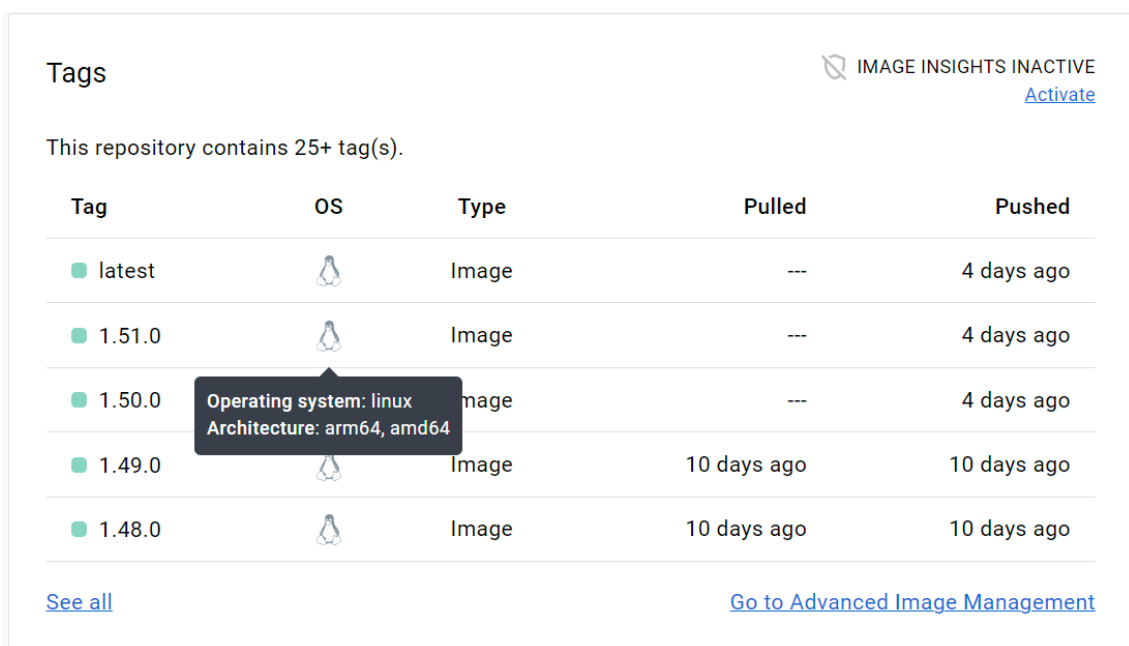
A lépés során a git commit üzenetek címkéi alapján kiszámítja az új alkalmazás verziót, mely visszaírásra kerül a gradle.properties fájlba, ahonnan a Gradle az alkalmazás és konténer verziókat kiveszi építésnél.



38. ábra- Teszteredmények

Ezután következik az alkalmazás fordítása, JAR fájl elkészítése majd a konténer összeépítése, valamint feltöltése a Docker Hub konténerkép tárolóba, mely nyilvántartja az összes elkészült konténer verziót, illetve elérhetővé teszi azt, ahogy a 39. ábra mutatja.

A konténer minden verzióját elkészítettem amd64 és arm64 alapú platformra. Utóbbi az új típusú Mac számítógépek miatt szükséges – amely M1 vagy újabb processzorral szerelt. A konténerek alapja az Amazon Corretto 17-es JVM-et tartalmazó kép fájl. A konténereken a 8000-es és 9000-es portokat engedélyeztem, az előbbi az alkalmazás kommunikációs portja, utóbbi pedig az alkalmazás telemetriáit szolgálja ki.



Tags

IMAGE INSIGHTS INACTIVE [Activate](#)

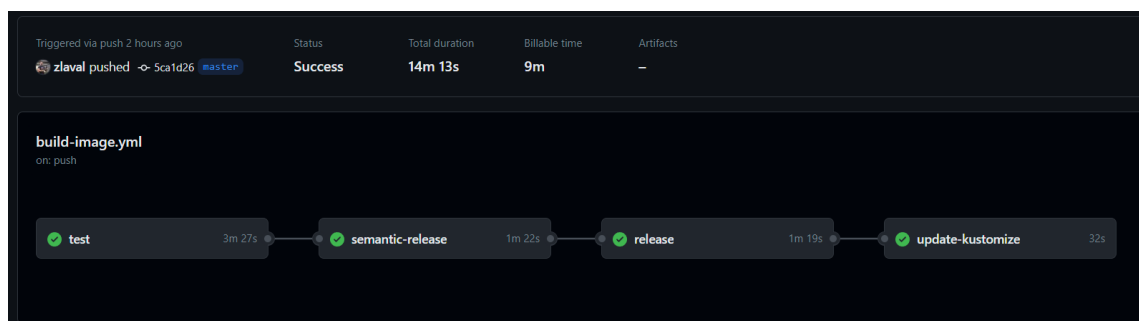
This repository contains 25+ tag(s).

Tag	OS	Type	Pulled	Pushed
latest		Image	---	4 days ago
1.51.0		Image	---	4 days ago
1.50.0		Image	---	4 days ago
1.49.0		Image	10 days ago	10 days ago
1.48.0		Image	10 days ago	10 days ago

[See all](#) [Go to Advanced Image Management](#)

39. ábra Subject Service listázása Docker hubon

Sikeres feltöltés után a Kubernetes telepítésleíró fájlokban automatizáltan kicserélem az adott konténer verziószámát az új verzióra.



40. ábra Építés munkafolyamat

A konténer elkészülte után lehetőség van konténer automatizált telepítésére. Ehhez a szintén Kubernetesen futó ArgoCD nevű szoftvert használtam. A program a GitHubon

elérhető telepítés leírókat 5 percenként - mely személyre szabható - letölti és elküldi a Kubernetes vezérlőnek, amely így verzió frissítés esetén telepíti az újabb képfájlt. Ennek segítségével a teljes folyamat automatizált az építéstől a telepítésig. A fejlesztőnek elég csak a forráskódot feltöltenie a master ágra.

A frontend építéséhez az npm csomagkezelőt használtam. Mivel az npm sok függőséget letölt, melyekre a végleges csomagban nincs szükség, valamint maga az npm alapú futtatás erőforrás igényes, így a konténer felépítését két részre bontottam. Ezt a Dockerfile nevű leírófájlban fejlesztettem le. Első lépésben egy köztes konténer jön létre, mely installálja ezeket a függőségeket, majd elkészíti a csomagolt alkalmazást. A második lépésben pedig ezt a csomagolt alkalmazást, illetve az nginx konfigurációs fájlt bemásolom az Nginx webserver alapú konténerbe. A konténer indításakor az nginx elindul és kiszolgálja a frontend alkalmazást a 80-as porton.

```
FROM node:lts-alpine as build-stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM --platform=$TARGETPLATFORM
      nginx:stable-alpine as production-stage
COPY --from=build-stage /app/dist /usr/share/nginx/html
COPY ./default.conf /etc/nginx/conf.d
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

65. forráskód Frontend konténer létrehozása Nginx alapon

A konténerek elkészülte után telepíthető a rendszer Kubernetes-re vagy akár Dockerben is futtatható.

3.4 Kubernetes alapú futtatás

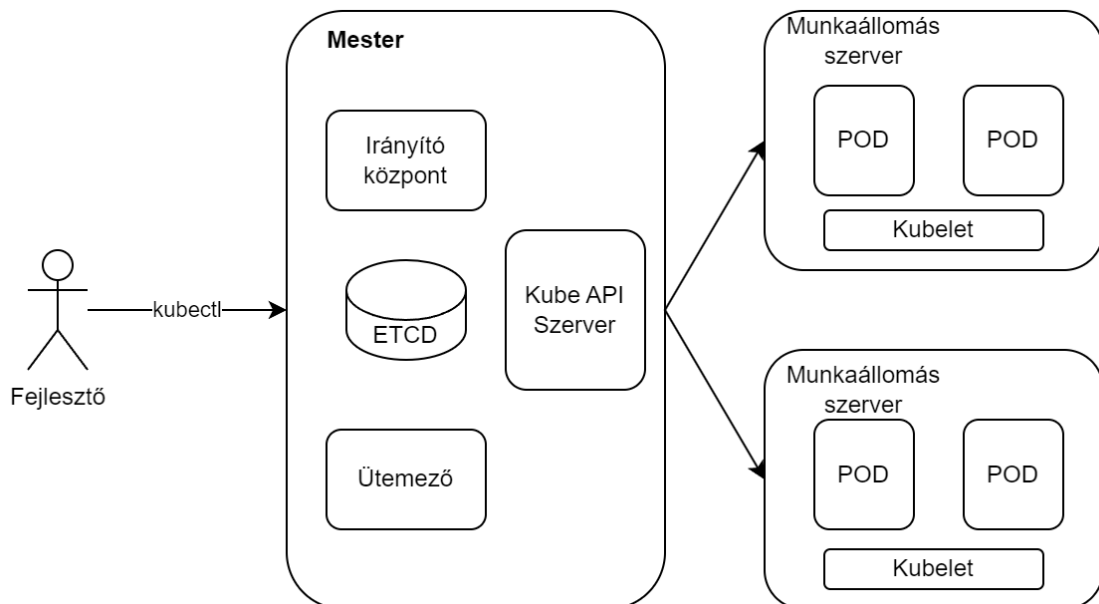
A rendszer futtatása Kubernetes – röviden k8s – platformon történik, melyet konténer alapú alkalmazások üzemeltetésére kifejlesztettek ki. Az alkalmazás telepítéséhez a Kubernetes parancssori eszközbe – mely neve kubectl - épített Kustomize értelmező használandó. Ez az általam definiált leíró fájlokból előállítja a Kubernetes erőforrás leíró fájlokat és ezeket az erőforrás objektumokat létrehozza a k8s klaszteren.

A platform nagy előnye, hogy az alkalmazások üzemeltetését a leírófájlok alapján automatikusan végzi. Elindítja a konténereket és amennyiben valamilyen oknál fogva egy konténer működésében hiba lép fel, új példányt indít belőle. Létrehozza a hálózati végpontokat, tárhelyeket. Biztosítja az automatikus skálázást, erőforrás menedzselést és sok egyebet. Az alkalmazás szempontjából előnyös a rolling update képessége. Amikor új alkalmazás verzió települ, az előző verziót csak akkor állítja le, amikor az új már hálózatba van kötve és kész a kérések kiszolgálására.

Az alkalmazás elvárt állapotát leíró fájlokban lehet deklaratíván megfogalmazni.

3.4.1 Kubernetes alapok

A k8s master-worker alapú architektúrát alkalmaz. A master node adja ki a telepítési és egyéb parancsokat és monitorozza a rendszert, valamint reagál az eseményekre.



41. ábra- Kubernetes architektúra

A 41. ábra egyszerűsítve mutatja be a platform felépítését. A fejlesztő a kubectl parancssori eszközzel tud kommunikálni a platformmal. Az API szerver fogadja és dolgozza fel ezen kéréseket. Az etcd egy elosztott kulcs-érték tároló, mely tárolja a klaszter elvárt és aktuális állapotát. Az ütemező osztja ki a feladatokat a worker-ek között, míg az controller monitorozza a klaszter állapotát és amennyiben az eltér a kívánttól, beavatkozik. A master node felelős továbbá a konténerek és azok erőforrásainak kezeléséért. Például a jobb erőforrás kiosztás érdekében futó POD-okat helyezhet át a worker-ek között. A worker node-okon futnak az alkalmazások, a

legkisebb telepítési egység a POD, mely akár több konténert is tartalmazhat. A POD-hoz tartozó erőforrások megoszlának a benne futó konténerek között. A kubelet a worker-ön futó POD-ok felügyeletét végzi és erről jelentést küld a controller-nek.

3.4.2 Telepítés leírók

A telepítés leírók a deployment könyvtárban található. Az alkalmazáshoz 3 környezetet hoztam létre.

A dev környezet csak az alkalmazás futtatásához feltétlen szükséges konténereket tartalmazza, illetve nem tartalmaz tartós adattárat, emiatt minden újraindításnál a program alapállapotról indul. Előnye, hogy kisebb az erőforrás igénye.

A prod és prod-win-wsl környezetek az alkalmazás mellett a monitorozáshoz szükséges programokat is futtatják, valamint tartós adattárat is biztosítanak.

A base könyvtár tartalmazza a telepítendő objektumok általános leírását, melyeket a customization.yml leíró foglal egységbe. Ebben a fájlban kell felsorolni a telepítendő erőforrások leíróit, illetve a képfájlokat verziószámmal. A korábban említett automatikus építés ezt a verziószámot írja felül a megfelelő szolgáltatáshoz.

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- api-gateway/deployment.yaml
- api-gateway/cluster-ip.yaml
images:
- name: zalerix/thesis-api-gateway:latest
  newName: zalerix/thesis-api-gateway
  newTag: 1.33.0
```

66. forráskód Kustomization erőforrás és képfájl leírás minta

Minden ms-hez több erőforrás leírófájl tartozik, melyek különböző célt szolgálnak. A következő bekezdésekben ezeket ismertetem egy-egy példa segítségével.

A deployment.yml fájlban a konténerből készülő POD-ot definiáltam. Itt adható meg az erőforrás címkéje, melyet a k8s a többi erőforrással történő összekötésre használ fel. Ezek a labels bejegyzés alatti kulcs-érték párok. Amikor erőforrásokat kell összekötni, azt ezekre a címkékre hivatkozva lehet megtenni. A 67. forráskód egy ilyen címkézést mutat

be. Ez a telepítés leíró egy POD definícióját tartalmazza, mely címkéje a `name: api-gateway` kulcs-érték pár. Az összes POD, amely ebből a leíróból készül, megkapja ezt a címkét. Emellett létrejön egy Deployment objektum is `api-gateway-deployment` néven. Ennek az objektumnak a `selector` mezője a POD címkéjére való illeszkedést írja le. Így az összes POD, ami ilyen címkével készül, bekerül ez alá a Deployment objektum alá. Ennek segítségével ellenőrizhető például, hány példány fut az adott POD-ból.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-gateway-deployment
spec:
  selector:
    matchLabels:
      name: api-gateway
  template:
    metadata:
      labels:
        name: api-gateway
```

67. forráskód Kubernetes címkék

Ezután a konténerek specifikációja következik. A Tudástérnél minden POD egy konténert futtat, mely az aktuális ms. Minden ms-hez készült Deployment leíróban beállítottam a használni kívánt képfájlt a latest verzióra, amely verziót a fent említett `kustomization.yaml` fájlban írtam felül, így az itt szereplő érték nem releváns a telepítés szempontjából. Ezután a liveness és readiness probe-ok leírása következik. A k8s az itt megadott URL-eken keresztül monitorozza az alkalmazást. A liveness probe segítségével vizsgálja, hogy az alkalmazás elindult-e. Amennyiben ezen a vizsgálaton a küszöbértéknél – alapesetben három – többször elbukik a kérés egymás után, új POD kerül létrehozásra. A readiness probe azt vizsgálja, hogy az alkalmazás képes-e kiszolgálni a kéréseket, a k8s csak akkor terel forgalmat az adott POD-ra, amikor ez a végpont HTTP 200-as válasszal tér vissza. Bármikor, amikor a vizsgálat elbukik, a k8s lekapcsolja a POD-ot a hálózatról és nem terel több forgalmat rá. Ezekhez az alkalmazás 9000-es portját használtam, ahol a Spring keretrendszer biztosítja a végpontokat és automatikusan beköti az adatbázisok és egyéb külső rendszerek megfigyelését is. Leállításnál beállítottam továbbá 5 másodperc várakozást. Erre az alábbiak miatt van szükség: a Spring alapú ms-ek leállításnál már nem fogadnak újabb kérést, de a még folyamatban lévőket kiszolgálják. A k8s SIGTERM eseménynél párhuzamosan törli a

POD-ot, valamint frissíti a hálózati beállításokat és nem tudható, melyik történik meg előbb. Emiatt előfordulhat, hogy a forgalmat olyan POD-hoz tereli, ami már leállítás alatt van és bár a POD-hoz beérkezik, az alkalmazás már elutasítja a kérést. Ilyen esetben ez 500 Internal Server Error válasszal tér vissza. Az említett beállítással a POD 5 másodperc késéssel kapja meg a SIGTERM eseményt. A hálózati beállítások már frissülnek addigra, így új forgalom nem érkezik a POD-hoz és megkezdődhet az alkalmazás leállítása, amely a folyamatban lévő kéréseket még kiszolgálja. A k8s SIGKILL eseményt küld, amennyiben a POD a SIGTERM esemény után 30 másodperccel sem áll le. Ezek konfigurációját mutatja be a 68. forráskód.

```
containers:
- name: api-gateway
  image: zalerix/thesis-api-gateway:latest
  livenessProbe:
    initialDelaySeconds: 10
    httpGet:
      port: 9000
      path: /actuator/health/liveness
  readinessProbe:
    initialDelaySeconds: 10
    httpGet:
      port: 9000
      path: /actuator/health/readiness
  lifecycle:
    preStop:
      exec:
        command: [ "sh", "-c", "sleep 5" ]
```

68. forráskód Képfájl, probeok és életciklus esemény leírása

A telepítés leíróban adtam meg továbbá a konténer által használható hardveres erőforrásokat. Ez két részből áll. A requests struktúra a POD-nak garantált erőforrásokat biztosít, míg a limits a maximálisan használható erőforrásokat írja le. Amennyiben nincs elegendő erőforrás a requests-ben igényelt érték biztosítására, a POD hibás állapotba kerül és nem indul el. A hibaüzenet a 69. forráskód mintában látható.

```
0/1 nodes are available: 1 Insufficient memory.
preemption: 0/1 nodes are available: 1
No preemption victims found for incoming pod.
```

69. forráskód Elégtelen memória mennyiség

A garantált erőforráson felüli erőforrásokat a k8s bármikor kioszthatja más POD-nak. Ez memória esetén problémákat okozhat JVM alapú alkalmazásoknál, erről részletesebben a 3.4.3 fejezetben írok.

A 70. forráskód példában a POD garantáltan kap 1 gigabájt memóriát és 300 millicpu-t. A millicpu a k8s által használt mértékegység, ahol 1000 millicpu jelöl egy virtuális processzormagot. Ezen felül további 700 millicpu erőforrást használhat szükség esetén, amennyiben ez rendelkezésre áll.

```
resources:
  limits:
    cpu: "1000m"
    memory: "1Gi"
  requests:
    cpu: "300m"
    memory: "1Gi"
```

70. forráskód Memória requests és limits

A konténer alapesetben nem érhető el a POD-on kívülről. Ahhoz, hogy az ms hívható legyen k8s hálózathoz, a használni kívánt portokat engedélyezni kell. Az ms-ek a 8080-as porton szolgáltatnak, illetve a 9000-es porton biztosítanak adminisztrációs API-t.

```
ports:
  - containerPort: 8080
    name: app-port
  - containerPort: 9000
    name: actuator-port
```

71. forráskód Konténer portok beállítása

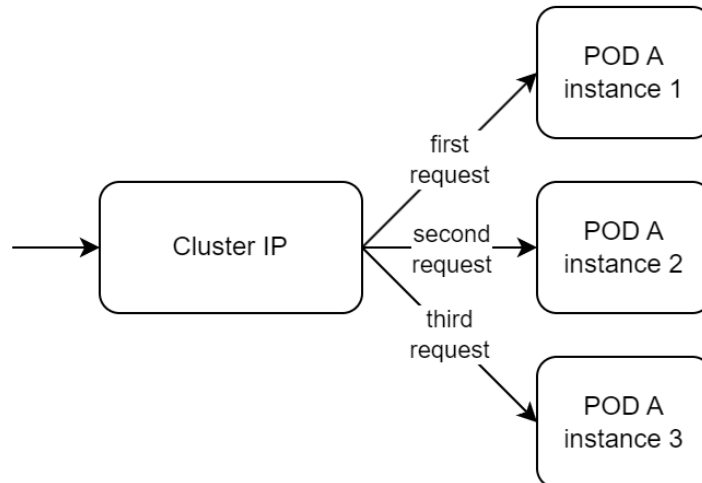
Utolsó lépésként pedig a környezeti változók beállítása következik. Ehhez a k8s által biztosított ConfigMap és Secret objektumokat hoztam létre minden környezethez, melyeket linkelek a POD-hoz. Az ms konténerébe indítás előtt az ezekben található kulcs-érték párok automatikusan beállításra kerülnek, így amikor az ms elindul, már látja a környezeti változókat.

```
envFrom:
  - configMapRef:
      name: common-properties
  - secretRef:
      name: common-secrets
```

72. forráskód Környezeti változók hivatkozásának beállítása

Ezzel az ms-t tartalmazó POD-ok akár futtathatók is, de k8s-en kívülről nem érhetők el, illetve a klaszteren belül is csak IP cím alapján lehet elérni őket. Ez nem túl szerencsés, mivel a k8s bármikor áthelyezhet egy POD-ot másik klaszterre a jobb erőforrás elosztás

érdekében, új példányokat hozhat létre terheléstől függően, vagy hiba esetén újraindíthatja azt, emiatt az IP cím folyamatosan változik. Erre kínál megoldást a ClusterIP objektum, mely terheléelosztóként működik az azonos címkéjű POD-ok előtt. Ez automatikusan nyilvántartja a példányok IP címét, valamint a terhelést is egyenletesen osztja el közöttük ahogy a 42. ábra mutatja.



42. ábra ClusterIP objektum

Az ClusterIP metaadatként megadott neve URL-ként funkcionál. Az API Gateway beállítás fájljában ezeket a címeket használtam a kérés továbbításokhoz. A selector mezőben az előzőleg tárgyalt címkét állítottam be, így minden ilyen címkével ellátott POD ugyanaz alá a ClusterIP alá kerül. Meg kell adni továbbá a portokat, amelyen a ClusterIP hívható és a cél portot, amelyre a hívást irányítja. Az egyszerűség kedvéért ugyanazt a portot használtam, mint az alkalmazás portja, ahogy a 73. forráskód példában is látható.

```
apiVersion: v1
kind: Service
metadata:
  name: subject-service-url
spec:
  type: ClusterIP
  ports:
    - name: app-port
      port: 8080
      targetPort: 8080
    - name: monitoring-port
      port: 9000
      targetPort: 9000
  selector:
    name: subject-service
```

73. forráskód ClusterIP létrehozása

Utolsó lépésként az automatikus skálázásért felelős Horizontal Pod Autoscaler (HPA) objektumot mutatom be. Itt írható le, hogy mikor történjen az alkalmazás skálázása. A Tudástér esetében csak CPU és memória alapú metrikákat használok. Amikor ezek használata a megadott határértéknél nagyobb, új POD kerül elindításra. Amennyiben tartósan alatta van minden példány, a rendszer automatikusan leállítja a felesleges POD-okat. Ez hatékonyabb erőforrás kihasználást eredményez. Lehetőség van a minimum és maximum példányszám beállítására, mely alá, illetve fölé nem skáláz a rendszer. A HPA nem értelmezhető POD-onként, hisz szükség van a példányszámra is, így ezt a megfelelő Deployment objektumhoz kell csatolni.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: subject-service-hpa
spec:
  minReplicas: 1
  maxReplicas: 4
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: subject-service-deployment
  metrics:
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: 75
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 80
```

74. forráskód HPA létrehozása

A HPA - kiegészítők segítségével – lehetőséget biztosít bármely metrika használatára a skálázáshoz, illetve időpont alapon is megtehető ez. Egy oktatási intézménynél például a várható legnagyobb terhelés 8-16 óra között van, így a kritikus komponensek ebben az időintervallumban magasabb példányszámban futhatnak.

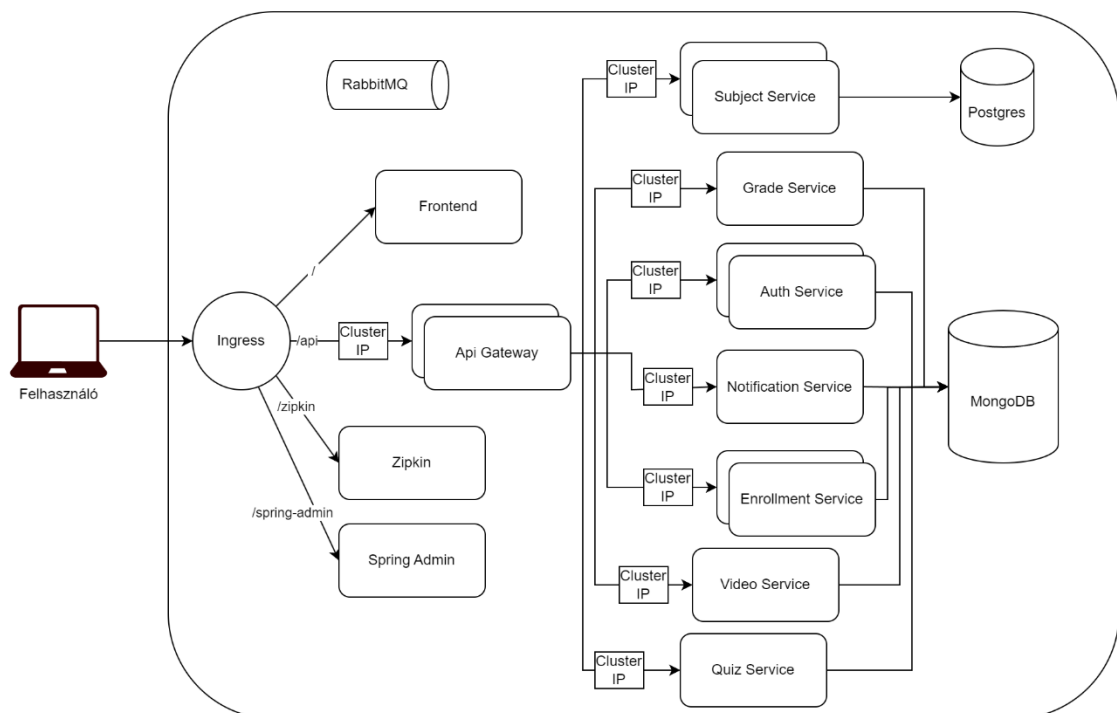
Az Tudástér használatához az API Gateway-t és a frontendet a k8s klaszteren kívülről is elérhetővé kell tenni - emellett további diagnosztikai eszközök is elérhetők, ezekről a 3.5 fejezetben írok. Ehhez az Ingress controllert [8] használtam, mely a k8s hálózat felé kinyitja - alapesetben - a 80-as portot, valamint a forgalmat a megfogalmazott szabályok

szerint a megfelelő ClusterIP-hoz irányítja. Az Ingress útvonalaknak egyedinek kell lenni. A 75. forráskód példában látható, hogy minden /api kezdetű kérés az API Gateway ClusterIP végpontjára kerül továbbításra a kezdőtagot levágása után. A kérések maximum 2Gb méretűek lehetnek. Ezeket Ingress annotációkkal oldottam meg.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-to-gateway
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
    nginx.ingress.kubernetes.io/proxy-body-size: 2000m
spec:
  rules:
    - http:
        paths:
          - path: /api(/|$)(.*)
            pathType: Prefix
            backend:
              service:
                name: api-gateway-url
                port:
                  number: 8080
  ingressClassName: nginx
```

75. forráskód Ingress controller létrehozása

A 43. ábra az alkalmazás architektúrát mutatja be k8s alatt. Az egyszerűség kedvéért az ms-ek közötti kapcsolatokat elhagytam.



43. ábra Tudástér Kubernetesen

A components könyvtárban a futtatáshoz szükséges adatbázisok és egyéb szoftverek leírói foglalnak helyet, melyeket adott környezethez dinamikusan csatolok szükség esetén.

Az overlays könyvtárban található a környezetek leírói, ezek lehetnek patch-ek, új erőforrások vagy konfigurációs fájlok. A környezetek a base mappában található leírók alapján épülnek fel, és azok módosításait fogalmazzák meg. A Kustomize yaml útvonalak alapján felülírja az eredeti fájlok megadott részeit és az így előállt végleges leíró fájlt küldi el a k8s felé telepítésre.

Példaként a prod környezetet mutatom be, ami ugyanezen nevű mappában található.

A környezet összeállítása a customization.yaml fájlban található. Ez foglalja össze, mely erőforrásokra van szükség, milyen komponensek kerüljenek telepítésre, valamint mely leírókat kell módosítani. Itt kapott helyet továbbá a ConfigMap és Secret objektum generálása is a könyvtárban található property fájlokból. A prod környezetben az adatbázisokhoz és alkalmazásokhoz virtuális meghajtót csatoltam az adatok tárolásához, melyek így túlélnek a rendszer újraindítását. Ehhez két k8s objektumra van szükség.

A PersistentVolume definiálja a tárhelyet és becsatolja azt a k8s klaszterbe. Ez lehet lokális lemez, – mint a Tudástér esetében – de akár hálózati vagy felhő alapú meghajtó is. Ennek leírófájlja látható a 76. forráskód példában.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: video-store-pv
spec:
  storageClassName: manual
  capacity:
    storage: 15Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/data/nowhere_store/video"
```

76. forráskód Lokális mappa becsatolása virtuális meghajtóként

A létrehozott meghajtóból a PersistentVolumeClaim objektum segítségével foglalható tárhely az alkalmazásnak.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: video-store-pvc
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 15Gi
```

77. forráskód Tárhely foglalása az alkalmazáshoz

Melyet ezután a Deployment leíró patch fájlban csatoltam a POD-okhoz. Ebben elég csak a megváltoztatni vagy hozzáadni kívánt struktúrát definiálni, a többi a base leírókból kerül beemelésre a végleges leíróba.

```
spec:
  containers:
    - name: video-service
      volumeMounts:
        - mountPath: /data
          name: video-store
  volumes:
    - name: video-store
      persistentVolumeClaim:
        claimName: video-store-pvc
```

78. forráskód Tárhely hivatkozás hozzáadása a Deployment leíróhoz

Az ms-ek a /data könyvtárba mentik a videókat, képeket, így erre az útvonalra csatolom fel a meghajtót.

A ConfigMap létrehozása a környezet mappájában lévő environments.properties fájlból történik. A Kustomize képes kezelni ezt a kulcs-érték párokat tartalmazó fájlt, így az objektum létrehozása egy egyszerű leíróval megadható ahogy a 79. forráskódban látható.

```
configMapGenerator:
  - name: common-properties
    envs:
      - environments.properties
```

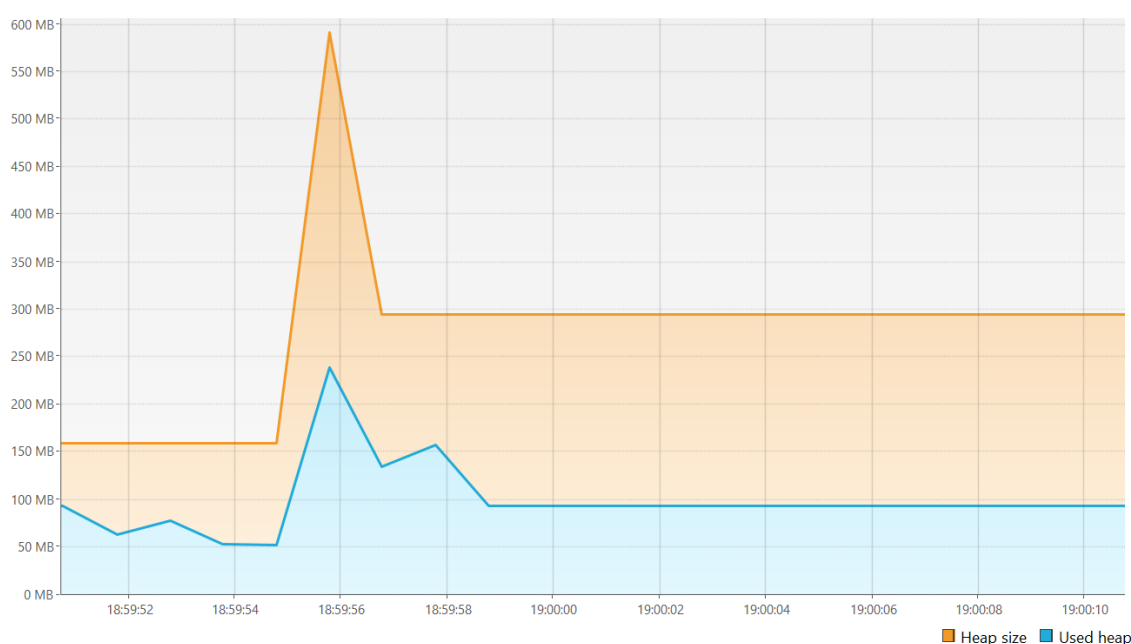
79. forráskód ConfigMap generálása

Ezután a telepítés a 2.3 fejezetben ismertetett paranccsal indítható.

A könyvtár további telepítési módokat is tartalmaz, melyet a 3.9 fejezetben mutatok be.

3.4.3 JVM és Kubernetes

Az alkalmazás erőforrásigénye k8s klaszteren nagy. Ez a probléma a virtuális gép alapú nyelvek és a Kubernetes működése közötti különbségekből adódik. A JVM a program indításakor betölti és értelmezi az osztályokat. Ennek memória - és számításigénye magas. A 44. ábra által bemutatott esetben egy ms indítási memóriaigényét mértem, ezen jól megfigyelhető a probléma. A program indulásakor az erőforrásigény megnő, majd amikor az alkalmazás betöltötte a szükséges osztályokat, lecsökken. A Java alapú szerverek a vertikális skálázást részesítik előnyben. Nagyobb terhelés esetén több memória és processzor hozzáadásával nagyobb áteresztőképesség érhető el.



44. ábra Java alkalmazás indításának memóriaképe

Ez a skálázási módszer viszont limitált képességekkel rendelkezik. Egy számítógépbe egy idő után nem lehet több erőforrást rakni. A k8s paradigma a kis CPU és memóriaigényű, gyorsan indítható alkalmazásokat részesíti előnyben, melyekből több példányt futtat, hogy nagyobb áteresztőképességet érjen el, mint egy vertikálisan skálázott rendszer.

Az előző fejezetben említett memória requests és limits beállításával megadható a POD által garantált és maximálisan használható memória. Viszont a garantált feletti részt a k8s controller bármikor elveheti az alkalmazástól, vagy megakadályozhatja, hogy az egyáltalán hozzáférjen. JVM esetén, ha ez indítási időben történik, akkor amennyiben a garantált memória nem elég az indításhoz, az alkalmazás Out of Memory - OOMKill - hibával leáll és a k8s új példányt hoz létre. Amennyiben az ilyen típusú leállások elérnek

egy meghatározott határértéket, a k8s hibás státuszba helyezi a Deployment-et. Spring alapú alkalmazások adatbáziskapcsolat felépítéssel, migrációval és egyéb gyakran használt funkcionalitással körülbelül 600 megabájt memóriát igényelnek indításkor. Indítás után viszont akár harmadannyival is üzemelni tudnak. Az indítás miatt viszont szükséges a requests méretét is erre beállítani, elkerülve, hogy a k8s visszavegye a memóriát az alkalmazás konténerétől. Emellett, ha meg is kapja indításnál a garantált feletti részt, az indítás után a JVM heap nem használt része nem kerül vissza az operációs rendszerhez azonnal, csak az következő teljes GC futásnál. Ha a kettő között a k8s visszaveszi a memóriát, az alkalmazás leáll. Utóbbi problémára a Shenandoah GC megoldást kínálja, mert minden GC futásnál képes a heap memóriát összehúzni és visszaadni az OS-nek a nem használt részt.

Nagy hátránya emiatt a JVM alapú szoftvereknek ilyen környezetben, hogy magas erőforrásigénnyel rendelkeznek, ami bérelt szervereken -például AWS, Azure - költséges lehet.

CPU-nál probléma lehet, hogy kis rendelkezésre álló mennyiség esetén az alkalmazás lassabban indul. Ez két problémát okozhat. Az alkalmazás nem indul el időben, így a liveness probe HTTP 500-as választ ad, ami a POD újraindítását eredményezi, illetve skálázásnál, ha hirtelen kap nagy terhelést az ms, nem tud időben elindulni az újabb példány. Ezért érdemes limitben legalább a normál működéshez szükséges CPU fogyasztás kétszeresét beállítani.

A probléma megoldása a GraalVm lehet, mely képes a JVM alapú programokat natív gépi kódra fordítani, így az alkalmazás indítási ideje és erőforrásigénye a töredékére csökken. A Spring keretrendszer már részlegesen támogatja ezt.

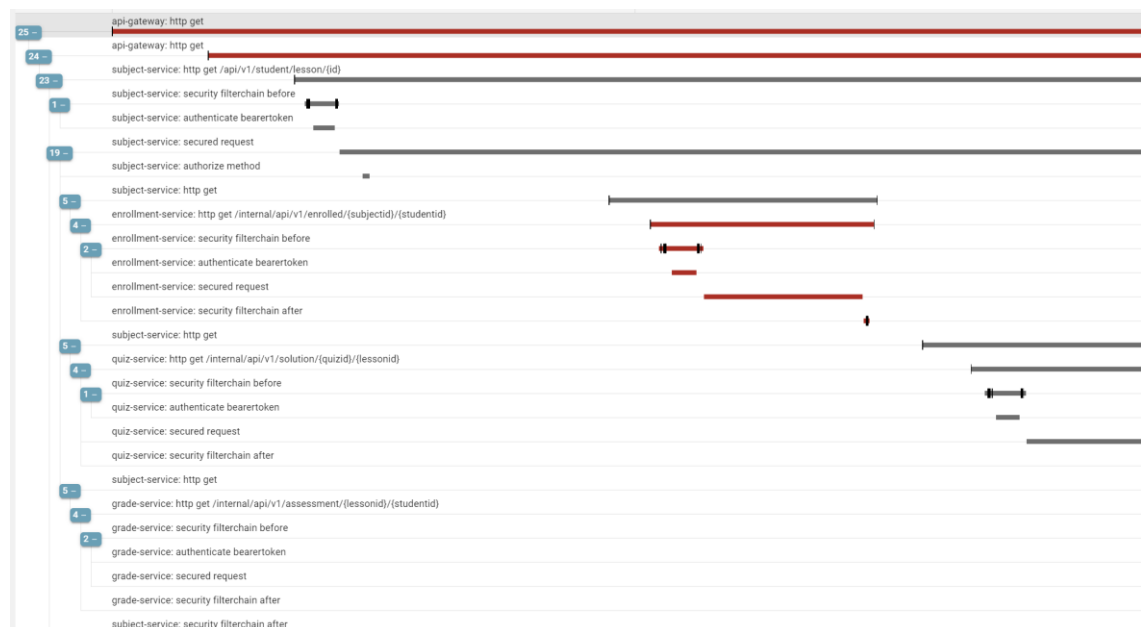
3.5 Az alkalmazás monitorozása

Mikroszolgáltatás alapú, automatikusan skálázott alkalmazások esetén is előfordulhatnak hibák. Ezen hibák felderítése nehéz, hisz egy kérés kiszolgálásában több szolgáltatás működik együtt, ahol minden ms több példányban futhat és a hiba bárhol előfordulhat. A Three Pillars of Observability [7] minta leírja, milyen lépések szükségesek ezen hibák felderítéséhez. Ez a loggolásból, monitorozásból és nyomkövetésből áll.

Az alkalmazás logbejegyzést készít a rendszerben bekövetkező fontosabb eseményekről, például a beérkező REST kérésről, Jobok indításáról és leállításáról, üzenetek küldéséről.

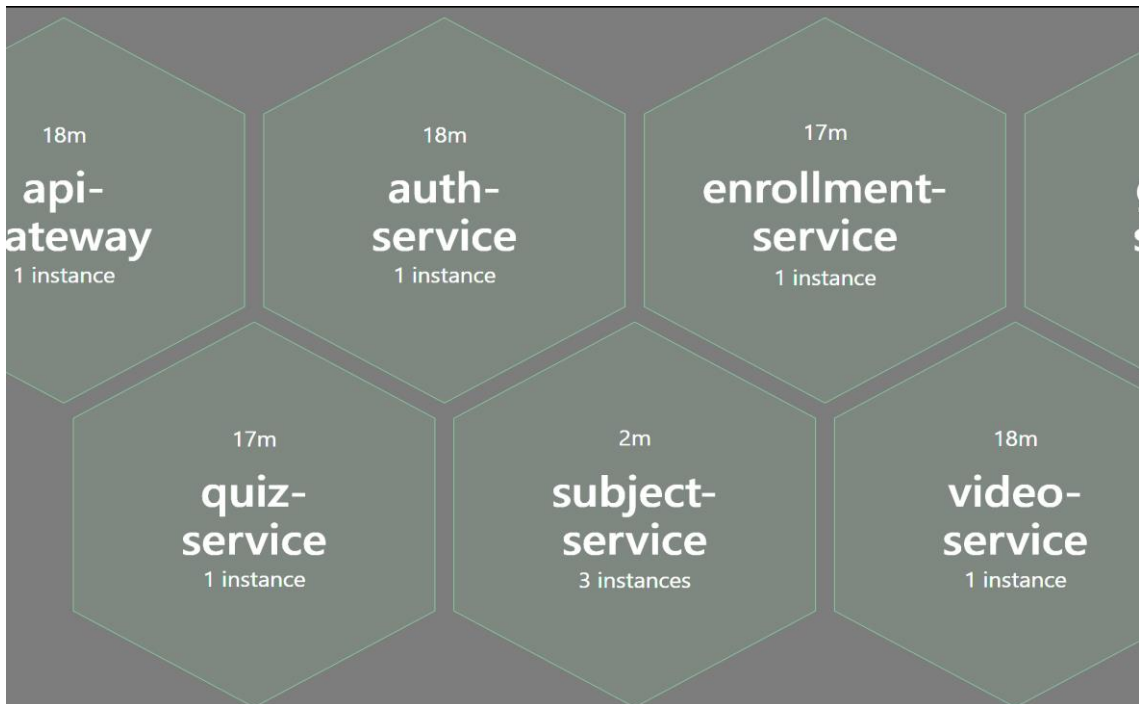
API Gateway a kérés fejlécét kiegészíti a nyomkövetésre használt trace id mezővel, mely az ms-ek közötti hívások esetén továbbításra kerül. Ez bekerül a logbejegyzésekbe, így ennek alapján szűrve az egy kérés kiszolgálásához tartozó összes log összegyűjthető. A metrikák és nyomkövetés megvalósítására a Micrometer által fejlesztett könyvtárat használtam, mely automatikusan összegyűjti és publikálja a fontosabb mérési adatokat. Ezek feldolgozására sokféle eszköz létezik. A Tudástér a Spring Admin programot telepíti a mérések megjelenítéséhez, míg nyomkövetéshez Zipkint. Ezek alaptudású eszközök, de az erőforrásigényük minimális.

Zipkin segítségével ábrázolható az ms-ek közötti hívási lánc grafikusán is. A 45. ábra által bemutatott képen látható, hogy a kérés belépési pontja a rendszerbe az Api Gateway, majd egymás alatt a többi szolgáltatás hívása jelenik meg időrendi sorrendben. A hívások mellett a feldolgozási idő is látható. Természetesen, ha egy szolgáltatás meghív egy másikat, akkor annak válaszát meg kell várnia, így beleszámít a futásidejébe. A grafikon alapján látható, a rendszer mely feldolgozással tölt sok időt. Így könnyebben megtalálható a rendszer szűk keresztmetszete. Az ms-eken belül tovább bontható a grafikon úgynevezett span id-k használatával. Ennek segítségével különválasztható például az üzleti logika, adatbázis hívás és egyéb funkcionalitások.



45. ábra Zipkin

A Spring Admin összegző képernyőjén látható az összes futó szolgáltatás, valamint ezek példányszáma.



46. ábra- Mikroszolgáltatások összegző képernyő

Továbbá az egyes szolgáltatásokhoz megjeleníthetők a kívánt metrikák, például kérések száma és átlagos kiszolgálási ideje, processzor - és memóriahasználat.

http.client.requests		COUNT	Integer	TOTAL_TIME	Milliseconds	MAX	Milliseconds
application:subject-service				132	895 ms	95 ms	
http.server.requests		COUNT		TOTAL_TIME	Milliseconds	MAX	Milliseconds
(no tags)				63	4073 ms	1193 ms	
jvm.memory.used				VALUE			
(no tags)				324867920			

47. ábra Metrikák

3.6 Tesztelés

A szoftverhibák kiszűrésének, és a stabil működés biztosításának érdekében a szoftvert több szinten teszteltem.

Az első szint az egységtesztelés. Ez a ms-ek üzleti logikai részét fedi le, ahol előre elkészített adatokkal megvizsgáltam a tesztelni kívánt metódus működését. Az egységtesztetekhez az adatbázis és külső rendszer hívásokat helyettes – másnéven mock - hívásokra cseréltem. A mock hívásokat a tesztesetek előtt definiáltam. A tesztek JUnit és Mockito teszt keretrendszerekre épülnek.

A 80. forráskódban a GradeRepository a felhasználót betöltő adatbázis hívás esetén az előre elkészített objektumot adja vissza.

```
private val repository: GradeRepository = mock() {
    on { findAllByUserId("1") } doReturn
        listOf(Grade.withTestData())
    on { save(any<Grade>()) } doReturn Grade.withTestData()
}
```

80. forráskód Adatbázis hívás mock

Ezután az egyes használati esetekre – valamint hibás esetekre is - elkészítettem a teszteket. A 81. forráskód példában az adott tantárgyhoz tartozó jegyek betöltését vizsgáltam. A Service réteget meghívva megnéztem, hogy az eredmény helyes-e, illetve, hogy a metódus a megfelelő szolgáltatásokat és külső rendszereket hívta-e meg a bemenő adat függvényében. Azt is vizsgáltam, további hívás nem érkezett-e ezekre a rendszerekre.

```
@Test
fun `should return grades for the subject`() {

    val result = sut.getGradesBySubjectId("1")
    assertThat(result).hasSize(1)
    val user = result.first()
    assertThat(user.mark).isEqualTo(5)
    assertThat(user.points).isEqualTo(25)
    assertThat(user.studentName).isEqualTo("Joe Doe")

    verify(repository).findAllBySubjectId("1", "1")
    verify(authIntegration).getUsers(listOf("1"))
    verifyAllNoMoreInteraction()
}
```

81. forráskód Tantárgyhoz tartozó jegyek betöltése

A végpont hívások, illetve külső rendszerekkel történő kommunikáció vizsgálatára integrációs tesztek készültek.

A végpontok tesztelésénél szükséges a felhasználó jogosultságának tesztelése is. Ehhez létrehoztam három annotációt, melyek segítségével a rendszer a teszt futtatása előtt a SecurityContextHolder-be helyez egy előre legyártott JWT tokent a kívánt felhasználói jogosultsággal, ahogy a 82. forráskód példában látható.

```
@JwtUser(TEST_STUDENT_TOKEN, Role.ROLE_STUDENT)
@Retention(AnnotationRetention.RUNTIME)
annotation class WithStudent

@Component
class WithBearerTokenSecurityFactory(
    private val jwtDecoder: JwtDecoder
) : WithSecurityContextFactory<JwtUser> {

    override fun createSecurityContext(annotation: JwtUser)
    : SecurityContext {
        return SecurityContextHolder.createEmptyContext().also {
            it.authentication = JwtAuthenticationToken(
                jwtDecoder.decode(annotation.token),
                listOf(SimpleGrantedAuthority(annotation.role.name))
            )
        }
    }
}
```

82. forráskód Token kontextusba helyezése

Ennek segítségével könnyen tesztelhető, hogy adott felhasználó hozzáfér, vagy sem a végponthoz. Például a diák jegyeihez nem férhet hozzá a tanár. A 83. forráskód által bemutatott tesztben a diákok által elérhető végpontot tanár jogkörű felhasználóval hívtam meg és megvizsgáltam, megfelelő státusszal tér-e vissza a kérés.

```
@Test
@WithTeacher
fun `should be forbidden if user is not student`() {
    mockMvc.get("/api/v1")
        .andExpect {
            status { isForbidden() }
        }
}
```

83. forráskód Tiltott hozzáférés tesztelése

Ez a végpont a diákoknak elérhető és a válaszban az elvárt objektumot kell visszakapnia a tanulóknak. Ezért a státusz mellett a választ is megvizsgáltam.

```

@Test
@WithStudent
fun `should return with the student grades`() {
    whenever(gradeService.getMyGrades()) doReturn
    listOf(GradeResponse.withTestData())

    mockMvc.get("/api/v1")
        .andExpect {
            status { isOk() }
            content {
                objectMapper.writeValueAsString(
                    listOf(GradeResponse.withTestData())
                )
            }
        }
}

```

84. forráskód Diákok jegyeinek lekérése teszt

Az integrációs teszt elindítja az alkalmazást és valós kérések segítségével vizsgálja a lekérdezés működőképességét. A külső rendszerek, mint például RabbitMQ, MongoDB integrációs tesztjei TestContainers segítségével készültek, mely a teszt indítása előtt Docker konténerben elindítja a külső szolgáltatást, majd az alkalmazás beállításait felülírja a konténerek kapcsolódási adataival, így az alkalmazás csatlakozni tud hozzá és a tesztek valós környezetben futnak. A tesztek után a konténerek törlésre kerülnek. Ennek összeállítása a TestUtils osztályban található. Üzenetsor alapú funkciók tesztelése esetén az üzenetek nem érkeznek meg azonnal, ezért ezek a tesztek az eredmény vizsgálata előtt várakoznak.

A többi ms felé történő hívások teszteléséhez a Wiremock szolgáltatást használtam, mely segítségével programozottan lehet hívható végpontokat létrehozni és megadni az elvárt választ. Az alkalmazás a valós végpont helyett ezt hívja meg a teszt futtatása során, így megvizsgálható, sikeres a kifelé történő kommunikáció. A 85. forráskód egy ilyen helyettes végpontot mutat be.

Ezeknek a technológiáknak a segítségével az ms-ek összes unit-, és integrációs teszt esete lefedhető. Az integrációs tesztek futtatásához telepített Docker szükséges. A tesztek a `gradle test` paranccsal indíthatók.

```

@Test
fun `subject api should provide subjects`() {
    val expectedSubjects = listOf(Subject.withTestData())
    stubFor {
        get(urlEqualTo("/internal/api/v1?subjectIds=1"))
            .willReturn(
                aResponse()
                    .withStatus(200)
                    .withHeader(
                        "Content-Type",
                        "application/json"
                    ).withBody(
                        objectMapper.writeValueAsString(
                            expectedSubjects
                        )
                    )
            )
    }

    val subjects = subjectApi.getSubjects(listOf("1"))

    assertThat(subjects).isNotEmpty
    assertThat(subjects).isEqualTo(expectedSubjects)
}

```

85. forráskód Wiremock stub létrehozása és az azt hívó API tesztelése

A tesztek futtatásától jegyzőkönyv készül, mely tartalmazza az egyes osztályok lefedettségét, valamint összegzést is. Az alábbi ábrán a Grade Service teszt lefedettsége látható.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.zlrx.thesis.gradeservice.service		93%		57%	18	43	7	95	6	29	0	6
com.zlrx.thesis.gradeservice.api.auth		60%		n/a	4	9	3	11	4	9	2	4
com.zlrx.thesis.gradeservice.api.subject		54%		n/a	4	8	3	10	4	8	2	4
com.zlrx.thesis.gradeservice.domain		88%		50%	7	34	0	40	6	33	0	4
com.zlrx.thesis.gradeservice.stream		95%		75%	4	31	1	76	2	27	2	8
com.zlrx.thesis.gradeservice		15%		n/a	1	2	3	5	1	2	1	2
com.zlrx.thesis.gradeservice.controller		100%		100%	0	8	0	18	0	7	0	2
com.zlrx.thesis.gradeservice.controller.model		100%		n/a	0	14	0	14	0	14	0	3
Total	159 of 1 564	89%	15 of 40	62%	38	149	17	269	23	129	7	33

48. ábra Grade Service teszt lefedettség

A teljes rendszer együttműködésének tesztelése nem lehetséges ezekkel a módszerekkel, illetve a rendszer teljesítménye sem mérhető vele. Ezért ennek megvalósítására a K6 teszt rendszert használtam. Segítségével meghívhatók a valós környezetben futó rendszer végpontjai, akár több szálon, több felhasználóval, illetve megadható az elvárt válasz. A tesztek futásának eredményéből a K6 egy összegzést készít, ahol látható a hívások száma, eredménye, a kérések válaszüzeje különböző statisztikai eloszlások szerint - például p50, p99.

Ezek a tesztek egy külön projectben találhatók a perf-test könyvtárban. Futtatásukhoz szükséges a Tudástér futtatása, valamint a K6 program telepítése. A tesztek parancssorból indíthatók, például a `k6 run student-get-subjects.js` parancs kiadásával a teszt könyvtárában.

A teszt két részből áll. A 86. forráskód példában beállítottam, hogy a tesztelendő végpontot összesen 50 virtuális felhasználóval – `vu`, többes számban `vus` – szeretném meghívni, 10 másodpercen keresztül. Az időtartam alatt, ha egy `vu` kérése befejeződik, új kérést küld.

```
export let options = {  
  vus: 50,  
  duration: '10s'  
}
```

86. forráskód K6 beállítások

Ezután a teszt funkcióban a végpont és az elvárt válasz eredményét írtam le. Az egy másodperc várakozás biztosítja, hogy egy `vu` egy másodpercet várakozzon két kérés között. Ennek csökkentésével, illetve növelésével lehet szimulálni gyors és lassú felhasználót, de az általános méréseknél a kérés/másodperc mennyiséget szokás használni.

```
export default function () {  
  const params = {  
    headers: {  
      'Authorization': 'token_for_call'  
    }  
  }  
  
  const res =  
    http.get(`${baseUrl}/subject/api/v1/student`, params);  
  
  check(res, {"status was 200": (r) => r.status === 200},  
    {'app': 'subject', 'user': 'student'});  
  
  sleep(1)  
}
```

87. forráskód K6 teszt

A 49. ábra ismerteti egy teszt futási eredményeinek kimenetét.

```
script: student-get-subjects.js
output: -

scenarios: (100.00%) 1 scenario, 50 max VUs, 40s max duration (incl. graceful stop):
  * default: 50 looping VUs for 10s (gracefulStop: 30s)

✓ status was 200

checks.....: 100.00% ✓ 1391      X 0
data_received.....: 891 kB  87 kB/s
data_sent.....: 570 kB  56 kB/s
http_req_blocked.....: avg=73.63µs min=0s      med=0s      max=3.29ms  p(90)=0s    p(95)=0s
http_req_connecting.....: avg=66.71µs min=0s      med=0s      max=3.29ms  p(90)=0s    p(95)=0s
http_req_duration.....: avg=162.38ms min=11.95ms med=114.11ms max=1.37s   p(90)=257.72ms p(95)=303.82ms
  { expected_response:true }...: avg=162.38ms min=11.95ms med=114.11ms max=1.37s   p(90)=257.72ms p(95)=303.82ms
http_req_failed.....: 0.00% ✓ 0      X 1391
http_req_receiving.....: avg=5.79ms min=0s      med=0s      max=190.53ms p(90)=6.42ms p(95)=43.46ms
http_req_sending.....: avg=14.5µs min=0s      med=0s      max=1.5ms    p(90)=0s    p(95)=0s
http_req_tls_handshaking.....: avg=0s min=0s      med=0s      max=0s       p(90)=0s    p(95)=0s
http_req_waiting.....: avg=156.57ms min=11.95ms med=111.59ms max=1.37s   p(90)=219.5ms p(95)=301.36ms
http_reqs.....: 1391 135.251296/s
iteration_duration.....: avg=365.42ms min=214.34ms med=317.27ms max=1.59s   p(90)=461.25ms p(95)=506.55ms
iterations.....: 1391 135.251296/s
vus.....: 50 min=50 max=50
vus_max.....: 50 min=50 max=50

running (10.3s), 00/50 VUs, 1391 complete and 0 interrupted iterations
default ✓ [=====] 50 VUs 10s
```

49. ábra K6 futás eredmények

A Kubernetes POD-ok skálázódásának teszteléséhez egy terheléses tesztet hoztam létre, mely új tantárgyakat szűr be az adatbázisba. A Subject Service maximális példányszámát ötre növeltem, de a minimum egy maradt, így a kiinduló helyzetben egy példány futott belőle.

A teszt beállításoknál a terhelés növekedését felosztottam több fázisra mely a 88. forráskód példában látható.

```
export let options = {
  stages: [
    {duration: '1m', target: '50'},
    {duration: '5m', target: '500'},
    {duration: '1m', target: '100'},
    {duration: '20s', target: '10'},
    {duration: '5s', target: '0'},
  ]
}
```

88. forráskód Terheléses teszt beállításai

A teszt az első egy percben nulláról fokozatosan eléri az 50 virtuális felhasználót. Ez megfelel 50 kérés/másodperc terhelésnek a korábban említett sleep funkció miatt. A következő fázisban fokozatosan – öt perc alatt – ötvenről 500 vus-re növekszik a felhasználók száma, majd egy perc alatt visszacsökken 100-ra, később 10-re végül 0-ra. A teljes tesztelés 7 perc 25 másodpercig tartott. Ezalatt 90083 kérés indult a rendszer felé. Ebből 89055 volt sikeres, azaz került új tantárgy az adatbázisba. 1028 esetben pedig sikertelen volt. Ennek oka, hogy az egyik POD-nál elfogyott a maximálisan beállított 1

gigabájt memória. A Subject Service példányszáma a kezdeti egyről fokozatosan ötre növekedett, ahogy az 50. ábra mutatja.

```
C:\Users\zlava
λ kubectl get deployment subject-service-deployment
NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
subject-service-deployment         1/1      1              1            34m

C:\Users\zlava
λ kubectl get deployment subject-service-deployment
NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
subject-service-deployment         5/5      5              5            45m
```

50. ábra Subject Service deployment példányszám változás terhelés alatt

A 51. ábra bemutatja a példányok indításának időpontját a POD-ok lekéréséhez viszonyítva.

```
subject-service-deployment-5f4c86d46b-jc92w      1/1      Running    0            4m4s
subject-service-deployment-5f4c86d46b-lv57r      1/1      Running    0            3m4s
subject-service-deployment-5f4c86d46b-pvctd      1/1      Running    0            4m4s
subject-service-deployment-5f4c86d46b-sgmbn      1/1      Running    0            47m
subject-service-deployment-5f4c86d46b-wd627      1/1      Running    0            5m4s
```

51. ábra Példányok indulásának időpontja

Illetve az 52. ábra bemutatja a maximális terhelés mellett használt erőforrásokat. Látható, hogy minden példány elérte a maximális memória használatot az 500 kérés/másodperc esetén.

```
subject-service-deployment-5f4c86d46b-jc92w      177m      1023Mi
subject-service-deployment-5f4c86d46b-lv57r      136m      1020Mi
subject-service-deployment-5f4c86d46b-pvctd      194m      1023Mi
subject-service-deployment-5f4c86d46b-sgmbn      206m      1020Mi
subject-service-deployment-5f4c86d46b-wd627      195m      1023Mi
```

52. ábra POD-ok terhelése

Végül pedig a teszt összefoglalását az 53. ábra mutatja be.

```
scenarios: (100.00%) 1 scenario, 500 max VUs, 7m55s max duration (incl. graceful stop):
* default: Up to 500 looping VUs for 7m25s over 5 stages (gracefulRampDown: 30s, gracefulStop: 30s)

X status was 201
  98% - ✓ 89055 / X 1028

checks.....: 98.85% ✓ 89055      X 1028
data_received.....: 59 MB  133 kB/s
data_sent.....: 52 MB  116 kB/s
http_req_blocked.....: avg=4.92µs  min=0s      med=0s      max=1.5ms   p(90)=0s    p(95)=0s
http_req_connecting.....: avg=2.69µs  min=0s      med=0s      max=1.5ms   p(90)=0s    p(95)=0s
http_req_duration.....: avg=177.69ms min=1.59ms  med=13.27ms max=30.11s  p(90)=101.03ms p(95)=208.08ms
{ expected_response:true }...: avg=53.19ms min=3.64ms  med=13.12ms max=16.22s  p(90)=99.83ms p(95)=199.49ms
http_req_failed.....: 1.14% ✓ 1028      X 89055
http_req_receiving.....: avg=397.33µs min=0s      med=0s      max=100.14ms p(90)=545.2µs p(95)=1.58ms
http_req_sending.....: avg=9.92µs  min=0s      med=0s      max=1.52ms   p(90)=0s    p(95)=0s
http_req_tls_handshaking.....: avg=0s      min=0s      med=0s      max=0s       p(90)=0s    p(95)=0s
http_req_waiting.....: avg=177.28ms min=1.59ms  med=12.87ms max=30.1s    p(90)=100.47ms p(95)=206.88ms
http_reqs.....: 90083  202.371748/s
iteration_duration.....: avg=1.17s   min=1s      med=1.01s   max=31.11s   p(90)=1.1s   p(95)=1.21s
iterations.....: 90076  202.356022/s
vus.....: 1      min=1      max=499
vus_max.....: 500  min=500    max=500
```

53. ábra Terheléses teszt összefoglalása

Az átlagos kiszolgálási idő 13 millisecondum volt. Ez annak köszönhető, hogy az adatbázis és az alkalmazás egy számítógépen futott, illetve a kérések is erről a gépről indultak, így nem volt hálózati késleltetés. Ettől függetlenül a rendszer reszponzivitását jól mutatja. A leglassabb kiszolgálás 30 másodperc volt, ez a hibás kérések időtúllépésének ideje. Az alkalmazás terhelhetőségéről jobban képet ad a p90 és p95 statisztika, mely 101, illetve 208 ms válaszütemet jelzett. Ez annyit jelent, hogy a kérések 90, illetve 95 százalékának kiszolgálása ennél az értéknél gyorsabb volt.

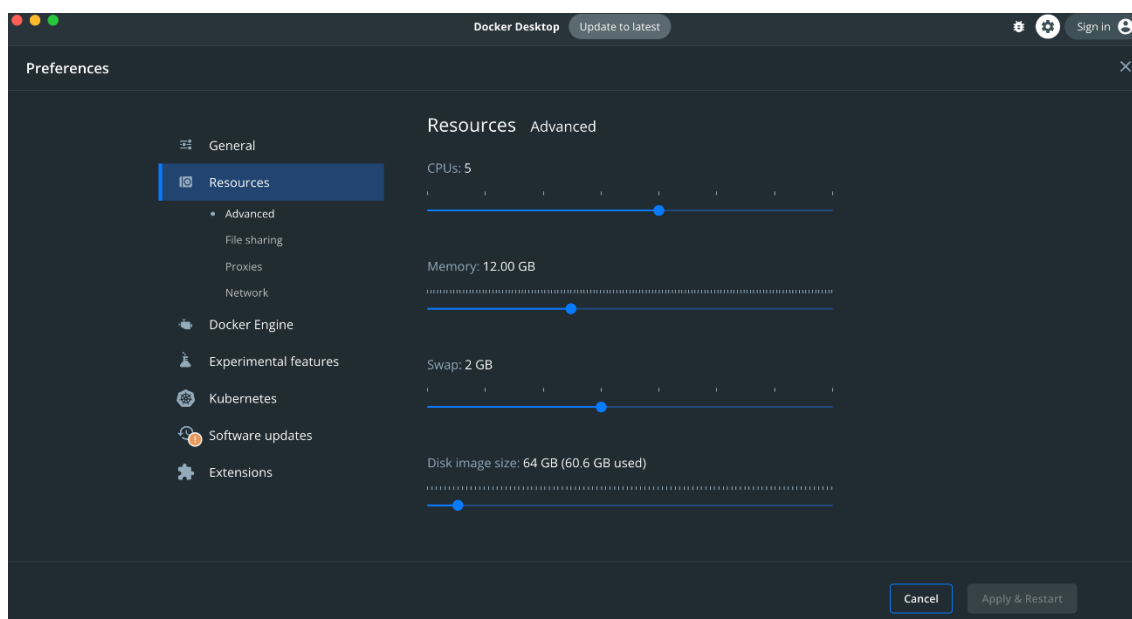
Az alkalmazást manuálisan is teszteltem. Így bizonyosodtam meg a frontend, és a teljes rendszer hibamentességéről. Adminisztrátor szerepkörrel a felhasználók kezelését vizsgáltam meg - úgy, mint felhasználók hozzáadása, törlése, valamint szerkesztése. Tanári szerepkörrel a tantárgyak felvételét és lekérdezését teszteltem, melybe beletartozik a kvízek összeállítása és a videók feltöltése is. A diákokkal pedig a tantárgyak felvételét, teljesítését, jegyek beírását, valamint az üzenetküldő rendszert vizsgáltam. Fontos, hogy az egyes intézmények felhasználói nem láthatják a többi intézmény adatait, ezért ezeket a vizsgálatokat több intézményt regisztrálva teszteltem.

3.7 Kubernetes telepítése

Több felhő szolgáltató is nyújt menedzselt Kubernetes klasztert. Azonban ezek szinte kizárólag előfizetői konstrukcióban érhetőek el. Lehetőség van azonban lokális telepítésre is. Ennek legegyszerűbb módja a [Docker Desktop](#) telepítése (minimum 4.17-es verzió), mely rendelkezik beépített Kubernetes-zel. Ennek nagy előnye, hogy elérhető a három legelterjedtebb operációs rendszerre – Linux, Mac, Windows.

A Docker telepítése és indítása után engedélyezni kell a Kubernetes klasztert. Ez a beállításokban, a Kubernetes menüpont alatt tehető meg az *Enable Kubernetes* jelölőnégyzet kiválasztásával, majd az *Apply & Restart* gomb megnyomásával. Legalább 1.26-os verzió szükséges a Tudástérhez.

Második lépés az erőforrás limitek beállítása. Linux és Mac esetén a beállítások alatt, a *Resources* menüpontban található csúszkák segítségével ez könnyedén megtehető.



54. ábra Kubernetes beállítások

Windows operációs rendszer esetében érdemes engedélyezni a [WSL2](#) alrendszert. Így a konténerek natív környezetben futhatnak. Enélkül a k8s egy virtuális Linuxon fut, ami teljesítmény problémákat okozhat.

Windows-on az Docker által elérhető erőforrások beállítása kissé bonyolultabb. A felhasználói fiók könyvtárában létre kell hozni egy `.wslconfig` nevű fájlt, melyben megadható a maximális processzor és memória mennyiség. Erről részletesen a [Microsoft](#) oldalán lehet olvasni. Az alábbi mintafájl 20 gigabájt memória és 10 virtuális processzor felhasználását teszi lehetővé a Docker számára.

```
[wsl2]
memory=20GB
processors=10
```

Módosítás után a Docker újraindítása szükséges.

Az alapbeállítás a konfigurációs fájl nélkül 50% vagy 8Gb memória (amelyik kisebb) illetve az összes virtuális processzor.

Érdemes a Tudástér használatához szükséges memóriánál magasabb értéket beállítani, mivel a Docker a Kuberneteset is konténerizáltan futtatja, így ennek erőforrásigénye is beleszámít a fent meghatározott limitbe.

Ezután a `kubectl` parancssori eszközzel ellenőrizhető a `k8s` állapota. A Docker Desktop alkalmazás automatikusan telepíti a parancssori eszközt. Amennyiben ez nem történne meg, lehetőség van [különálló](#) telepítésre is.

A `kubectl version` parancs visszaadja a kliens és szerver verzióját. A biztos működéshez ezek verziójának meg kell egyeznie. Sikeres parancsfuttatás esetén a `k8s` klaszter használatra kész.

A Tudástér használatához telepíteni kell az Ingress controllert, mely lehetővé teszi a Kubernetesen futó alkalmazások elérését a klaszteren kívülről, valamint a Metrics Servert, mely az erőforrás felhasználást monitorozza. Ezek telepítéséhez a programcsomag tartalmaz egy scriptfájlt OS függően `k8s_env.*` néven, mely feltelepíti a csomagokat. Linux és Mac esetén a fájlnak futási jogot kell beállítani, mely a `chmod +x k8s_env.sh` paranccsal tehető meg. Ezután a scriptet megfuttatva a fent említett csomagok telepítésre kerülnek.

Ezek telepítése manuálisan is elvégezhető. Ehhez a `/deployment/external` könyvtárban kell az alábbi két parancsot kiadni:

- `kubectl apply -f components.yaml`
- `kubectl apply -f deploy.yaml`

A prod környezetek az adatok mentésére csatolt lemezterületet használnak. Ezek elérési útvonala az adott környezet `volume.yaml` illetve `volume-patch.yaml` fájljában vannak definiálva. A könyvtárakat létre kell hozni és az `k8s`-nek írás-olvasás joggal kell rendelkeznie a használatához. Enélkül az alkalmazás ezen verziója nem indul el.

Windows esetében a WSL-t használó Docker Unix rendszer szerű elérési útvonalat lát. Így a megszokott Windows formátum helyett az alábbi minta szerint kell megadni az elérési útvonalat.

`"/run/desktop/mnt/host/c/data/nowhere_store/subject"`

Ez a C meghajtó `/data` könyvtárára mutat. Az alkönyvtárakat a Tudástér automatikusan létrehozza.

Az alapbeállítások megtartása esetén a következő könyvtárat kell létrehozni:

- Windows: C meghajtó gyöker könyvtárában a `data` nevű könyvtárat.
- Mac és Linux esetében a gyökérben a `data` nevű könyvtárat.

Mac esetén a root útvonal csak olvasható, így érdemes a könyvtárat a felhasználói könyvtár alatt létrehozni és a volume.yaml-ben az útvonalakat erre módosítani. A volume-ok a default névtérhez vannak kötve, így amennyiben az alkalmazás más névtérbe kerül telepítésre, ezt módosítani kell.

3.8 Hasznos Kubernetes parancsok

Parancs	Hatás
<code>kubectl apply -k .\prod-win-wsl\</code>	Windows prod környezet telepítése. A parancsot a deployment\overlays könyvtárban kell kiadni
<code>kubectl delete -k . \ prod\</code>	Linux prod környezet törlése
<code>kubectl get pods</code>	Futó POD-ok lekérdezése
<code>kubectl top pod</code>	POD-ok aktuális erőforrásainak megtekintése
<code>kubectl get services</code>	ClusterIP objektumok listázása
<code>kubectl get ingress</code>	Ingress controllerek listázása
<code>kubectl get deployments</code>	Deploymentek listája
<code>kubectl get configmap</code>	ConfigMap objektumok listája
<code>kubectl describe pod <pod neve></code>	A POD részletes adatai
<code>kubectl logs <pod neve> --tail=10</code>	A POD logjának utolsó 10 sora

5. táblázat Kubernetes parancsok

További parancsok a [Kubectl cheatsheet](#) oldalon találhatók. Bár parancssorból minden a k8s klasztert érintő művelet végrehajtható, sokszor gyorsabb a grafikus felületek használata. Ezek közül a [Lens](#) és a [K9s](#) a legelterjedtebb.

3.9 Egyéb telepítési módszerek

A k8s-en kívül az alkalmazás elindítható Docker konténerként, illetve a JAR fájlokat lokálisan futtatva is.

A konténerizált indításhoz a deployment könyvtár local-docker alkönyvtárában kell kiadni a `docker compose up` parancsot. Ez letölti az összes ms-t és a környezethez szükséges egyéb szoftvereket, majd elindítja azokat. A szükséges portokat pedig a lokális

portokra köti. Ahhoz, hogy ez megtörténjen, a 10000 – 10005 portoknak szabadnak kell lenni. A környezet törléséhez a `docker compose down` parancsot kell használni.

Az alkalmazás a <https://localhost:10000> porton elérhető.

A lokális futtatáshoz az alkalmazást a forráskódból kell lefordítani majd ezeket elindítani, illetve a `/deployment/local-env` könyvtárban kiadni a `docker compose up` parancsot a szükséges környezet elindításához. Az indítás előfeltétele a telepített Java 17+, npm 8.19+ és [Vue CLI](#), valamint a Dockerben futó környezet.

A fordítás és indítás menete a JVM alapú komponenseknél:

- belépés az alkalmazás könyvtárába
- fordítás a `gradlew build -x test` paranccsal (Windows esetén `gradlew.bat`)
- belépés a `build/libs` alkönyvtárba
- indítás a `java -jar -Dspring.profiles.active=local <jar_fájl_neve>` paranccsal

Ezt mind a nyolc szolgáltatással meg kell tenni.

A frontend fordítása és futtatása:

- belépés a `knowhere-vui` könyvtárba
- függőségek telepítése az `npm install` paranccsal
- indítás az `npm run dev` paranccsal.

Az alkalmazás ebben az esetben minden szolgáltatást a lokális portokon futtat, így ezeknek szabadnak kell lenni. Az alábbi táblázat összefoglalja, mely szolgáltatás melyik portot használja.

Alkalmazás	Port
MongoDB	27017
RabbitMQ	5672, 15672
Zipkin	9411
Spring Admin	9155
PostgreSQL	5432
Frontend (knowhere-vui)	8080
Api Gateway	8000, 9000
Grade Service	8006, 9006
Video Service	8001, 9001
Auth Service	8003, 9003
Subject Service	8004, 9004
Quiz Service	8005, 9005
Notification Service	8008, 9008
Enrollment Service	8007, 9007

6. táblázat Portok

A lokális indítás megkönnyítése érdekében készítettem egy script fájlt, ami az összes lépésen végigmegy. Ez saját használatra készült, így nem tesztelt. A Gradle parancs Windows verzióját használja, de tetszés szerint átírható Linuxra. A script a főkönyvtárban található `install_local.py` néven. Futtatásához Python 3 telepítése szükséges.

3.10 Jelszavak

A következő táblázatban a rendszerben előforduló felhasználónév/jelszó párokat gyűjtöttem össze.

Admin felhasználók	admin
Tanár felhasználók	teacher
Diák felhasználók	student
MongoDB	admin/admin123
PostgreSQL	admin/admin123
RabbitMQ	admin/admin123

7. táblázat Jelszavak

3.11 Felhasznált eszközök

Az alábbi táblázat összefoglalja a szakdolgozathoz felhasznált keretrendszereket, főbb könyvtárakat és egyéb szoftvereket, eszközöket.

IntelliJ Idea
GitHub Actions
Docker Hub
Docker Desktop
Kubernetes
Spring Boot
Gradle
Jib
SemVer
TestContainers
JUnit
Mockito
Micrometer
Kotlin
PostgreSQL
MongoDB

RabbitMQ
Spring Admin
Zipkin
Kustomize
ShedLock
Awaitility
JavaScript
VueJS
Vuetify
Axios
Tailwind CSS
K6
ffmpeg
Liquibase
Mongock
npm

8. táblázat Felhasznált eszközök

3.12 Továbbfejlesztési lehetőségek

Az alkalmazásba bármely, az iskolákban szükséges funkcionalitás beépíthető. Az alábbiakban összegyűjtöttem pár funkciót, mely hasznos lehet.

Élő óra, ahol a tanárok nem videókat töltenek fel, hanem webkamerán keresztül tartják meg az órát, melyhez az offline verzióhoz hasonlóan kérdéssor rendelhető.

Analitikák bevezetése. Itt megjeleníthetők lennének a különböző statisztikák. Például mikor nézték a legtöbbet a videókat, átlagosan hány diák és milyen eredménnyel végezte el az egyes kurzusokat és bármi, amit az adatok alapján érdemes megjeleníteni.

Publikus kurzusok, melyek regisztráció nélkül is megtekinthetők.

Fizetős kurzusok. A kurzus felvétele csak fizetés után lenne lehetséges.

Mikrotanúsítványok, melyeket bizonyos kurzusok vagy kurzus csoportok elvégzése után kaphat meg a tanuló.

Üzenetküldő rendszer, melyen keresztül az intézmény felhasználói kommunikálhatnak.

Tanulmányi időszakok kezelése. Félév, szemeszter iskolatípustól függően.

Órarend és naptár szerkesztő, mely segítené a diákokat a szervezésben.

Digitális asszisztens, akitől kérdezhetnek a tanárok és diákok.

Kurzusfájlok feltöltése. Szöveges leírások, feladatlapok kezeléséhez.

Irodalomjegyzék

- [1] Sam Newman: Building Microservices Designing Fine-Grained Systems (2. kiad.), O'Reilly Media, 2021, [586], ISBN 9781492034025.
- [2] Craig Walls: Spring in Action Sixth Edition, Manning, 2022, [520], ISBN 9781617297571
- [3] Marko Luksa: Kubernetes in Action, Manning, 2017, [624], ISBN 9781617293726
- [4] Dmitry Jemerov és Svetlana Isakova: Kotlin in Action, Manning, 2017, [360], ISBN 9781617293290
- [5] Erik Hanchett: Vue.js in Action, Manning, 2018, [304], ISBN 9781617294624
- [6] Benjamin Muschko: Gradle in Action, Manning, 2014, [480], ISBN 9781617291302
- [7] Cindy Sridharan: Distributed Systems Observability, O'Reilly Media, 2018, [36], ISBN 9781492033424
- [8] Ingress. [Link](#) (2023.04.05)
- [9] OWASP. [Link](#) (2023.03.28)
- [10] Problem Details RFC. [Link](#) (2023.03.26)
- [11] Resilience4J. [Link](#) (2023.03.27)
- [12] Spring Data. [Link](#) (2023.03.26)
- [13] Szemantikus Verziózás. [Link](#) (2023.03.29)
- [14] N+1 lekérdezés probléma. [Link](#) (2023.05.06)