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

Gorondi János

RAKTÁRKÉSZLET KEZELŐ ALKALMAZÁS BACKENDJÉNEK FEJLESZTÉSE SPRING BOOT PLATFORMON

KONZULENS

Imre Gábor

BUDAPEST, 2024

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott Gorrondi János, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2024. 12. 05

...…….
Gorondi János

Összefoglaló

A digitalizáció gyors fejlődése alapvetően formálta át az üzleti életet, különösen a logisztikát és a készletgazdálkodást. Napjainkban egy modern vállalat számára elengedhetetlen, hogy készleteit hatékonyan kezelje, valós időben nyomon kövesse, és minimalizálja az emberi hibákból fakadó veszteségeket. Ez a digitális kereskedelemben, az ipari gyártásban és az egészségügyben különösen fontos, hiszen a pontatlan vagy késlekedő raktárkezelés komoly gazdasági és működési kockázatokat jelenthet. Az automatizált rendszerek ebben nyújtanak segítséget: átláthatóbbá, kezelhetőbbé és hatékonyabbá teszik a folyamatokat, miközben csökkentik a költségeket.

Ezekre a kihívásokra reagálva dolgoztam ki egy modern, több szerepkört támogató raktárkezelő rendszer backendjét. A rendszer képes valós időben nyomon követni a készleteket, automatizált figyelmeztetéseket és utánrendeléseket generálni, valamint különféle szerepköröket (adminisztrátor, felhasználó, beszállító) kiszolgálni. A tervezés során kiemelt figyelmet fordítottam a skálázhatóságra, a biztonságra és a felhasználóbarát kialakításra.

A fejlesztés során a Spring keretrendszer rugalmasságát és megbízhatóságát használtam ki. A Spring Boot gyors indítást és hatékony REST API implementációt tett lehetővé, míg a Spring Security a JWT token alapú hitelesítéssel gondoskodott az alkalmazás biztonságáról. Az adatbázis-műveleteket a JPA/Hibernate egyszerűsítette, lehetővé téve azok zökkenőmentes integrálását az üzleti logikával. A Docker alapú konténerizáció biztosította a rendszer hordozhatóságát, míg a GitHub Actions segítségével automatizált CI/CD folyamatok garantálták a kód minőségét és a fejlesztési folyamat hatékonyságát.

A projekt eredményeként egy stabil és megbízható rendszer született, amely valós üzleti igényekre ad választ. Az automatizált figyelmeztetésektől a részletes riportok generálásáig a rendszer számos olyan funkcióval rendelkezik, amelyek egy modern vállalat működéséhez nélkülözhetetlenek.

Abstract

The rapid advancement of digitalization has fundamentally transformed the business world, particularly logistics and inventory management. Today, it is essential for modern companies to manage their inventory efficiently, monitor it in real time, and minimize losses caused by human errors. This is especially critical in digital commerce, industrial manufacturing, and healthcare, where inaccurate or delayed warehouse management can result in significant economic and operational risks. Automated systems help address these challenges by making processes more transparent, manageable, and efficient while reducing costs.

In response to these challenges, I developed the backend of a modern warehouse management system supporting multiple roles. The system tracks inventory in real time, generates automated alerts and reorder requests, and caters to various roles such as administrators, users, and suppliers. During the design phase, I prioritized scalability, security, and user-friendliness.

The development process leveraged the flexibility and reliability of the Spring framework. Spring Boot enabled rapid initialization and efficient REST API implementation, while Spring Security ensured application security with JWT-based authentication. Database operations were streamlined using JPA/Hibernate, enabling seamless integration with business logic. Docker-based containerization guaranteed the system's portability, and GitHub Actions facilitated automated CI/CD processes to maintain code quality and streamline development workflows.

The result of the project is a stable and reliable system that addresses real business needs. From automated alerts to generating detailed reports, the system offers numerous features that are indispensable for the operation of a modern enterprise.

1 Bevezetés

Az informatika és a digitalizáció forradalma az elmúlt évtizedekben gyökeresen átalakította mindennapjainkat és a gazdasági folyamatokat. A számítógépek, hálózati technológiák és automatizált rendszerek szinte mindenhol jelen vannak, az otthonoktól az iparig, a kereskedelemtől a szolgáltatásokig. A webes technológiák fejlődése lehetővé tette, hogy hatalmas mennyiségű adat váljon gyorsan elérhetővé, miközben a felhőalapú megoldások új szintre emelték a skálázhatóságot, rugalmasságot és költséghatékonyságot. Egy vállalat számára ma már az információk hatékony kezelése és gyors feldolgozása nemcsak előny, hanem alapfeltétel a versenyképességhez. Az ellátási lánc és a logisztika különösen sokat profitál ezekből az innovációkból, hiszen az automatizáció csökkenti az emberi hibákat, mérsékli a költségeket és növeli az ügyfélelégedettséget.

A raktárkezelő rendszerek mára alapvető eszközeivé váltak a kereskedelmi cégeknek, gyártóknak és online áruházaknak. A korábbi papíralapú nyilvántartások és manuális folyamatok helyett a modern rendszerek gyorsabbá, pontosabbá és átláthatóbbá teszik a raktározási munkát. Például egy online áruház esetében a készlet valós idejű frissítése biztosítja, hogy a vásárlók mindig pontos információt kapjanak az elérhető termékekről, és az alacsony készletszintek figyelmeztetése segíti az utánrendelések időben történő elindítását.

Az automatizált raktárkezelő rendszerek nemcsak a kereskedelemben, hanem az egészségügyben és az iparban is kiemelt szerepet játszanak. A gyógyszerek és orvosi eszközök nyomon követése, a gyártási alapanyagok folyamatos elérhetősége vagy a logisztikai folyamatok hatékonyságának növelése mind olyan területek, ahol az ilyen rendszerek minimalizálhatják a hibalehetőségeket és csökkenthetik a költségeket. A prediktív funkciók lehetővé teszik az igények és készlethiányok előrejelzését, így a vállalatok időben reagálhatnak a kihívásokra, ami stratégiai előnyt jelent.

A technológiai fejlődés mára elérhetővé tette ezeket a rendszereket nemcsak nagyvállalatok, hanem kisebb cégek számára is. Az automatizált raktárkezelő rendszerek csökkentik az adminisztrációs terheket, kiküszöbölik az emberi hibákat és javítják a működés megbízhatóságát. Ezek az eszközök nemcsak a hatékonyságot növelik, hanem a versenyképesség javításához is hozzájárulnak.

Ezek az előnyök inspiráltak arra, hogy a szakdolgozatom témájául egy modern raktárkezelő rendszer tervezését és fejlesztését válasszam. Nemcsak egy működő szoftver létrehozása volt a célom, hanem egy olyan eszköz megalkotása, amely valós üzleti problémákra kínál hatékony és fenntartható megoldást. A fejlesztés során kiemelt figyelmet fordítottam arra, hogy a rendszer rugalmas, könnyen kezelhető legyen, és egyszerre több szerepkör – például adminisztrátorok, felhasználók és beszállítók – igényeit is kielégítse. Az alkalmazás valós idejű készletfrissítéseket végez, figyelmeztetéseket küld alacsony készletszint vagy lejáró termékek esetén, és támogatja az automatizált utánrendeléseket.

A fejlesztés során a Java programozási nyelv adta meg a rendszer alapját. A Java népszerűsége és rugalmassága lehetővé tette, hogy a projekt során biztonságos, skálázható és hosszú távon is karbantartható kódot készítsek. A választás egyik fő szempontja az volt, hogy a Java platformfüggetlenségével biztosítani tudjam a rendszer hordozhatóságát és zökkenőmentes telepíthetőségét különböző környezetekben. A Spring keretrendszer szorosan összefonódott a Java-val, és kulcsszerepet játszott a projekt sikerében. A Spring legnagyobb előnye a moduláris felépítésében rejlik, amely lehetővé teszi, hogy pontosan azokat a komponenseket használjam, amelyekre az adott projektben szükség van. A Spring Boot modult választottam a projekt indításához, amely rendkívüli módon leegyszerűsítette az alkalmazás alapjainak felépítését.

1.1 Dolgozat felépítése

A következő fejezetben ismertetem az alkalmazás létrehozásához használt technológiákat és fejlesztési eszközöket. Ezután bemutatom az architektúrát és a logikai adatmodellt, majd a későbbi fejezetekben részletesen tárgyalom a szerveroldali komponensek felépítését, működését, tesztelését, valamint a konténerizálás folyamatát.

2 Követelmények

2.1 Megvalósított funkciók

A raktárkezelő rendszer fejlesztése során számos funkciót valósítottam meg, amelyek a különböző szerepkörök igényeit szolgálják ki, miközben biztosítják a rendszer átláthatóságát, hatékonyságát és megbízhatóságát:

Felhasználók kezelése

- Adminisztrátor felhasználó: Az adminisztrátor (admin) teljes jogosultsággal rendelkezik, új felhasználókat hozhat létre, beszállítókat regisztrálhat, és véglegesítheti a megrendeléseket.
- Általános felhasználó: Alapvető jogosultságokkal rendelkezik, például megrendelések leadása és saját rendeléseinek kezelése.
- Beszállító felhasználó: A beszállítók saját készleteiket kezelhetik, termékeket tölthetnek fel és vihetnek el a raktárból.

Termékek kezelése

- Kategóriák kezelése: Az adminisztrátorok kategóriákat hozhatnak létre, amelyek segítik a termékek rendszerezését.
- Terméksablonok kezelése: Az adminisztrátorok termék sablonokat hozhatnak létre, melyek alapján az egyes beszállítók termékeit nyomon lehet követni.
- Termékek kezelése: A rendszer a beszállítóktól érkező tényleges termékeket rögzíti, kezeli azok állapotát és leltárba vételét.

Beszállítók kezelése

- Beszállítók regisztrálása: Az adminisztrátor új beszállítókat regisztrálhat, akikhez automatikusan hozzárendelődik egy dedikált raktáregység, amelyben kizárólag saját készleteiket kezelhetik.
- Raktári műveletek: A beszállítók termékeket tölthetnek fel vagy vihetnek el saját raktáraikból.

Megrendelések kezelése

- Rendelések leadása: Bármely jogosultságú felhasználó leadhat rendelést, feltéve, hogy van elegendő készlet, és a rendelés megfelel a feltételeknek. Az ilyen rendelések függőben lévő státuszba, miközben a bennük szereplő termékek lefoglalásra kerülnek.
- Rendelések véglegesítése: Az adminisztrátor a rendelések véglegesítésével a megrendelést teljesített állapotba helyezi, miközben a rendelt termékek kikerülnek a raktárból.

Valós idejű készletkezelés

- Készletváltozások követése: Azonnali naplózás minden készletváltozásról.
- Lefoglalt készletek: A megrendelt termékek státusza lefoglalt, így más rendelés nem érheti el azokat.
- Automatikus figyelmeztetések: Alacsony készletszám esetén figyelmeztetést generál a rendszer, szükség esetén automatikus utánrendelést indít.
- Rendszeres szavatosság-ellenőrzés: A rendszer meghatározott időközönként ellenőrzi a teljes készletet, és ha közelgő lejáratú termékeket talál, automatikus figyelmeztetéseket generál a megfelelő felhasználók számára.

Riportok készítése

- Megrendelések riportja: Átlátható statisztikát nyújt a leadott és teljesített rendelésekről.
- Készletállapot riport: Információt ad a raktár aktuális kapacitásáról és kihasználtságáról.
- Lejárati idő riport: Figyeli a termékek szavatosságát és átfogó jelentést ad lejáratukról.

2.2 Nem funkcionális követelmények

Biztonság

A rendszer biztonsági követelményeit a Spring Security és JWT (JSON Web Token) token-alapú hitelesítés biztosítja:

- A felhasználói adatok bizalmas kezelése.
- A jelszavak titkosítása.

- Jogosulatlan hozzáférés elleni védelem.

Hordozhatóság

A rendszer Docker konténerizációval lett kialakítva, amely biztosítja a platformfüggetlen futtatást. A Docker Compose használatával az alkalmazás komponensei (adatbázis, szerver oldali applikáció) egyszerűen elindíthatók és konfigurálhatók.

Tesztelés

A fejlesztés során a magas szintű tesztlefedettséget prioritásként kezeltem:

- Az automatikus tesztelés során a különböző rétegek (üzleti logika, adatforrás) működését izoláltan és integráltan is ellenőriztem.
- A folyamatos integráció és szállítás (CI/CD) folyamata minden egyes kódváltoztatás után automatikusan lefuttatja a teszteket, ezzel biztosítva a program minőségét és megbízhatóságát. Ez a megközelítés segít az esetleges hibák gyors észlelésében és kijavításában.

3 Felhasznált technológiák

3.1 Spring

A Spring keretrendszer egyik kiemelkedő előnye, hogy erőteljes támogatást nyújt a különböző adatkezelési és tranzakciókezelési mechanizmusokhoz, amelyek lehetővé teszik a fejlesztők számára a komplex üzleti logikát és adatkezelést is egyszerűen implementálni. A Spring Data és a Spring Transaction Management segítségével a fejlesztők könnyedén dolgozhatnak adatbázisokkal, kezelhetik az adatok integritását, és biztosíthatják, hogy a tranzakciók megbízhatóan és hatékonyan hajtódjanak végre. Ez különösen fontos nagyvállalati alkalmazások esetén, ahol a teljesítmény, megbízhatóság és az adatkonzisztencia kritikus tényezők.

Az én projektemben a Spring keretrendszer használata alapvető fontosságú volt, hiszen a modern raktárkezelő rendszerekhez elengedhetetlen egy erős és megbízható backend architektúra. A Spring alapmoduljai – például az objektumok életciklusának kezelése és a függőség-injektálás (Dependency Injection, DI) – biztosítják az alkalmazás skálázhatóságát, könnyű karbantarthatóságát és hatékony fejlesztését.

3.1.1 Spring IoC és Bean-ek kezelése

A Spring keretrendszer egyik alapvető eleme a Bean, amely az alkalmazás működéséhez szükséges objektumokat jelöli. Ezeket a Spring Inversion of Control (IoC) konténer kezeli, amely felelős a Bean-ek példányosításáért, életciklusuk irányításáért és konfigurációjukért. [1]

Az IoC konténer lényege, hogy hatékonyan kezelje az osztályok közötti függőségeket. Az objektumok nem önállóan hozzák létre a szükséges függőségeket, hanem a konténer végzi el ezt a feladatot. Ez a megközelítés csökkenti az osztályok közötti szoros összekapcsolódást, így az alkalmazás rugalmasabb, könnyebben tesztelhető és karbantartható lesz.

Egy példa erre, amikor egy Company osztály egy Address típusú objektumot igényel. Az IoC konténer biztosítja a szükséges Address példányt, anélkül, hogy azt a Company osztály maga hozná létre. Ehhez elegendő a Company és Address osztályokat Bean-ként regisztrálni, például az @Component vagy @Configuration annotációk használatával.

Ez a megközelítés nemcsak egyszerűsíti a függőségek kezelését, hanem lehetővé teszi az objektumok újrafelhasználását az alkalmazás különböző részein

3.1.2 Spring Boot

A Spring Boot a Spring keretrendszer egy kiterjesztése, amelynek célja, hogy a fejlesztést jelentősen leegyszerűsítse. Az egyik legnagyobb előnye az automatikus konfiguráció, amely megszünteti az alapvető beállítások manuális elvégzésének szükségességét.

A Spring Boot használatával az alkalmazások indítása rendkívül gyors és egyszerű. Az úgynevezett Spring Initializer eszköz segítségével néhány kattintással létrehozható egy előre konfigurált projektváz, amely azonnal használatra kész.

A Spring Boot kulcselemei:

- application.yaml: Az alkalmazás konfigurációit átlátható és könnyen módosítható formában tárolhatjuk YAML fájlban. Itt határozhatók meg például az adatbáziskapcsolatok, a környezeti beállítások és egyéb paraméterek.
- Beépített szerverek: A Spring Boot saját szervereket (például Tomcat) tartalmaz, így nincs szükség külön környezet telepítésére.
- Egyszerű függőségkezelés: A Maven build-eszközökkel könnyedén hozzáadhatók a szükséges modulok.

A saját projektemben a Spring Boot nagyban hozzájárult a hatékony fejlesztéshez, mivel minimalizálta a konfigurációs feladatokat, miközben biztosította a modern raktárkészlet kezelő alkalmazásokhoz szükséges skálázhatóságot és megbízhatóságot.

3.1.3 REST API és kommunikáció kezelése

A Spring keretrendszer egyik legnagyobb erőssége, hogy egyszerűvé és hatékonnyá teszi a REST API-k (Representational State Transfer) fejlesztését. A REST egy olyan architekturális stílus, amely a HTTP (HyperText Transfer Protocol) protokollra épül, és lehetőséget biztosít az erőforrások interneten keresztüli elérésére és kezelésére. Az egyes REST API (Application Programming Interface) végpontokat URL-ek (Uniform Resource Locator) azonosítják, melyeken keresztül a kliensoldali alkalmazások vagy más külső rendszerek hozzáférhetnek a szerver által kínált funkciókhoz és adatokhoz.

A REST API-k a HTTP protokoll kulcsfontosságú metódusait használják a különböző műveletek végrehajtására:

- GET: Erőforrások lekérdezésére vagy olvasására szolgál.
- POST: Új erőforrások létrehozására.
- PUT: Meglévő erőforrások frissítésére vagy felülírására.
- DELETE: Erőforrások törlésére.

Ezek a metódusok lehetővé teszik, hogy az alkalmazás által kezelt adatokkal, például termékekkel különböző műveleteket hajtsunk végre, mint például új elemek hozzáadása, módosítása vagy törlése.

A Spring keretrendszer REST Controller osztályokat használ a REST API-k végpontjainak definiálására és az azokhoz kapcsolódó logika megvalósítására. Ezek az osztályok annotációk segítségével konfigurálhatók, amelyek leegyszerűsítik és felgyorsítják az implementációt. Az olyan annotációk, mint a @RestController, a végpontokat kezelő osztályokat definiálják, míg a @RequestMapping az alapértelmezett útvonalakat határozza meg. A HTTP metódusokhoz, például GET vagy POST kérésekhez, a @GetMapping és hasonló annotációk használhatók, amelyek lehetővé teszik a kérések egyszerű kezelését. [2]

Ezek az annotációk nagyban hozzájárulnak ahhoz, hogy a REST API-k strukturáltak, könnyen érthetőek és karbantarthatóak legyenek. Az API-kon keresztül a backendem tiszta és hatékony módon szolgáltatta az adatokat, miközben az üzleti logika és az adatkezelés a szerveroldalon maradt.

3.2 ORM és JPA/Hibernate

A modern alkalmazások, különösen azok, amelyek nagy mennyiségű adatot kezelnek – mint például a raktárkészlet-kezelő rendszerek –, elképzelhetetlenek hatékony adatbázis-kezelés nélkül. Az objektum-relációs leképezés (Object-Relational Mapping, ORM) technológia jelentős előnyt nyújt az adatbázisok és az alkalmazások közötti integrációban, mivel lehetővé teszi, hogy az adatbázis-táblák Java objektumokként jelenjenek meg. Ezáltal az adatkezelés átláthatóbbá válik, miközben csökkenti a manuális SQL-lekérdezések szükségességét. Az ORM használata gyorsabb fejlesztést, jobb karbantarthatóságot és nagyobb biztonságot eredményez, hiszen az adatbázis-műveletek zökkenőmentesen illeszkednek az alkalmazás logikájába.

A projekt során a Spring Data JPA (Java Persistence API) modult alkalmaztam, amely a Hibernate-et, az ORM egyik legismertebb implementációját használja. [3] Ez a modul előre megírt és testre szabható metódusokat kínál a leggyakoribb adatbázis-műveletekhez, mint például mentés, törlés vagy lekérdezés. Ezen kívül lehetőséget biztosít a komplex lekérdezések egyszerű megfogalmazására, ami jelentősen növeli a fejlesztés hatékonyságát.

3.3 PostgreSQL

A PostgreSQL egy nyílt forráskódú, relációs adatbázis-kezelő rendszer, amely az egyik legnépszerűbb választás a modern alkalmazások körében. Különösen a stabilitása és az ACID-tulajdonságok (Atomicity, Consistency, Isolation, Durability) biztosítása miatt választottam, amelyek garantálják az adatok integritását és megbízhatóságát. Ez különösen fontos egy raktárkezelő rendszer esetében, ahol az adatvesztés vagy az inkonzisztens állapotok komoly problémákat okozhatnak.

A Spring keretrendszer alapértelmezett támogatást biztosít a PostgreSQL-hez, így az adatbázis könnyen illeszkedett a projekt technológiai eszközei közé.

3.4 Biztonság

Mivel a rendszernek személyes adatokat és érzékeny információkat kell kezelnie, elengedhetetlen, hogy megfelelő védelmet biztosítsunk a jogosulatlan hozzáférés és az adatlopás ellen.

3.4.1 Spring Security

A Spring Security egy rendkívül erőteljes és rugalmas biztonsági keretrendszer, amely a felhasználói jogosultságok kezelésére és az alkalmazás védelmére szolgál. A felhasználói autentikáció és autorizáció fontos szerepet játszik a raktárkezelő rendszeremben, mivel különböző szerepkörökkel rendelkező felhasználók dolgoznak a rendszerben (például adminisztrátorok, felhasználók és beszállítók). A Spring Security segítségével biztosíthatom, hogy az alkalmazás minden végpontját megfelelően védjem, és csak az arra jogosult felhasználók férhessenek hozzá az érzékeny adatokhoz.

Gyakorlati példaként, ha valaki adminisztrátori jogosultságot kap, akkor számára minden végpont elérhető lesz, míg egy felhasználó csak a saját profilját és a nyilvános adatokat érheti el. Az autentikációs beállításokat úgy végezhetjük el, hogy a regisztrációs

vagy bejelentkezési végpontok szabadon hozzáférhetőek, míg minden más kéréshez előzetes bejelentkezés szükséges.

3.4.2 Jwt token autentikáció

A másik fontos biztonsági mechanizmus, amit alkalmaztam, az JWT (JSON Web Token), amely az alkalmazás autentikációs folyamatát könnyíti meg. A JWT egy állapotmentes, biztonságos megoldás, amely lehetővé teszi, hogy a felhasználó a rendszerhez való hozzáférését egy egyszerű token segítségével hitelesíthesse. Mivel az alkalmazásom nem igényel állapotot a szerveroldalon (stateless), így a JWT ideális megoldás volt.

A JWT token egy JSON formátumban kódolt token, amely tartalmazza az autentikációhoz szükséges információkat, például a felhasználó azonosítóját és jogosultsági szintjét. [4] Az alkalmazás a felhasználó sikeres bejelentkezése után egy aláírt JWT tokent generál, amelyet a kliensoldali alkalmazás a jövőbeli kérésekhez csatolhat.

Amikor a felhasználó újabb kérést küld, a JWT a kérés fejlécében kerül továbbításra, és a szerver ellenőrzi annak érvényességét. Ha a token érvényes, a kérés teljesítése megtörténik. Így a JWT nemcsak az autentikációt, hanem az állapotmentes biztonságot is garantálja, mivel nincs szükség session állapot fenntartására a szerveroldalon.

3.5 Tesztelés

A tesztelés minden fejlesztési projekt szerves részét képezi, hiszen biztosítja a kód helyességét, megbízhatóságát és teljesítményét. A tesztelési részletekről bővebben egy későbbi fejezetben számolok be.

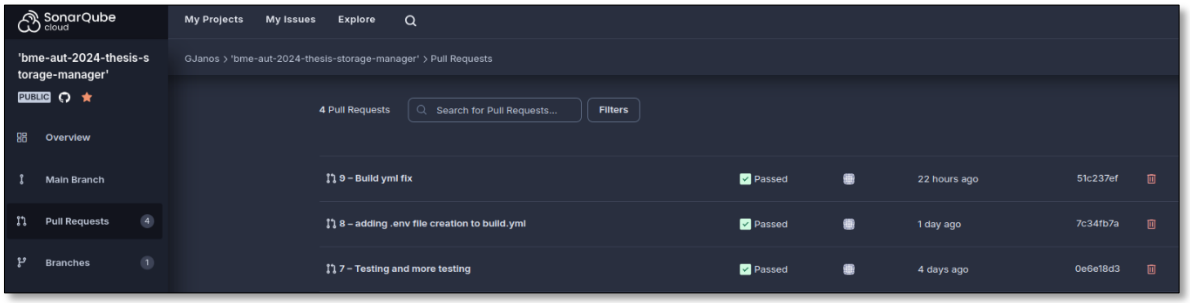
3.5.1 Spring Starter Test

A teszteléshez a Spring keretrendszerhez tartozó Starter Test modult használtam, amely előre konfigurált környezetet biztosít az alkalmazás teszteléséhez. A Spring Boot egyszerűsíti a tesztelést, mivel integrált megoldásokat kínál a különböző komponensek, mint például a konténerek, adatbázisok és webes végpontok teszteléséhez. Ezen modul segítségével könnyedén konfigurálhatóak a tesztek, anélkül, hogy manuálisan kellene beállítani a különböző függőségeket és környezeteket. Ez jelentősen felgyorsítja a tesztelési folyamatokat, és csökkenti a hibák lehetőségét. [5]

3.5.2 SonarQube, Jacoco

A kódminőség folyamatos biztosítása érdekében két fontos eszközt használtam: SonarQube és Jacoco. A SonarQube egy statikus kódelemző eszköz, amely automatikusan elemzi a kódot, és különböző kódminőségi problémákat, mint például a duplikált kódrészletek, potenciális hibák, nem használt változók és biztonsági rések, azonnal jelez.

A Jacoco egy kódlefedettségi eszköz, amely azt méri, hogy a tesztek a kód mely részeit fedik le. A Jacoco integrálása az alkalmazásba lehetővé tette, hogy mérjem, mely kódrészeket teszteltük és melyek azok, amelyek további tesztelést igényelnek. A kódlefedettségi adatok a SonarQube webes felületén is megjelennek, így könnyen áttekinthető a tesztelt kódrészek aránya. Az integráció biztosította, hogy az összes teszt lefedje a legfontosabb funkciókat, és semmilyen kritikus részegység ne maradjon tesztelés nélkül. [6] A SonarQube és Jacoco kombinálásával nemcsak a kód minőségét, hanem a tesztelés hatékonyságát is sikerült maximalizálni.



1. ábra: SonarQube webes felület.

3.5.3 Github Actions

A CI/CD rendszer kulcsfontosságú szerepet játszott a projekt automatizált tesztelésében. A GitHub Actions segítségével automatizáltam a tesztelési folyamatokat, így minden egyes kódváltoztatás után automatikusan lefutottak a tesztek, valamint a kód minőségi ellenőrzése (SonarQube) és a kódlefedettség vizsgálata (Jacoco). [7] Ez biztosította, hogy a fejlesztés minden fázisában azonnali visszajelzést kapjak a kód állapotáról, minimalizálva a hibák esélyét a későbbi szakaszokban.

3.5.4 OpenAPI

Bár a projektben nem volt frontend fejlesztés, az API végpontok dokumentálására és tesztelésére az OpenAPI specifikációkat és a Swagger annotációkat használtam. A

Swagger annotációk lehetővé tették, hogy a Spring Boot alkalmazásomban az egyes API kontrollereket és végpontokat megfelelően dokumentáljam, így az API-k részletes specifikációi automatikusan generálódtak. A Swagger annotációk segítségével a rendszerem gyorsan elérhetővé tette az interaktív API dokumentációt, amelyen keresztül egyszerűen tesztelhettem az API végpontokat.

A Swagger UI-t (felhasználói felület) a backend alkalmazásban való teszteléshez használtam, amely automatikusan létrehozta az interaktív felületet a Swagger annotációk alapján. Ez a felület lehetővé tette, hogy az API végpontok gyorsan kipróbálhatók legyenek, így a fejlesztés során könnyedén validálhattam, hogy a rendszer minden végpontja megfelelően működik, anélkül, hogy külön frontend alkalmazásra lett volna szükség.

2. ábra: Swagger annotációk által generált UI, példa POST kéréssel.

3.6 Docker és Docker-Compose

A konténerizáció egy olyan technológiai megközelítés, amely lehetővé teszi, hogy egy alkalmazás és annak függőségei egy konténerben futtathatók legyenek. A konténer lényegében egy izolált környezet, amely biztosítja, hogy az alkalmazás minden szükséges erőforrással és konfigurációval rendelkezzen, függetlenül attól, hogy milyen operációs rendszeren fut. Ennek a megközelítésnek az egyik legnagyobb előnye, hogy az alkalmazás hordozható: a konténer ugyanúgy fut mindenféle platformon (fejlesztői gépen, teszt környezetben, éles környezetben), így kiküszöbölhetők a környezeti problémák, amelyek máskülönben az alkalmazások működését befolyásolhatják. [8]

A Docker Compose egy olyan eszköz, amely lehetővé teszi több konténer egyidejű kezelését és konfigurálását egyetlen fájl segítségével. A Docker Compose lehetővé teszi, hogy egyszerre indítsunk el több konténert, és ezek közötti kapcsolatokat könnyen definiáljuk. Ez különösen hasznos olyan alkalmazások esetén, amelyek több

szolgáltatásra (pl. adatbázis, backend, frontend) építenek, és szükség van arra, hogy ezeket egyszerre futtassuk.

3.7 Git és GitHub

A Git és a GitHub a szoftverfejlesztés egyik legismertebb verziókezelő eszköze, amelyek lehetővé teszik a kód változásainak nyomon követését és a csapatmunkát. A Git egy elosztott verziókezelő rendszer, amely lehetővé teszi, hogy minden fejlesztő a saját helyi példányán dolgozzon, majd később könnyedén szinkronizálja a módosításokat a központi adattárral (repository). A GitHub pedig egy online platform, amely segíti a kód tárolását, megosztását és a közös munkát, miközben biztosítja a verziók és ágak kezelését.

Bár egyedül dolgoztam a projekten, a verziókezelési munkafolyamatokat szigorúan követtem, hogy biztosítsam a legjobb gyakorlatokat. A GitHub platformon a feature branch (fejlesztési ág) megközelítést alkalmaztam, így minden új fejlesztés külön ágon zajlott, miközben a fő ágat (master/main) folyamatosan stabilan tartottam. Minden változtatás után commitáltam (mentettem) és pusholtam (feltöltöttem) a kódot az adattárba, majd a kód integrálásához pull requestet (összevonási kérést) használtam. Ez lehetőséget biztosított arra, hogy átnézzem a módosításokat, mielőtt azok a fő ágba kerültek volna.

4 Tervezés

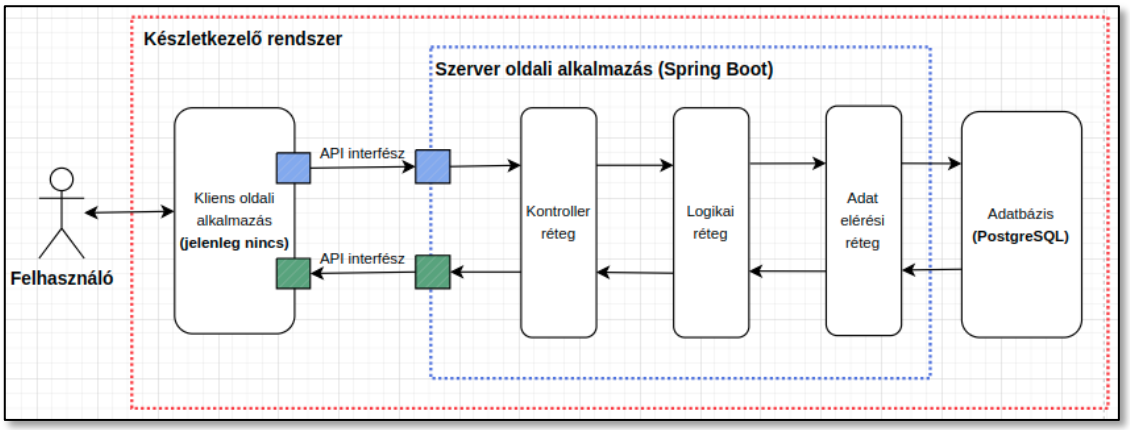
A tervezési folyamat során törekedtem egy egyszerű, ugyanakkor rugalmas architektúra kialakítására, amely megfelel a jelenlegi követelményeknek és alkalmazkodik a jövőbeni igényekhez is.

Az alkalmazás két fő rétegből áll:

- Üzleti logikai réteg: Ez a réteg valósítja meg az üzleti folyamatokat és szabályokat, biztosítva a funkcionalitás koherenciáját.
- Adatforrás réteg: Ez a réteg az adatok tárolásáért és kezeléséért felelős. Általában relációs adatbázison (RDBMS) alapul, de más adattárolási megoldások is alkalmazhatók, például NoSQL rendszerek.

4.1 Logikai architektúra

Az alábbiakban bemutatom a rétegek szerepét és működését, valamint ismertetem, hogyan járulnak hozzá az alkalmazás funkcionalitásához. A réteges architektúra felépítését az alábbi diagram szemlélteti:



3. ábra: Réteges architektúra.

4.1.1 Üzleti logikai réteg

Az üzleti logikai réteg az alkalmazás központi eleme, amely az üzleti szabályok és folyamatok megvalósításáért felel. Az általam fejlesztett rendszerben ezt a Spring Boot alapú szerveroldali alkalmazás biztosítja. Mivel jelenleg nincs frontend komponens, az API interfészek közvetítik az adatokat a backend szolgáltatások felé. Ez a megoldás rugalmas, hiszen a már kialakított API lehetővé teszi, hogy később bármilyen frontend

vagy külső rendszer csatlakozzon az alkalmazáshoz anélkül, hogy jelentős módosításokat kellene végezni a backend kódbázison.

A réteg három fő alkomponensből áll:

- Kontroller réteg: Az API-n keresztül fogadja a felhasználói kéréseket. A REST vezérlők az adatok továbbításáért és az eredmények visszaküldéséért felelnek.
- Logikai réteg: Az üzleti szabályok végrehajtásáért és a megfelelő szolgáltatások hívásáért felelős. Ez biztosítja az üzleti logika működésének integritását.
- Adatelérési réteg: Az adatbázis-műveleteket végzi, az adatokat a tárolóból nyeri ki, és továbbítja az üzleti logikai réteg felé. Ez a réteg JPA és Hibernate ORM technológiákon alapszik, amelyek hatékony adatkezelést biztosítanak.

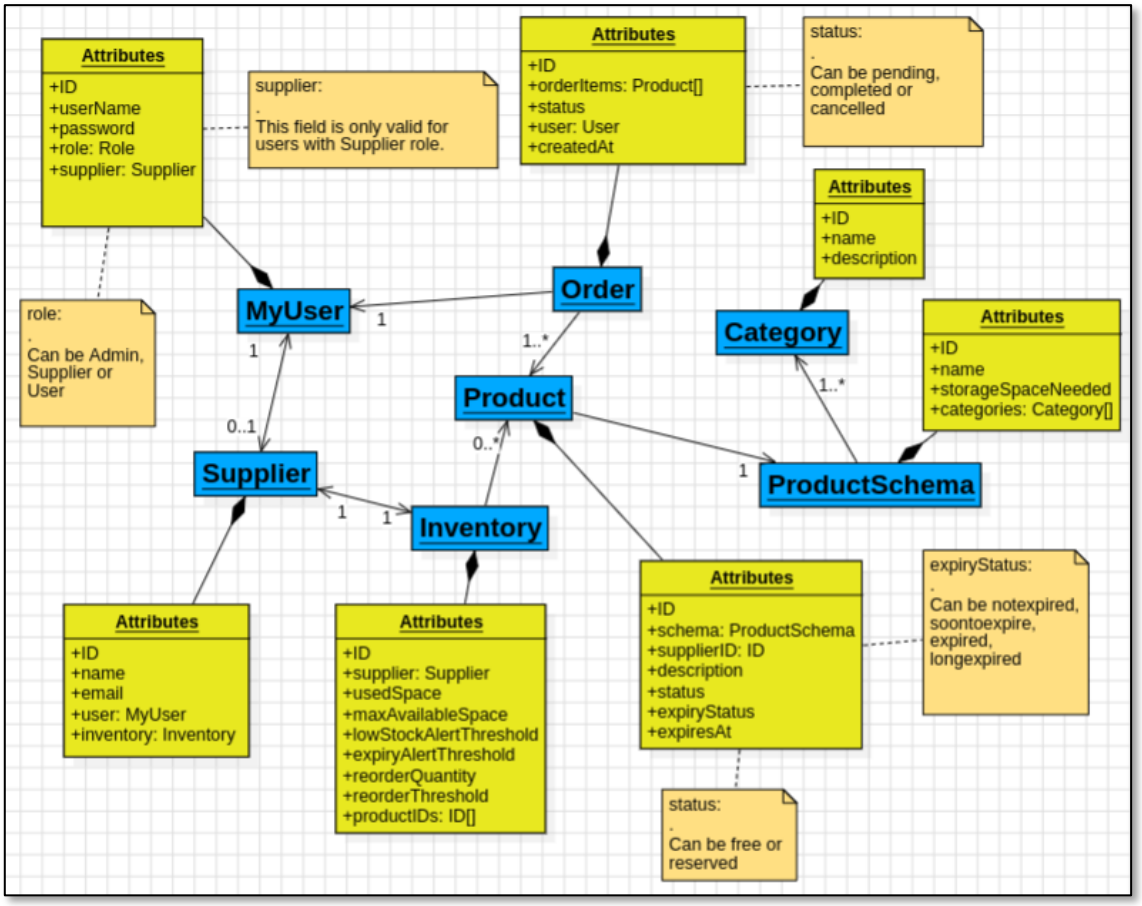
4.1.2 Adatforrás réteg

Az adatforrás réteg az alkalmazás által kezelt adatok tárolásáért felelős, és relációs adatbázison, konkrétan PostgreSQL-en alapul. Ez a réteg biztosítja az adatok tárolását, kezelését és hosszú távú megőrzését.

A két réteg szoros együttműködésének köszönhetően a rendszer hatékonyan kezeli a készletkezelési folyamatokat, miközben könnyen igazítható bármilyen jövőbeni technológiai követelményhez. Az ORM technológiák alkalmazása továbbá csökkenti a manuális adatkezelésből adódó hibalehetőségeket.

4.2 Logikai adatmodell

Az alkalmazás logikai adatmodellje az entitások és azok közötti kapcsolatok átfogó leírását tartalmazza. A rendszer célja egy több szerepkörös készletkezelő alkalmazás támogatása, így az adatmodell kifejezetten ennek a funkciónak a megvalósítását szolgálja. Az entitások közötti viszonyokat az alábbi entitásdiagram szemlélteti:



3. ábra: Alkalmazás entitás diagramja.

Entitások kék, attribútumaik sárga, információk narancsszínnel jelöltek.

4.2.1 Entitások

4.2.1.1 MyUser

A MyUser entitás az alkalmazás felhasználóit írja le, és tartalmazza az azonosításhoz szükséges alapvető adatokat, például a felhasználónevet és a jelszót. Az entitás egyedi azonosítóval rendelkezik, amely az adatbázisba történő mentéskor automatikusan generálódik. A MyUser entitás fontos eleme a szerepkör (role), amely meghatározza a felhasználó jogosultságait és hozzáférési lehetőségeit az alkalmazásban.

A rendszerben háromféle szerepkör létezik, amelyek közül a regisztráció során a felhasználók alapértelmezés szerint a User szerepkört kapják, amely korlátozott jogosultságokat biztosít számukra. Ezzel szemben a Supplier szerepkörű felhasználókhoz automatikusan kapcsolódik egy beszállítói (Supplier) entitás és egy készletkezelési (Inventory) modul, amelyek a beszállító adatait és a hozzá tartozó készleteket kezelik. Az Admin szerepkörrel rendelkező felhasználók a legmagasabb szintű hozzáféréssel bírnak, így teljes körű irányítást gyakorolhatnak az alkalmazás minden funkciója felett. Ez a

felépítés biztosítja, hogy az egyes felhasználók pontosan azokhoz a funkciókhoz férjenek hozzá, amelyek a szerepükből adódó feladataik elvégzéséhez szükségesek.

4.2.1.2 Supplier

A Supplier entitás a beszállítók speciális adatait kezeli. Minden Supplier entitás kapcsolódik egy MyUser entitáshoz, amely az adott beszállítóhoz tartozó felhasználói fiókot jelképezi. Ez az entitás tartalmazza a beszállítókhoz kapcsolódó olyan adatokat, mint a név, e-mail cím, valamint a hozzájuk rendelt készlet (Inventory).

4.2.1.3 Inventory

Az Inventory entitás a beszállítók által kezelt készleteket képviseli. Ez a teljes raktár egy olyan elkülönített része, amely felett kizárólag a hozzárendelt beszállító rendelkezik. A termékek betárolását és kivételét kizárólag az adott Inventory-hoz tartozó beszállító végezheti, még az adminisztrátornak sincs jogosultsága ezekre a műveletekre.

Ez az entitás tartalmazza a készletek kapacitásával és állapotával kapcsolatos adatokat, például a maximálisan elérhető tárolóhelyet (maxAvailableSpace), a már felhasznált kapacitást (usedSpace), valamint a készletek utánrendelési és lejárati figyelmeztetéseinek paramétereit. Minden Inventory entitás szorosan kapcsolódik egyértelműen meghatározott Supplier (beszállító) entitáshoz.

4.2.1.4 Category

A Category entitás a termékekhez rendelt kategóriákat definiálja. Minden kategória tartalmaz egy egyedi azonosítót és nevet. A kategóriák lehetővé teszik a termékek logikus csoportosítását, egyszerűsítve a keresést és kezelést.

4.2.1.5 ProductSchema

A ProductSchema (termék sablon) entitás az Adminisztrátor által létrehozott, előre meghatározott sablonokat tartalmazza, amelyek meghatározzák, hogy milyen termékek tárolása engedélyezett a raktárban. Minden egyes terméknek kötelező rendelkeznie egy hozzá kapcsolódó termék sablonnal; enélkül nem helyezhető el a raktárban. A beszállítóknak, amikor terméket szeretnének behozni, meg kell adniuk a kívánt mennyiséget és azt, hogy melyik termék sablonra vonatkozik a szállítás.

Ez az entitás tartalmazza a termékekhez tartozó kategóriákat és a tároláshoz szükséges helyigényt, amelyek már a sablon létrehozásakor rögzítésre kerülnek. Ezen információk előzetes ismerete nemcsak a raktár kezelhetőségét segíti, hanem

hatékonyabbá teszi a rendezést és keresést is, hiszen az azonos sablonú termékek együttesen kezelhetők.

Példa: Az Adminisztrátor létrehozott egy „Samsung_Galaxy_S24” nevű termék sablont, amely a „Mobil_Telefon” kategóriába tartozik. Ezután egy beszállító például 10 darab ilyen sablonú terméket szállíthat be a saját dedikált raktárhelyiségébe. A raktárba a ténylegesen tárolt elemek már Product entitásként jelennek meg, amely egy-egy sablonhoz tartozó konkrét terméket reprezentál.

4.2.1.6 Product

A Product entitás a rendszerben tárolt konkrét termékeket írja le, amelyek a készletkezelés alapvető elemei. Minden termékhez egyedi azonosító tartozik, valamint kapcsolódik hozzá egy ProductSchema, amely meghatározza a termék tulajdonságait és tárolási követelményeit. Az entitás továbbá tartalmazza a beszállító azonosítóját, amely rögzíti, hogy melyik beszállító helyezte el a terméket a raktárban, valamint a termék lejárati dátumát, amely a készlet ellenőrzésének és a raktárkezelés hatékonyságának kulcsfontosságú eleme.

A termék aktuális állapotát két fő mező határozza meg. Az első a *status*, amely a termék elérhetőségét mutatja, például szabad, foglalt vagy eltávolított állapotban. A másik a *expiryStatus*, amely a lejárati dátum alapján változik, és információt nyújt arról, hogy a termék érvényes, hamarosan lejár, már lejárt, vagy régóta lejárt állapotban van. A termékek a raktárkészlethez (Inventory) kapcsolódnak, ahol fizikailag tárolják őket, és ezek a készletek a rendszer rendelési funkcióján keresztül érhetőek el.

4.2.1.7 Order

Az Order entitás a rendszer megrendeléseit írja le, egyedi azonosítójával lehetővé téve a rendelések pontos azonosítását és nyomon követését. Az entitás tartalmazza a megrendelt termékek listáját, valamint a hozzájuk tartozó státuszadatokat, amelyek jelzik a rendelés aktuális feldolgozottsági állapotát. Ezek az állapotok többek között lefedhetik a függőben lévő, teljesített vagy visszavont rendeléseket, biztosítva a megrendelések egyértelmű kezelését.

Az Order entitás szoros kapcsolatban áll a MyUser entitással, amely hozzárendeli a rendelést a megfelelő felhasználóhoz. Ez az összeköttetés garantálja, hogy a rendszer pontosan nyilvántartsa, mely felhasználók kezdeményezték az egyes rendeléseket.

4.3 API interfész

Az alkalmazás API-rétege a rendszer egyik legfontosabb eleme, amely biztosítja a felhasználói szerepkörök, funkciók és rendszerelemek közötti kommunikációt. Az API tervezése az API-first design módszertan szerint történt, amelyben az API specifikációi már a fejlesztés kezdetén kialakításra kerültek. Ez a megközelítés biztosította, hogy az alkalmazás funkcionalitása egyértelműen definiált és jól strukturált legyen, megkönnyítve a későbbi fejlesztéseket és integrációkat.

Az API specifikációi egyfajta szerződésként szolgáltak a rendszer különböző komponensei között, elősegítve a konzisztensebb architektúra kialakítását, amely pontosan meghatározta az egyes végpontok funkcióit. [9] Ez a módszer lehetővé tette a rendszer egyszerű integrálhatóságát bármilyen frontenddel vagy külső komponenssel. Az API design során külön hangsúlyt fektettem a felhasználókezelés és a jogosultsági szintek korai definiálására, amely a szerepkörök – Adminisztrátor, Beszállító és Felhasználó – pontos meghatározásával már a tervezés szakaszában hatékonyan támogatta a jogosultságok kezelését.

Az API logikai szerkezete moduláris felépítésű, amely jól elkülöníti a különböző funkciókat. Az egyes interfészek felelnek például a felhasználók hitelesítéséért, a rendelési műveletek kezeléséért, a készletkezelési műveletek támogatásáért vagy az általános rendszerinformációk eléréséért. Ez a modularitás biztosítja a rendszer funkcionalitásának átláthatóságát és könnyű bővíthetőségét. Az API interfészek részletes bemutatása a következő fejezetben található.

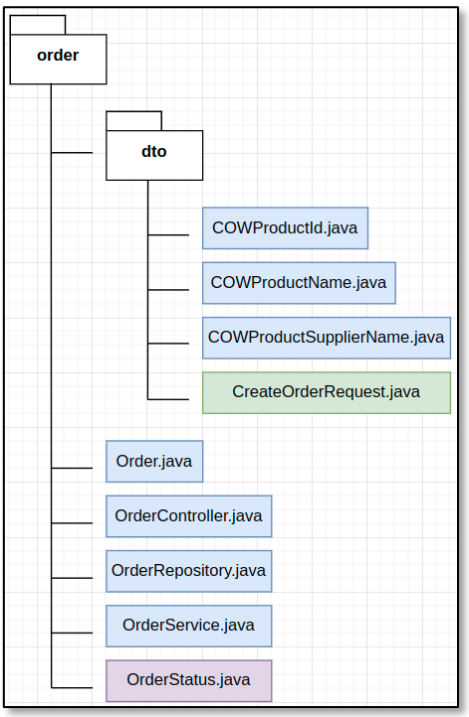
5 Megvalósítás

A következő fejezetben a megvalósított raktárkezelő alkalmazásom főbb építő elemeit és üzleti funkcióit fogom bemutatni.

5.1 Alkalmazás felépítése

5.1.1 Projekt struktúra

A projekt felépítését a funkcionalitások szerint strukturáltam, nem pedig a hagyományos, rétegek szerinti logika alapján. Ez a megközelítés lehetővé tette, hogy a kapcsolódó elemek, például a kontrollerek, szolgáltatások, adatelérési rétegek és adatszállító objektumok (DTO) egyetlen csomagon belül legyenek csoportosítva, amely egy adott funkciót fed le. Ezáltal a kód könnyebben kezelhetővé és átláthatóvá vált, mivel minden funkcióhoz tartozó komponens egyértelműen megtalálható.



4. ábra: Megrendelést megvalósító mappa struktúrája.
Kékkel a sima, zölddel az interfész, lilával pedig az enumerációs objektumok jelöltek.

A csomagstruktúrában minden funkcióhoz dedikált csomag tartozik, ahogy az a feltöltött ábrán is látható. Például az "order" csomagban található minden, ami a rendelésekkel kapcsolatos, beleértve az entitást (Order.java), a DTO-kat (például

CreateOrderRequest.java), a kontrollert (OrderController.java), a szolgáltatást (OrderService.java) és a repository interfészt (OrderRepository.java).

5.1.2 Függőségek

Az alkalmazás fejlesztéséhez a Maven build tool-t használtam, amely egyszerű és hatékony függőségkezelést biztosít. Az előbb említett függőségeket a Maven konfigurációs fájljában, a pom.xml-ben határoztam meg. A projekt függőségeinek meghatározásában a funkcionalitási és tesztelési igényekre helyeztem a hangsúlyt, miközben a Spring Boot ökoszisztéma által nyújtott előnyöket maximálisan kihasználtam. Bár számos más függőség is fontos szerepet játszott (például Spring Boot Starter - Security, Web, Data JPA, PostgreSQL, Jacoco, Sonarqube), ezeket nem ismertetném ismét.

```
<dependencies>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>${jjwt.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <version>${spring.boot.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <version>${spring.security.test.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>${h2db.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

A fenti pom.xml kódrészletben kiemeltem pár fontosabb függőséget. JWT Token kezelés (jjwt-api): Az alkalmazásban a felhasználók hitelesítése JSON Web Token (JWT) technológiával történik. Ehhez a jjwt-api csomagot használtam, amely lehetővé teszi a tokenek generálását és validálását. Spring Boot Starter Test: Az egység- és integrációs tesztek írásához a spring-boot-starter-test függőséget alkalmaztam, amely magában foglalja a JUnit, AssertJ és Mockito keretrendszereket, biztosítva a tesztelési folyamat hatékonyságát.

Spring Security Test: A jogosultságkezelési funkciók teszteléséhez a spring-security-test csomagot integráltam, amely segített a biztonsági mechanizmusok validálásában. H2 Database: Tesztelési környezetben egy beágyazott H2 adatbázist használtam, amely gyors és könnyen konfigurálható. Ez lehetővé tette az adatbázisfüggő funkciók izolált tesztelését.

5.1.3 Konfiguráció

A Spring Boot alkalmazások konfigurációit az application.yaml fájl tartalmazza, amely a projekt resources mappájában található. Ebben a fájlban adhatók meg az alkalmazás működését meghatározó különböző beállítások és paraméterek. Az application.yaml előnye, hogy jól tagolt és könnyen olvasható formátumban tartalmazza a konfigurációkat, ami megkönnyíti azok kezelését. Az itt megadott beállítások lehetővé teszik az alkalmazás bizonyos funkcióinak testreszabását anélkül, hogy újra kellene fordítani a programot.

```
spring:
  application:
    name:depot
  datasource:
    driverClassName: org.postgresql.Driver
  jpa:
    database: POSTGRESQL
    hibernate:
      ddl-auto: update

application:
  security:
    jwt:
      expiration: 86400000  # 1 day in milliseconds

custom:
  inventory:
    max-depot-space: 10000
    max-inventory-space: 1000
    should-check-expiration: true
  supplier:
    generate-random-password: false
```

Az alkalmazásom application.yaml fájlja az alapvető működéshez és testreszabhatósághoz szükséges konfigurációkat tartalmazza, amelyek különböző funkciókhoz nyújtanak támogatást. A Spring beállításai között megtalálható az adatforrás konfigurációja, amely meghatározza az adatbázis-illesztőprogramot (PostgreSQL) és a Hibernate automatikus sémakezelésének engedélyezését. Ezek a beállítások biztosítják az adatbázis kapcsolatának zavartalan működését és a sémák dinamikus frissítését.

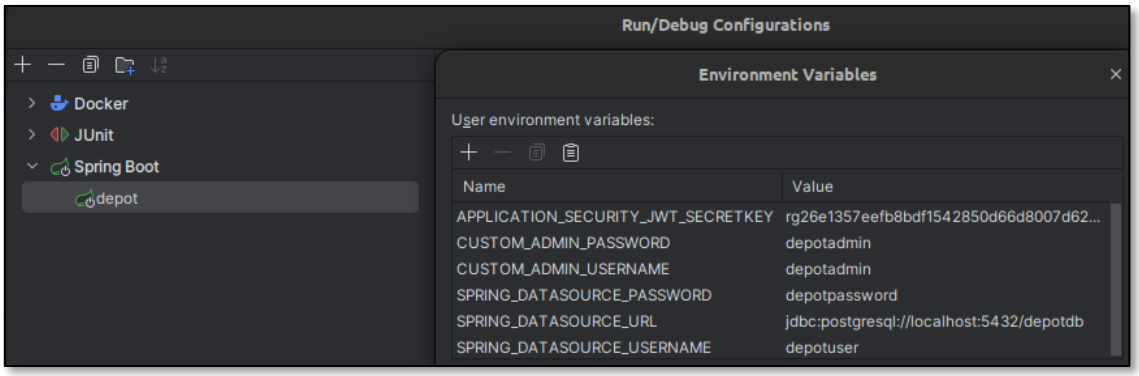
A biztonsági beállítások a JWT tokenek érvényességi idejét is definiálják, amely az alkalmazásban 24 órára van konfigurálva. Ez a paraméter a hitelesítési rendszer megbízhatóságának és hatékonyságának alapvető eleme.

Az üzleti logikát támogató egyedi beállítások a fájl custom blokkjában találhatók, amely tartalmazza például a készlet maximális kapacitásának beállítását és sok más fontos értéket, amelyeket egy későbbi fejezetben ismertetek részletesebben.

5.1.3.1 Futási konfiguráció

Az alkalmazás futtatásához szükséges érzékeny információk és konfigurációk biztonságos kezelésére a környezeti változók használata kulcsfontosságú volt. A fejlesztés során az IntelliJ IDEA indítási konfigurációjában definiáltam ezeket az értékeket, ezzel biztosítva, hogy az érzékeny adatok – például az adatbázis-hozzáférési információk vagy a titkosítási kulcsok – ne kerüljenek közvetlenül a kódbázisba. Ez a megközelítés minimálisra csökkentette a biztonsági kockázatokat, miközben lehetővé tette a dinamikus konfigurációt futási időben.

A legfontosabb környezeti változók közé tartozott a SPRING_DATASOURCE_URL, amely az adatbázis elérésének alapvető paramétereit határozta meg, az APPLICATION_SECURITY_JWT_SECRETKEY, amely a JSON Web Tokenek generálásához és aláírásához biztosított titkos kulcsot, valamint a CUSTOM_ADMIN_PASSWORD, amely a kezdeti adminisztrátori fiókhoz tartozó jelszóként szolgált. Ezek a változók biztosították, hogy az alkalmazás biztonságos és rugalmas módon legyen konfigurálható különböző környezetekben, miközben az érzékeny adatok nem kerültek a verziókövetési rendszerbe.



5. ábra: Applikáció futási konfigurációja IntelIJ IDE-ben.

5.2 Web biztonság

Az alkalmazás webbiztonsági rétegét a Spring Security segítségével valósítottam meg, amely lehetőséget ad a biztonsági szabályok rugalmas és testreszabott kezelésére. A biztonsági beállításokat a SecurityConfig osztályban definiáltam, ahol a különböző biztonsági mechanizmusokat és szabályokat határoztam meg. Ezek a szabályok az API végpontok elérését felügyelik, garantálva az alkalmazás integritását és a felhasználói jogosultságok megfelelő kezelését.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
  http
    .csrf().disable()
    .cors().disable()
    .authorizeHttpRequests(req ->
      req.requestMatchers(WHITE_LIST_URL)
        .permitAll()
        .requestMatchers("/user/**").hasRole(ADMIN.name())
        .requestMatchers("/admin/**").hasRole(ADMIN.name())
        .requestMatchers("/report/**").hasRole(ADMIN.name())
        .requestMatchers("/info/**")
          .hasAnyAuthority(USER_READ.getPermission())
        .requestMatchers(HttpMethod.POST, "/order")
          .hasAnyAuthority(USER_CREATE.getPermission())
        .requestMatchers("/order/**").hasRole(ADMIN.name())
        .requestMatchers("/supplier/**").hasRole(SUPPLIER.name())
        .anyRequest()
          .authenticated()
    )
    .sessionManagement(session-> session.sessionCreationPolicy(STATELESS))
    .authenticationProvider(authenticationProvider)
    .addFilterBefore(jwtAuthFilter,
      UsernamePasswordAuthenticationFilter.class);
  return http.build();
}
```

A WHITE_LIST_URL tömbben szerepelnek azok a végpontok, amelyek autentikáció vagy külön jogosultság nélkül is elérhetők. Ide tartoznak például:

- Az autentikációs végpontok (pl. /auth/),
- A dokumentációs felülethez tartozó URL-ek (pl. Swagger),
- A rendszerfigyeléshez szükséges végpontok (pl. Actuator).

Mivel az alkalmazás jelenleg nem tartalmaz frontend komponenst, a CORS (Cross-Origin Resource Sharing) és CSRF (Cross-Site Request Forgery) védelmet kikapcsoltam.

- CORS akkor releváns, ha a frontend külön domainről próbál hozzáférni az erőforrásokhoz.
- CSRF akkor szükséges, ha a kliens böngészőalapú és session alapú hitelesítést használ, amely ebben az alkalmazásban nem releváns.

5.2.1 Jogosultsági szabályok

A nem fehérlistázott végpontok elérését szigorú szabályokhoz kötöttem, amelyeket az authorizeHttpRequests metódusban definiáltam:

- Az adminisztrációs végpontok (pl. /admin/, /report/) kizárólag az Admin szerepkörrel rendelkező felhasználók számára elérhetők.
- Az információs végpontokhoz (pl. /info/) csak azok férhetnek hozzá, akik rendelkeznek az olvasási jogosultságokkal (pl. USER_READ).
- Az új rendelés leadása (pl. POST /order) minden regisztrált felhasználó és beszállító számára elérhető, mivel ez az alkalmazás alapvető funkciója.
- A beszállítói műveletek (pl. /supplier/) kizárólag a Supplier szerepkörrel rendelkező felhasználók számára érhetők el.

Az alkalmazás stateless módon kezeli a munkameneteket, azaz minden kérés független az előzőektől. Ez különösen fontos a JWT alapú hitelesítés használata miatt, mivel a session-állapotot nem a szerver, hanem maga a token tárolja. A kliens minden kéréshez csatolja a JWT tokent, amelyet a rendszer minden alkalommal ellenőriz. [10]

5.3 Adatforrás réteg

Az adatelérési réteg az alkalmazás egyik legfontosabb komponense, amely biztosítja a perzisztens adatok kezelését és tárolását. Ennek megvalósításához a Spring Data JPA keretrendszert használtam, amely egy beépített ORM megoldást kínál. Ez lehetővé teszi az adatbázisban tárolt adatok egyszerű és hatékony Java objektumokká történő leképezését.

5.3.1 Entitások

Előző fejezetben már részletesen leírtam minden entitás feladatát, most egy példán keresztül bemutatom, milyen az implementációja az Inventory entitásnak.

```
@Entity
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Inventory {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToOne(mappedBy = "inventory", cascade = CascadeType.ALL)
  private Supplier supplier;

  private int usedSpace;
  private int maxAvailableSpace;
  private int lowStockAlertThreshold;
  private int expiryAlertThreshold; // in days
  private int reorderThreshold;
  private int reorderQuantity;

  @ElementCollection(fetch = FetchType.EAGER)
  @CollectionTable(name="inventory_products",joinColumns =
  @JoinColumn(name = "inventory_id"))
  @Column(name = "product_id")
  private List<Long> productIds = new ArrayList<>();

  @CreationTimestamp
  private LocalDateTime createdAt;

  @UpdateTimestamp
  private LocalDateTime updatedAt;
}
```

Az Inventory entitás a készlet kezeléséhez szükséges legfontosabb adatokat tárolja, és kialakítása biztosítja a hatékony adatkezelést. Az entitás azonosítója, az id mező, automatikusan generált egyedi azonosítót kap a @Id és @GeneratedValue annotációk segítségével, amely garantálja az egyediséget az adatbázisban. Az Inventory és a beszállítót reprezentáló Supplier entitás között @OneToOne kapcsolat áll fenn, amely meghatározza, hogy minden készlet egy adott beszállítóhoz tartozik. A kapcsolat a mappedBy attribútum segítségével irányított, jelezve, hogy a beszállító az irányító oldal.

A készlet kapacitását és állapotát olyan mezők tartják nyilván, mint a usedSpace (használt tárhely) és a maxAvailableSpace (maximális tárhely), valamint a különböző riasztási küszöbértékek, például az alacsony készletre és a lejárati figyelmeztetésre vonatkozó beállítások. Ezek a mezők a proaktív készletkezelést és a készletállapot folyamatos monitorozását támogatják.

A termékek azonosítóit a productIds mező tárolja, amely az adatbázisban külön inventory_products táblában kerül kezelésre az @ElementCollection és @CollectionTable annotációk segítségével. Az entitás ezen felül automatikusan rögzíti a

létrehozás és az utolsó módosítás időpontját a @CreationTimestamp és @UpdateTimestamp annotációkkal, biztosítva az adatok naprakészségét és nyomon követhetőségét.

5.3.2 Adatelérési réteg interfészek

A Spring Data JPA megkönnyíti az adatbázissal való műveletek végrehajtását, mivel a fejlesztőket mentesíti az alacsony szintű adatbáziskezelési feladatok megírása alól. A JPA-repository interfészek segítségével a standard CRUD (létrehozás, olvasás, változtatás, törlés) -műveletek automatikusan rendelkezésre állnak, és az egyedi lekérdezések is könnyen definiálhatók.

Az alkalmazásban a InventoryRepository interfész felelős az Inventory entitáshoz kapcsolódó adatbázisműveletekért. Az interfész a JpaRepository kiterjesztésével örökli a gyakran használt műveleteket, mint például az entitások mentése, törlése vagy azonosító szerinti lekérdezése.

Az adatelérési rétegben néha szükség van egyedi adatbázisműveletek definiálására, amelyeket az alapértelmezett JPA-metódusok nem támogatnak. Az InventoryRepository példaként egyedi lekérdezést is tartalmaz:

```
@Repository
public interface InventoryRepository extends JpaRepository<Inventory, Long>
{
  @Query("SELECT DISTINCT i FROM Inventory i
          JOIN Product p ON p.id MEMBER OF
          i.productIds WHERE p.schema.id = :productSchemaId")
  List<Inventory> findAllByProductSchemaId
    (@Param("productSchemaId") Long productSchemaId);
}
```

Ez a lekérdezés az Inventory entitások és a hozzájuk kapcsolódó termékek kapcsolatát vizsgálja. A működése során az Inventory entitásokat a hozzájuk tartozó Product entitásokkal kapcsolja össze a productIds lista alapján. Csak azokat az Inventory elemeket adja vissza, amelyekhez tartozik olyan termék, amely megfelel a megadott terméksablon azonosítójának.

A metódus megvalósításához a Spring JPA több hasznos funkcióját alkalmaztam. Az @Repository annotáció biztosítja, hogy az interfész automatikusan a Spring adatelérési rétegének része legyen. Az @Query annotáció segítségével egyedi JPQL lekérdezést definiáltam, amely lehetővé tette a testreszabott adatbázis-műveleteket, míg az @Param annotáció gondoskodott a paraméterek pontos párosításáról a lekérdezés változóival.

A raktárkezelő alkalmazásomban az InventoryRepository mellett több más adatelérési interfészt is megvalósítottam:

1. UserRepository: A felhasználók adatainak kezeléséért felelős.
2. ProductRepository: A rendszerben tárolt termékek kezelését végzi.
3. ProductSchemaRepository: A terméksablonok kezelését biztosítja.
4. OrderRepository: A megrendelések adatainak elérését kezeli.
5. CategoryRepository: A termékkategóriákhoz kezelését végzi.
6. SupplierRepository: A beszállítók adatainak kezelését látja el.
7. InventoryRepository: A készletekkel kapcsolatos adatbázisműveleteket végzi.

5.4 REST elérési pontok

Az alkalmazás szerveroldali REST API-ja biztosítja a kommunikációt a kliensoldali és szerveroldali komponensek között, HTTP kéréseken és válaszokon keresztül. Az API kialakításához a Spring Boot @RestController annotációját használtam, amely egyszerűvé és átláthatóvá teszi az osztályok REST végpontokká alakítását.

5.4.1 RestContoller osztályok

5.4.1.1 AuthController

Az AuthController osztály az autentikációs funkciók végrehajtására szolgál, biztosítva a felhasználók regisztrációját és bejelentkezését. Ezen osztály végpontjai nyilvánosan hozzáférhetők, mivel nem igényelnek autentikációs tokent.

- POST /auth/register: Ez a végpont lehetővé teszi új felhasználók regisztrációját a rendszerben. Válaszként egy JWT tokent ad.
- POST /auth/login: A bejelentkezési végpont a rendszerben már regisztrált felhasználók számára biztosít hozzáférést. Válaszként egy új JWT tokent ad.

5.4.1.2 UserController

A UserController osztály az adminisztrátorok által végrehajtható felhasználókezelési műveleteket valósítja meg. Az itt található végpontok kizárólag az Admin szerepkörrel rendelkező felhasználók számára érhetők el.

- GET /user: Ez a végpont lehetővé teszi az összes felhasználó adatainak lekérdezését.
- GET /user/{id}: A megadott azonosítójú felhasználó adatainak lekérdezésére szolgál.
- PUT /user/{id}: Lehetővé teszi egy meglévő felhasználó adatainak módosítását a megadott azonosító alapján.
- DELETE /user/{id}: Ez a végpont egy felhasználó törlésére szolgál az azonosítója alapján.

5.4.1.3 AdminControllers

Az adminisztrációs funkciókat három különálló kontroller kezeli: SupplierAdminController, CategoryAdminController, és ProductSchemaAdminController. Ezek kizárólag az Admin szerepkörrel rendelkező felhasználók számára érhetők el, és biztosítják a beszállítók, kategóriák, valamint terméksablonok kezelését CRUD (Create, Read, Update, Delete) műveletek végrehajtásával.

SupplierAdminController

A beszállítók adatainak kezeléséért felelős CRUD műveleteket biztosítja. Az /admin/supplier/{id} URL-hez kapcsolódó metódusok lehetővé teszik a beszállítók adatainak létrehozását, lekérdezését, módosítását, valamint törlését.

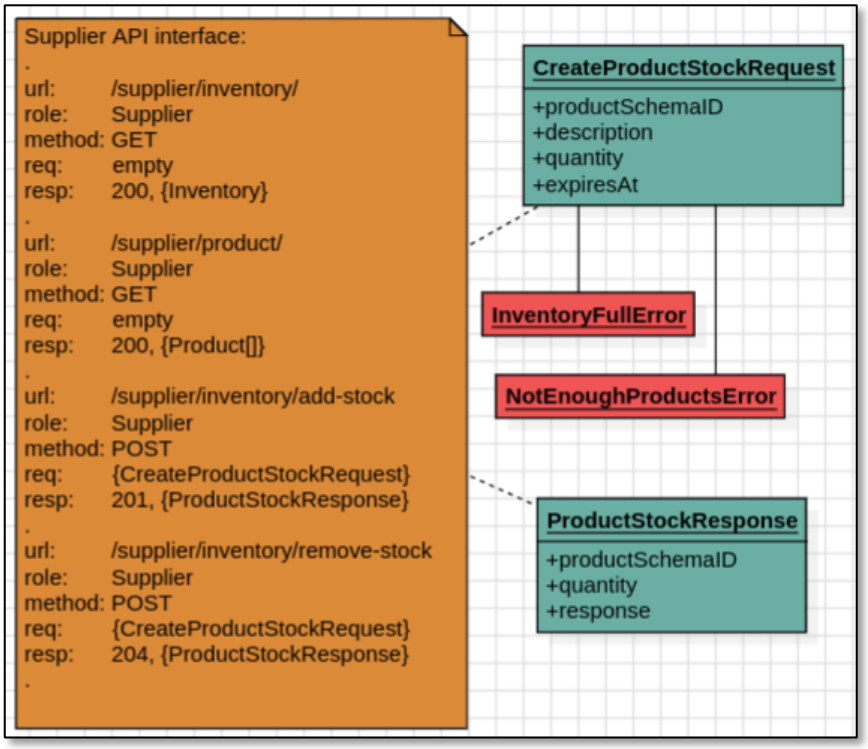
CategoryAdminController

A kategóriák kezelését végző CRUD műveleteket valósítja meg. Az /admin/category/{id} végponthoz tartozó metódusok biztosítják az egyes kategóriák létrehozását, lekérdezését, módosítását, valamint törlését.

ProductSchemaAdminController

A terméksablonok kezelését végző CRUD műveletekért felelős. Az /admin/product-schema/{id} URL-t kezelő metódusok lehetővé teszik új sablonok létrehozását, a meglévők módosítását, lekérdezését, valamint törlését.

5.4.1.4 SupplierStockController



6. ábra: Beszállítók API interfésze.
Narancsszín az API deffiníció, kék a DTO objektumok, piros a hibaüzenetek jelölési színe.

A SupplierStockController osztály a beszállítók készletkezelési műveleteit valósítja meg. A végpontok kizárólag a Supplier szerepkörrel rendelkező felhasználók számára érhetők el. Az adatokat a backend és a kliensoldal között DTO (Data Transfer Object) objektumok segítségével továbbítja.

- GET /supplier/inventory/: Lehetővé teszi a beszállítók számára saját készletük adatainak lekérdezését.
- GET /supplier/product/: A beszállító által eddig raktározott összes termék listáját adja meg.
- POST /supplier/inventory/add-stock: Termék készlet hozzáadását teszi lehetővé egy CreateProductStockRequest DTO használatával.
- POST /supplier/inventory/remove-stock: Termék készlet eltávolítására szolgál. A kérés szintén egy CreateProductStockRequest DTO-t fogad.

A bemenethez és kimenethez használt DTO-k egyszerű, de rugalmas struktúrával rendelkeznek, például a CreateProductStockRequest DTO a terméksablon azonosítóját (productSchemaID), a hozzáadandó vagy eltávolítandó mennyiséget (quantity), valamint

a termék lejárati idejét (expiresAt) tartalmazza. A kimeneti adatok, például a ProductStockResponse, pedig részletes visszajelzést adnak a készletmódosítás eredményéről.

5.4.1.5 OrderController

Az OrderController osztály a rendelésekkel kapcsolatos funkciókat valósítja meg, és eltérő szerepkörű felhasználók számára nyújt hozzáférést. Az adminisztrátorok részletesen megtekinthetik az összes rendelést, míg a felhasználók számára elérhetővé tettem a rendelés létrehozását és törlését.

Általános rendelési végpontok

- GET /order: Az adminisztrátorok lekérhetik az összes rendelés adatait.
- GET /order/{id}: Az adminisztrátorok részletes információt kaphatnak egy konkrét rendelésről az azonosító alapján.
- POST /order: A felhasználók új rendelést hozhatnak létre, amely során egy List<CreateOrderRequest> típusú objektumot adnak át a rendelés részleteivel.
- DELETE /order/{id}: A felhasználók törölhetik saját rendeléseiket az azonosító megadásával.

Függő rendelések kezelése (Pending Orders)

- GET /order/pending-order: Az adminisztrátorok lekérhetik az összes függő állapotú rendelést.
- GET /order/pending-order/{id}: Az adminisztrátorok egy adott függő rendelés részleteit kérhetik le az azonosító alapján.
- POST /order/pending-order/{id}: Az adminisztrátorok véglegesíthetik vagy törölhetik az adott rendelést.

POST /order végpont

Tervezésekor célom az volt, hogy a felhasználók számára rugalmas és intuitív módot biztosítsak a rendelés létrehozására. Sok időt fordítottam annak mérlegelésére, hogyan lehetne ezt megvalósítani úgy, hogy a kód ne váljon túlságosan összetetté.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")
@JsonSubTypes({
  @JsonSubTypes.Type(value = COWProductId.class,
          name = "productId"),
  @JsonSubTypes.Type(value = COWProductName.class,
```

```
            name = "productName"),
    @JsonSubTypes.Type(value = COWProductSupplierName.class,
            name = "productSupplierName")
})
public interface CreateOrderRequest {
  ...
}
```

Végül a CreateOrderRequest interfészen alapuló megoldást választottam, amely lehetővé teszi, hogy a felhasználók különböző módokon adhassák meg a rendeléshez szükséges adatokat:

- Termék azonosító alapján (COWProductId), típus: “productId”
- Termék név alapján (COWProductName), típus: “productName”
- Termék és beszállító neve alapján (COWProductSupplierName), típus: “productSupplierName”

Bemeneti JSON objektumoknál mindig meg kell adni az adott típusnak a nevét, ez alapján tudja a OrderController serializálni őket java objektumokká. Példa bemenet:

```
[
    {"type": "productSupplierName",
    "supplierName": "SAMSUNG",
    "productName": "Samsung_Galaxy_S24",
    "quantity": 5
    }
]
```

Ennek köszönhetően a felhasználók az igényekhez igazodó egyéni rendeléseket tudnak leadni, melyek fogadására teljesen fel van készítve a szerver oldali applikáció.

GET /order/pending-order végpont

A rendelési rendszer tervezése során külön figyelmet fordítottam arra, hogy biztonságos és reális megoldást nyújtsak. Ennek érdekében beépítettem egy extra biztonsági réteget, amely lehetővé teszi az adminisztrátorok számára, hogy bizonyos függő rendelések státuszát véglegesíthessék vagy törölhessék.

Ez a döntés lehetőséget ad arra, hogy az adminisztrátorok megakadályozzák a nem indokolt, túl nagy mennyiségű rendeléseket, amelyek veszélyeztethetnék a raktár működését. Az ilyen mechanizmus hozzájárul a készlet tudatos és biztonságos kezeléséhez, így csökkentve annak az esélyét, hogy a raktár kritikus szintre kerüljön. Ezzel a megoldással a rendszer rugalmas, de egyben ellenőrzött marad, és minden rendelés esetében biztosított a megfelelő döntéshozatal lehetősége.

5.4.1.6 InfoController

Az InfoController osztály a felhasználók számára általános információk elérését teszi lehetővé. Ezek a végpontok az User szerepkörrel rendelkező felhasználók számára hozzáférhetők.

- GET /info/user/me: A jelenleg bejelentkezett felhasználó adatait adja vissza.
- GET /info/order: A felhasználó összes megrendelésének listáját adja meg.
- GET /info/product: A rendszerben elérhető összes termék listáját szolgáltatja.
- GET /info/product/category/{id}: A rendszerben elérhető összes, kiválasztott kategóriával rendelkező, termékek listáját adja vissza.
- GET /info/supplier: A beszállítók listáját adja vissza, akik a rendszerben szerepelnek.

5.4.1.7 ReportController

A ReportController osztály az adminisztrátorok számára biztosít hozzáférést különböző jelentések generálásához. Ezek a végpontok lehetővé teszik a készlet és a rendelések állapotának részletes elemzését.

- GET /report/inventory/state: Az aktuális készlet állapotáról generál jelentést, amely tartalmazza a raktárkészlet főbb adatait.
- GET /report/inventory/expiry: A készletben található termékek lejárati állapotáról készít jelentést, például mely termékek fognak hamarosan lejárni.
- GET /report/order: Az összes rendelésről aggregált jelentést készít, amely tartalmazza a rendelések főbb statisztikáit és részleteit.

5.5 Üzleti logika

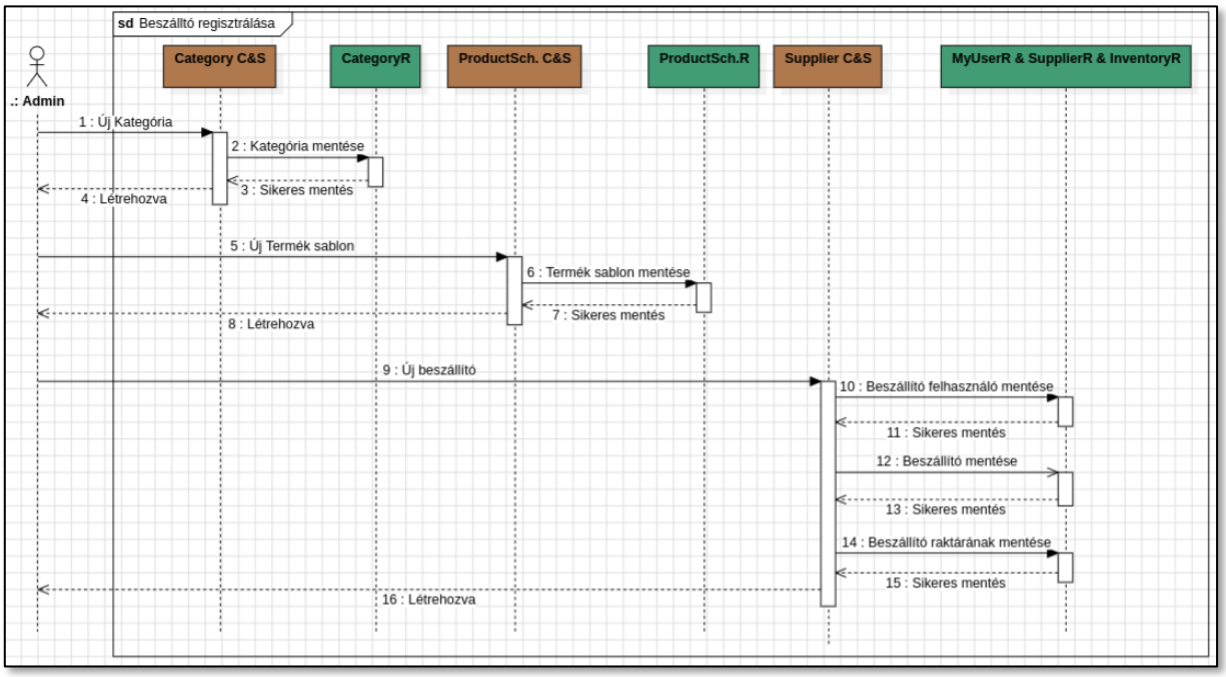
Az alkalmazás üzleti logikájának megvalósításáért a Service osztályok felelnek, amelyek a Spring keretrendszer által kezelt beanként működnek. Ezek az osztályok tartalmazzák az alkalmazás alapvető folyamatainak és szabályainak implementációját, biztosítva a különböző rétegek közötti tiszta elválasztást és a kód átláthatóságát.

Ahelyett, hogy az összes Service osztály részleteit külön-külön ismertetném, a következőkben az alkalmazás kulcsfontosságú rendszerfolyamatait mutatom be. Ez a

megközelítés jobb rálátást ad arra, hogyan működnek együtt az üzleti logika különböző elemei az alkalmazás egészének támogatása érdekében. A bemutatott folyamatok részletezik a rendszer inicializálását, a termékek beszállítását, a rendelések kezelését, az automatikus értesítések küldését, valamint a riportok generálását.

5.5.1 Rendszer inicializálása

Az alábbi ábrán bemutatom a beszállítók regisztrációjának folyamatát, amely magában foglalja a kategóriák és terméksablonok létrehozását, valamint a beszállítók adatainak és raktárkapacitásának rögzítését.



7. ábra: Rendszer inicializálása.
C&S a controller és service, az R pedig repository komponenseket jelöli.

A rendszer biztonsága érdekében kizárólag az adminisztrátor jogosult új kategóriák, terméksablonok és beszállítók létrehozására. Ezzel garantált, hogy csak az előre meghatározott szabályoknak megfelelő adatok kerüljenek be a rendszerbe.

A folyamat lépései:

1. Kategória létrehozása: Az adminisztrátor új kategóriát hoz létre, amelyet a rendszer a CategoryRepository segítségével tárol az adatbázisban. Sikeres mentés után a rendszer visszajelzést ad.

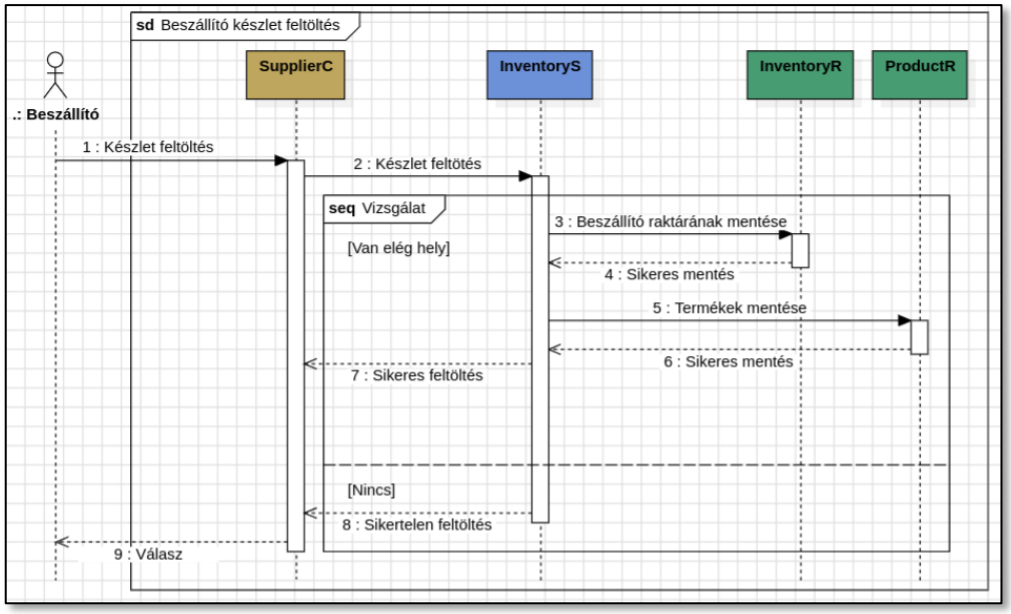
2. Terméksablon létrehozása: Az adminisztrátor létrehoz egy új terméksablont, amely meghatározza az adott termék tárolási követelményeit és kategóriáját. Ezt a rendszer a ProductSchemaRepository-ban menti.
3. Beszállító regisztrálása: Az adminisztrátor rögzíti az új beszállító adatait. A rendszer először a beszállító felhasználói fiókját menti a MyUserRepository-ba, majd a beszállítói entitást és hozzá tartozó raktáradatokat a SupplierRepository és az InventoryRepository segítségével.

Ez a folyamat biztosítja, hogy minden beszállító, kategória és terméksablon megfeleljen az előzetesen meghatározott szabályoknak, csökkentve az inkonzisztens adatok megjelenésének kockázatát.

5.5.2 Termékek beszállítása

A termékek beszállítását kizárólag a beszállítók végezhetik, garantálva a rendszer integritását és a jogosultságok helyes kezelését. A beszállítók kizárólag a saját, számukra elkülönített raktárterületeket használhatják, ezzel biztosítva, hogy csak a nekik kijelölt készleteket kezelhetik. Az adminisztrátorok kizárása a beszállítók készletfeltöltési és kiszállítási folyamataiból megakadályozza a manipulációkat és visszaéléseket, hozzájárulva a rendszer tiszta és megbízható működéséhez.

Ez a megközelítés lehetővé teszi, hogy a beszállítók kizárólag ellenőrzött és valós körülmények között kezelhessék a készleteiket, minimalizálva a helyhiány vagy a túlterhelés kockázatát. A rendszer tiszta felelősségi köröket biztosít.



8. ábra: Beszállító készlet feltöltése.

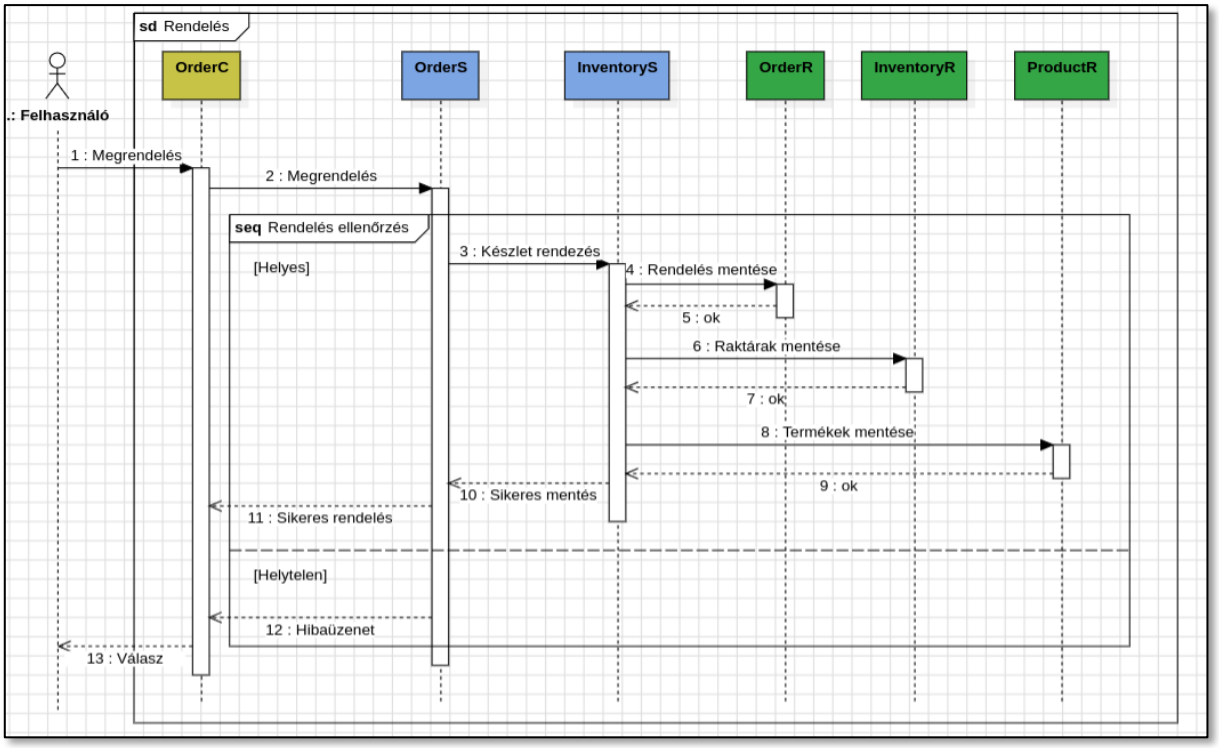
Előző ábrához hasonlóan, C controller, S service, R repository komponenseket rövidíti.

Diagram folyamatai:

1. A beszállító egy készletfeltöltési kérés küldésével kezdeményezi a folyamatot.
2. A rendszer ellenőrzi, hogy van-e elegendő hely a beszállító raktárában az új termékek számára.
    2.1. Ha elegendő hely áll rendelkezésre, a beszállító raktárának adatai frissülnek, majd az új termékek is rögzítésre kerülnek az adatbázisban. A rendszer ezután visszaigazolást küld a sikeres feltöltésről.
    2.2. Ha nincs elegendő hely, a rendszer elutasítja a kérést, és visszajelzést ad a sikertelenségről.
3. A folyamat minden lépését a megfelelő repository-k, például az InventoryRepository és ProductRepository, biztosítják.

5.5.3 Rendelés leadása

A rendelés leadása a rendszer egyik alapvető folyamata, amelyet bármely felhasználó – legyen az normál felhasználó (User), beszállító (Supplier), vagy adminisztrátor (Admin) – elindíthat. Ez azért lehetséges, mert a jogosultságkezelésben a Supplier és az Admin is rendelkezik a User jogosultságaival is.



9. ábra: Rendelés leadása.

Diagram folyamatai:

1. A rendelési igényt a felhasználó elküldi, amelyet a rendszer az OrderService segítségével ellenőriz.
2. Helytelen rendelés esetén (például nincs elegendő termék vagy hibás adatok) a folyamat hibaüzenettel zárul.
3. Helyes rendelés esetén:
    1.1. Az OrderService rögzíti a rendelést az OrderRepository segítségével, a rendelés státusza függőben lévő (Pending) lesz.
    1.2. Az InventoryService a rendelésben szereplő termékek státuszát foglalt (Reserved) állapotra változtatja, így más rendelés már nem érheti el azokat.
    1.3. A rendszer visszajelzést küld a felhasználónak a sikeres rögzítésről.

5.5.3.1 Termék kiválasztás

A rendszer háromféleképpen engedi a rendelés leadását, attól függően, hogy a felhasználó milyen adatokat ad meg:

1. Csak a termék neve van megadva:

    1.1. Minden beszállítót ellenőriz a rendszer, hogy szállítja-e az adott terméket.
    1.2. A rendszer megtalálja azt a beszállítót, akinél van lejárathoz legközelebbi, még nem foglalt kiválasztott termék, és az ő termékeit helyezi a rendelésbe.

2. A termék neve és a beszállító is meg van adva:

    2.1. A rendszer csak a megadott beszállító termékei között keres.
    2.2. A kiválasztás logikája ugyanaz: a lejárathoz legközelebbi, nem foglalt termékek kerülnek a rendelésbe.

3. Termék azonosító van megadva:

    3.1. Ha a megadott azonosítójú termék nem foglalt, akkor a rendszer a rendeléshez rendeli.

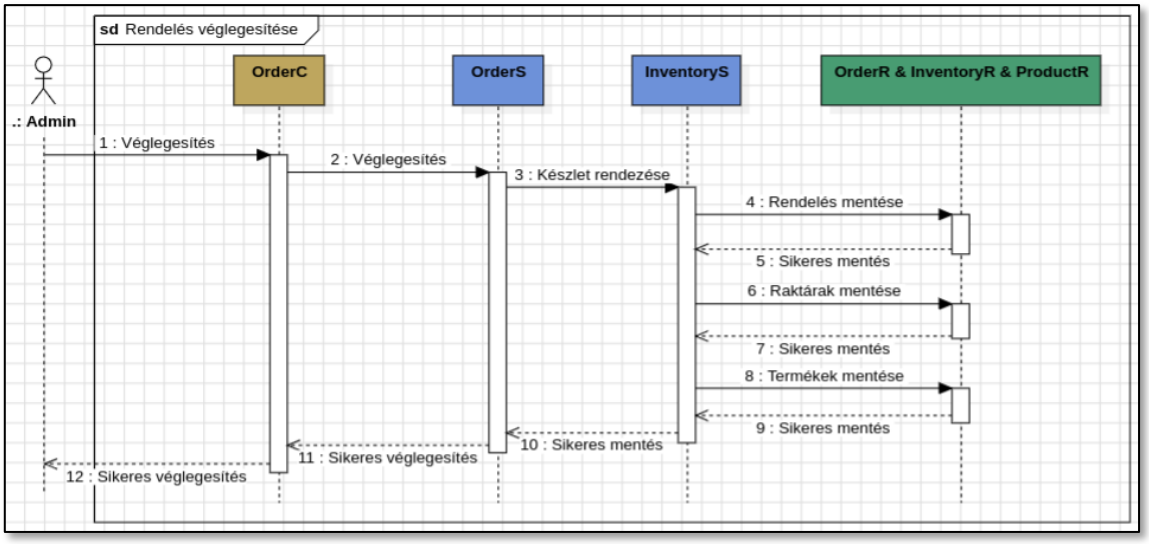
Ha egy rendelésben megadott mennyiséget egy beszállító nem tud teljesíteni:

1. A rendszer több beszállító készletét kombinálva tölti fel a rendelést.
2. Elsőként a lejárathoz legközelebbi termékekkel rendelkező beszállító készletét használja fel.
3. Ha még mindig hiány van, a következő beszállító készletéből pótolja, és így tovább, amíg a rendelés teljesül.

5.5.4 Rendelés véglegesítése

A rendelés véglegesítésének folyamata az adminisztrátor szerepkörű felhasználók számára elérhető kritikus funkció, amely biztosítja, hogy csak a megfelelően ellenőrzött és jóváhagyott rendelések kerüljenek teljesítésre. Ez a művelet kizárólag függőben lévő (Pending) rendeléseken hajtható végre, ezzel megakadályozva, hogy bármilyen, nem engedélyezett rendelés véletlenül teljesítésre kerüljön.

Ugyanakkor a véglegesítés folyamata nemcsak az adminisztrátor által történő elfogadást foglalja magában, hanem lehetőséget biztosít a rendelés visszavonására is. Ez utóbbi szintén kulcsfontosságú az alkalmazás integritásának megőrzésében.



10. ábra: Rendelés véglegesítése.

5.5.4.1 Rendelés elfogadása

Az adminisztrátor jóváhagyása során a rendelés státusza véglegesített (Completed) állapotra változik, míg a rendelésben lévő termékek státusza eltávolított (Removed) állapotot kap. Ez a változás biztosítja, hogy a teljesített rendeléshez tartozó termékek már ne legyenek elérhetők más rendelések számára. A rendszer az állapotváltozásokat az OrderRepository és InventoryRepository segítségével rögzíti, biztosítva a folyamat nyomon követhetőségét.

5.5.4.2 Rendelés visszavonása

A rendelés visszavonása vagy elutasítása során a rendelés státusza törölt (Cancelled) állapotra változik, míg a benne szereplő termékek státusza visszaáll szabad (Free) állapotra. Ez lehetővé teszi, hogy a termékek újra elérhetők legyenek más rendelések számára, biztosítva a készletek optimális kihasználását. A folyamatot az adminisztrátor vagy a rendelést leadó felhasználó kezdeményezheti, amelyet az OrderService ellenőriz.

5.5.5 Automatikus értesítések

Az automatikus értesítések biztosítják, hogy a készletkezelés mindig naprakész és megbízható legyen. Ezek az értesítések az application.yaml fájlban konfigurálhatók, ahol a funkciók, mint például az auto-reorder (automatikus utánrendelés), a low-stock (alacsony készlet) és az expiration (lejárat figyelése), egyszerűen ki- vagy bekapcsolhatók. Az értesítések működése a beszállítók létrehozásakor megadott egyedi

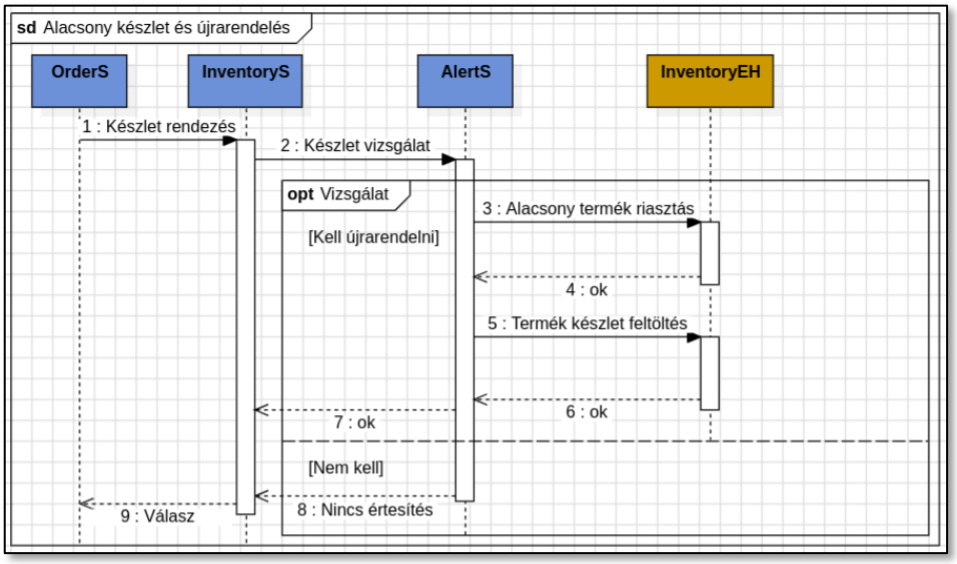
küszöbértékekhez igazodik, mint például az expiryAlertThreshold (lejárati riasztás küszöbértéke) vagy a lowStockAlertThreshold (alacsony készlet küszöbértéke).

```
spring:
  mail:
    host: localhost
    port: 1025

custom:
  alert:
    auto-reorder: true
    low-stock: true
    expiration: true
    expiry-check-interval-ms: 60000 # 1 hour in milliseconds
    mail:
      should-send: true
      from: depot.admin@localhost
```

A levelezési szolgáltatás helyi Simple Mail Transfer Protocol (SMTP) szerverre (*localhost:1025*) van konfigurálva a jelenlegi applikációmban, ami nem végleges megoldás, de elégségesen szimulál egy valós környezetet. Levélben történő értesítés csak tényleges, figyelmet igénylő helyzet esetén történik meg.

5.5.5.1 Alacsony készlet és utánrendelése



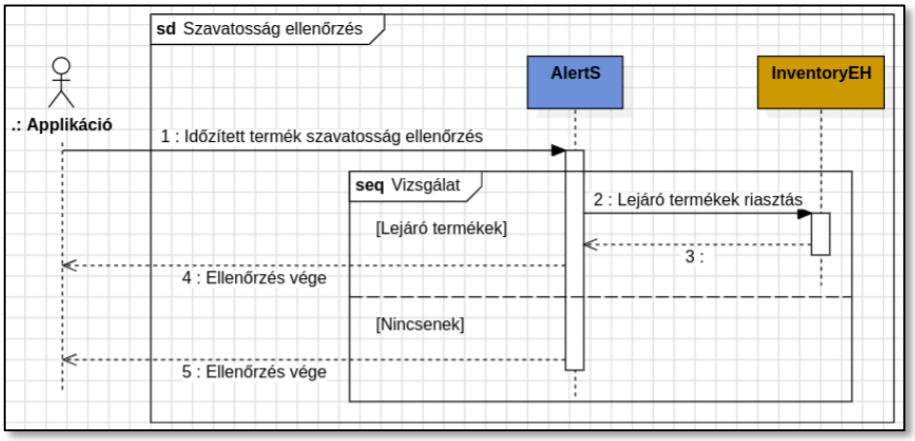
11. ábra: Alacsony készlet és újrarendelése.
S service, EH event handler (esemény kezelő) komponenseket rövidít.

Az alacsony készlet és utánrendelés ellenőrzés a megrendelés elfogadása folyamat közben, (10-es ábra 9-es és 10-es lépése között) zajlik. Az InventoryService a készletek állapotát vizsgálja, összehasonlítva a termékek szintjét az előre meghatározott küszöbértékekkel, mint az alacsony készletszint (lowStockAlertThreshold) és az

utánrendelési küszöb (reorderThreshold). Ha valamelyik termék szintje nem éri el ezeket az értékeket, az AlertService riasztást generál.

Alacsony készletszint esetén az AlertService értesíti az érintett beszállítókat a hiányzó termékekről és utánrendelési javaslatokat küld. Ha az automatikus utánrendelés (auto-reorder) engedélyezett, a rendszer azonnal elindítja a szükséges rendeléseket. Az e-mail értesítéseket a Spring Boot Starter Email segítségével küldtem, amely garantálja a gyors és pontos kézbesítést.

5.5.5.2 Szavatosság ellenőrzése



12. ábra: Szavatosság ellenőrzése.

Az alkalmazás a termékek szavatosságát automatikusan és periodikusan ellenőrzi, ezzel biztosítva a készlet frissességét és a lejárt termékek gyors kiszűrését. Az ellenőrzés a Spring Scheduled mechanizmusát használja, amely az application.yaml konfigurációs fájlban meghatározott időközök szerint fut le.

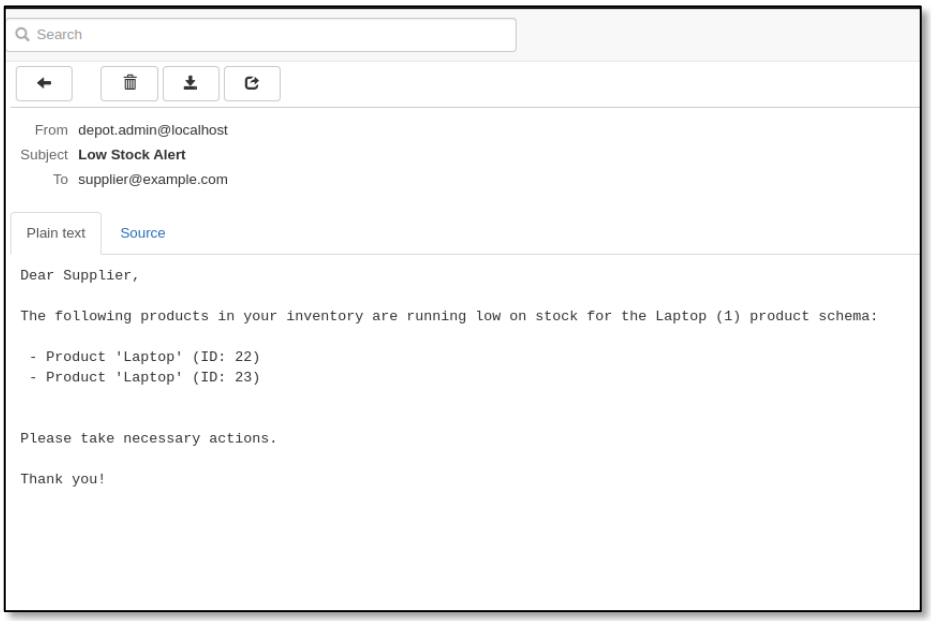
A funkció megvalósítása a következő kód segítségével történt:

```
@Value("${custom.alert.expiration}")
private boolean EXPIRY_ALERT_ENABLED;

@Scheduled(fixedRateString = "${custom.alert.expiry-check-interval-ms}",
initialDelay = 10000)
@Transactional
public void checkForExpiredProducts() {
  if (!EXPIRY_ALERT_ENABLED) {
    return;
  }
  ...
}
```

@Scheduled annotáció segítségével a rendszer a szavatosság ellenőrzését meghatározott időközönként automatikusan végrehajtja, a logikát pedig csak akkor

futtatja, ha az EXPIRY_ALERT_ENABLED paraméter engedélyezett. [11] Ez rugalmas és jól konfigurálható megoldást nyújt a szavatossági problémák kezelésére.



13. ábra: Alacsony készletszint figyelmeztető email.

5.5.6 Jelentés készítés

A jelentések készítése alapvető funkciója az alkalmazásnak, amely gyors, áttekinthető és könnyen feldolgozható adatokat biztosít a rendszer aktuális állapotáról. A jelentések Java objektumok formájában készülnek, amelyeket JSON formátumba serializál a rendszer. Az API interfész pontos specifikációja alapján az adatokhoz kapcsolódó komponensek feldolgozhatják ezeket a jelentéseket, így a feldolgozási logika nem a szerveroldalon, hanem a kliensoldalon valósulna meg.

A rendszer három különböző jelentést képes generálni:

- Készlet állapotról szóló jelentés: Az InventoryStateReport biztosítja a raktárak teljes kapacitásának, kihasználtságának, valamint az egyes termékkészletek aktuális állapotának részleteit.
- Lejárati állapotról szóló jelentés: Az InventoryExpiryReport a termékek lejárati állapotáról (például NOTEXPIRED, SOONTOEXPIRE, EXPIRED, LONGEXPIRED) ad pontos információt, kategorizálva és a szállítók szerint csoportosítva.
- Rendelések összesítő jelentése: Az OrderReport részletes statisztikát nyújt a rendelések számáról, állapotáról és a felhasználók rendelési aktivitásáról.

5.6 Platformfüggetlenség

Az alkalmazásom hordozhatóságának biztosítása érdekében a konténerizációs technológiákra, különösen a Dockerre építettem. A rendszer működését Dockerfile és docker-compose.yml fájlokkal valósítottam meg, amelyek a rendszer összes komponensének telepítését és konfigurációját egységesen kezelik.

A Docker Compose segítségével két fő szolgáltatás indítható el: az alkalmazás és a PostgreSQL adatbázis. Az adatbázis szolgáltatás kezeli a perzisztens adatokat, amelyeket egy helyi volume-ban tárol a későbbi elérhetőség biztosítása érdekében. Az alkalmazás a backend szolgáltatással kommunikál, és az adatbázis indítása után automatikusan csatlakozik hozzá. A komponensek egy privát hálózaton belül kommunikálnak, amely biztosítja a rendszer elszigeteltségét és biztonságát.

A Dockerfile két szakaszban építi fel az alkalmazást. Az első szakaszban a forráskód lefordítása és a függőségek kezelése történik, míg a második szakaszban a futtatási környezetet hozom létre egy minimalista OpenJDK képpel. Ez a megközelítés biztosítja, hogy az alkalmazás gyorsan és hatékonyan működjön, miközben a konténer mérete minimális marad.

A környezeti változók kezelésére .env fájlokat használtam, amelyek biztosítják, hogy az érzékeny információk – például az adatbázis hitelesítési adatai és a JWT titkos kulcs – ne kerüljenek közvetlenül a kódba. Ez egyszerűsíti a konfigurációt, és lehetővé teszi az alkalmazás különböző környezetekhez való egyszerű testreszabását.

Ez a megoldás biztosította, hogy az raktárkezelő alkalmazás bármilyen platformon azonos módon működjön, legyen szó helyi fejlesztési környezetről vagy éles szerverről.

6 Tesztelés

A tesztelés kiemelt szerepet játszott a projekt során, mivel a feladatkiírás hangsúlyozta a magas szintű tesztlefedettség elérésének és a CI/CD folyamatok gördülékeny integrációjának fontosságát. Ennek érdekében a tesztelési folyamatot átgondolt stratégiával építettem fel, amely biztosította a kód megbízhatóságát és robusztusságát.

A tesztelési stratégiám az iteratív fejlesztés elvét követte: minden újonnan implementált funkcióhoz azonnal írtam teszteket. Bár a tesztvezérelt fejlesztést (TDD) nem alkalmaztam, törekedtem a hozzá hasonló módszertan követésére. A tesztelési sorrend logikusan épült fel: először az adatbázis kapcsolatok tesztelése történt meg, majd az üzleti logikai réteget izoláltan teszteltem, végül pedig az integrációs tesztek során a teljes komponensek közötti együttműködést validáltam.

6.1 Profilok

A fejlesztés és a tesztelés során külön tesztkörnyezet-specifikus konfigurációs fájlt használtam: az application-test.yaml-t. Ez a fájl lehetővé tette, hogy a tesztelési környezet konfigurációja független legyen az éles vagy fejlesztési környezet beállításaitól. A teszt profil aktiválásakor a Spring automatikusan betöltötte az application-test.yaml-ben található beállításokat, például az in-memory H2 adatbázis használatát. [12]

A tesztelési konfiguráció egyik fő előnye az volt, hogy minimálisra csökkentette az adatvesztés vagy más nem szándékos konfliktusok esélyét. Emellett rugalmasan tudtam a tesztelési paramétereket módosítani. Például a kritikus helyzetek szimulálására a should-check-expiration értékét tesztkörnyezetben hamisra állítottam, amely lehetővé tette specifikus forgatókönyvek könnyebb tesztelését.

6.2 Tesztelési módszerek

6.2.1 Egység tesztek

A unit (egység) tesztek célja az alkalmazás üzleti logikai rétegének egyes komponenseinek izolált tesztelése. Ezek a tesztek garantálják, hogy az egyes osztályok, függvények vagy modulok a kívánt módon működjenek anélkül, hogy más

rendszerelemek működése befolyásolná az eredményt. A tesztelés során olyan helyzeteket szimulálok, amelyek éles környezetben is előfordulhatnak, például hiányzó adatok, érvénytelen bemenetek vagy váratlan helyzetek kezelése.

Az alábbi kódrészlet egy tipikus unit tesztet mutat be, amely a ProductService komponens viselkedését ellenőrzi abban az esetben, ha egy nem létező termékazonosítóra történik lekérdezés.

```
@ExtendWith(MockitoExtension.class)
@Test
void shouldThrowExceptionWhenProductNotFound() {
  Long productId = 1L;
  when(productRepository.findById(productId))
                        .thenReturn(Optional.empty());
  assertThrows(ProductNotFoundException.class,
    () -> productService.getProductById(productId));
}
```

Az @ExtendWith(MockitoExtension.class) annotáció megadja, hogy a teszt során a Mockito keretrendszerrel dolgozunk. Ez lehetőséget nyújt arra, hogy a tesztek során elkülönítetten, a valódi implementációk helyett szimulált (mock-olt) függőségekkel dolgozzunk, például az adatbázis elérési réteg viselkedésének imitálásával.

1. Adat előkészítése: A productId egy bemeneti adat, amely azt a termékazonosítót jelöli, amelyre a teszt során hivatkozunk.
2. (Helyettesített) Mock-olás: A when(productRepository.findById(productId)) .thenReturn(Optional.empty()); sor azt szimulálja, hogy a productRepository nem találja meg az adott azonosítójú terméket. Ezáltal az adatbázis elérési réteg visszatérési értéke egy üres Optional.
3. Viselkedés tesztelése: Az assertThrows metódus ellenőrzi, hogy a ProductService a megfelelő kivételt (ProductNotFoundException) dobja-e, amikor a getProductById metódust egy nem létező termékre hívják meg.
4. Eredmény: A teszt akkor sikeres, ha a metódus ténylegesen a várt ProductNotFoundException kivételt dobja. Ez biztosítja, hogy az üzleti logika megfelelően kezeli a nem létező termékek esetét.

Ez a teszt szemlélteti, hogy a Mockito segítségével milyen egyszerűen előállíthatók olyan szituációk, amelyek ritkábban fordulnak elő, és az applikáció határhelyzeteit feszegetik.

6.2.2 Integrációs tesztek

Az integrációs tesztek olyan tesztelési módszerek, amelyek célja a rendszer különböző komponensei – például az adatbázis, az adatforrás réteg, és a szolgáltatások – közötti együttműködés helyességének ellenőrzése. Ezek a tesztek nem egyedi metódusokat vagy osztályokat vizsgálnak, hanem azt, hogy a rendszer komponensei együtt képesek-e a várt funkciókat megvalósítani egy valósághű környezetet imitálva. Ehhez a tesztelés során egy in-memory adatbázist, az H2-t használtam, amely gyors és könnyen konfigurálható, így ideális az izolált tesztelési környezethez.

```
@SpringBootTest(webEnvironment                                          =
SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWebTestClient
@AutoConfigureTestDatabase(replace                                      =
AutoConfigureTestDatabase.Replace.ANY)
@ActiveProfiles("test")
@Test
void shouldReturnAllCategories() {
  Category category1 = Category.builder()
    .name("Category1")
    .description("Description1")
    .build();
  Category category2 = Category.builder()
    .name("Category2")
    .description("Description2")
    .build();
  categoryRepository.saveAll(List.of(category1, category2));
  getAllCategories()
    .expectStatus().isOk()
    .expectBodyList(Category.class)
    .hasSize(2)
    .value(categories -> {
      assertEquals("Category1", categories.get(0).getName());
      assertEquals("Category2", categories.get(1).getName());
    });
}
WebTestClient.ResponseSpec getAllCategories() {
  return webTestClient.get().uri(CATEGORY_PATH)
    .header(AUTHORIZATION_HEADER, BEARER_PREFIX + adminToken)
    .exchange();
}
```

A @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment. RANDOM_PORT) annotáció egy teljes Spring-környezetet indít a teszteléshez, véletlenszerűen kiválasztott porton. Az @AutoConfigureWebTestClient lehetővé teszi a WebTestClient használatát az API-végpontok tesztelésére. Az @ActiveProfiles("test") aktiválja a test profilt, így a test profilhoz tartozó application-test.yaml konfigurációk automatikusan betöltődnek. Az @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.ANY) biztosítja, hogy minden adatbázis-interakció az in-memory H2 adatbázissal történjen.

A kód működése:

1. Tesztelési adatok létrehozása: A teszt két kategóriát (category1 és category2) hoz létre és ment az in-memory adatbázisba a categoryRepository.saveAll() segítségével.
2. API-végpont tesztelése: A getAllCategories() metódus meghívja az API-t, hogy lekérje az összes kategóriát.
3. Eredmények ellenőrzése: A teszt validálja, hogy a válasz státusza 200 OK, a visszaadott lista mérete 2, és az adatok egyeznek a várt értékekkel ("Category1", "Category2").

6.2.3 Teljes rendszer tesztelés

Az teljes rendszerellenőrzést (End-to-End tesztelést) a projekt során manuálisan hajtottam végre, az OpenAPI Swagger által biztosított felület segítségével. Ezt a megközelítést tudatosan választottam, mivel a projekt időkerete és a rendszer összetettsége nem tette lehetővé automatizált tesztelési megoldások kialakítását. A rendszer minden funkcióját alaposan átvizsgáltam, beleértve a szélsőséges eseteket is, hogy biztosítsam az alkalmazás stabilitását és a hibák minimalizálását.

6.3 CI/CD folyamatok

A CI/CD folyamatok elengedhetetlenek az alkalmazás folyamatos fejlesztésének és megbízhatóságának fenntartásához. A GitHub Actions segítségével egy automatizált folyamat rendszert hoztam létre, amely a kód minden egyes változásakor – legyen az pull request vagy a main branch-re történő push – automatikusan lefuttatja az építési, tesztelési és minőségellenőrzési lépéseket.

```
name: CI/CD Pipeline
on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Build and Analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GIT_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
          JWT_SECRETKEY: ${{ secrets.JWT_SECRETKEY }}
          CUSTOM_ADMIN_USERNAME: ${{ secrets.
                              CUSTOM_ADMIN_USERNAME }}
          SPRING_DATASOURCE_PASSWORD: ${{ secrets.
                              SPRING_DATASOURCE_PASSWORD }}
        run: mvn -B verify org.sonarsource.scanner.
                    maven:sonar-maven-plugin:sonar
```

Automatizált építés és tesztelés

Az automatizált folyamat (pipeline) első lépése a forráskód ellenőrzése és letöltése a GitHub adattárolóból. Ez biztosítja, hogy mindig a legfrissebb verzió kerüljön feldolgozásra. A Maven segítségével az alkalmazást felépítem, és automatikusan lefuttatom a teszteket (unit és integrációs teszteket egyaránt). Ez garantálja, hogy a kód minden része megfeleljen az elvárt működésnek. Az építés során a SonarQube elemzi a kódot, kiemelve a potenciális hibákat, javításra szoruló részeket és a lefedettségi szintet.

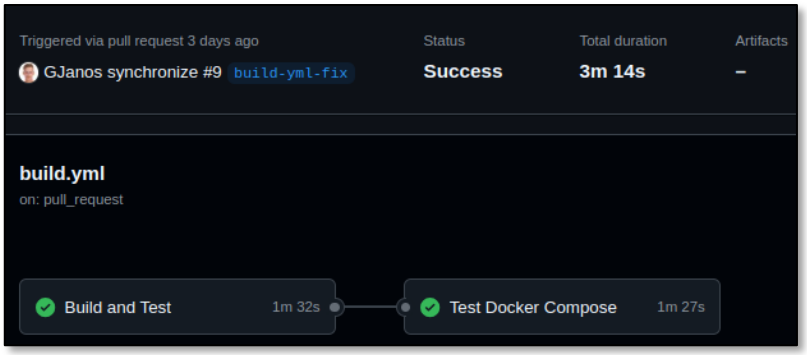
Titkos adatok biztonságos kezelése

Az olyan érzékeny adatokat, mint a JWT kulcs, az admin felhasználói adatok vagy az adatbázis kapcsolati paraméterei, a GitHub Secrets funkcióval tároltam. Ezek a pipeline futtatása során válnak elérhetővé, és biztosítják, hogy az adatok biztonságban maradjanak.

Például a SonarQube token és a környezeti változók mind a titkosított GitHub Secrets-ből kerülnek be a folyamatba, így nem jelennek meg nyilvánosan a pipeline során.

Konténerizált tesztelés Docker Compose segítségével

Az építés és tesztelés után az automatizált folyamat elindítja a konténerizált alkalmazást egy Docker Compose környezetben, amely tartalmazza az alkalmazás fő komponenseit, például az adatbázist és a backendet. A Docker Compose futtatásának egyik előfeltétele, hogy a megfelelő környezeti változók elérhetők legyenek a konténerek számára. Ezeket a környezeti változókat a pipeline során a GitHub Secrets segítségével hoztam létre, és automatikusan generáltam a szükséges .env fájlokat, például az .env.app-ot és az .env.postgres-t. A Docker Compose futtatás során a rendszer validálja, hogy a konténerek sikeresen létrejöttek és megszűntek.



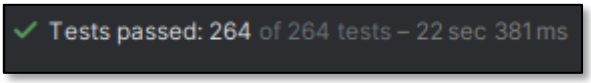
14. ábra: Sikeresen lefutott Github Actions munkák.

6.4 Tesztelés végeredménye

A tesztelés során elért eredmények jól mutatják a projekt stabilitását és megbízhatóságát. A 264 sikeresen lefutott teszteset az alkalmazás erősségét és megbízhatóságát tükrözi, és külön elégedettséggel tölt el, hogy a sorokra vetített lefedettség 95,2%-os. Az összesített lefedettségi érték ugyan 57,4%, azonban ennek oka, hogy számos adatátviteli objektumom (DTO) van, amelyek csupán egyszerű adatlekérő és -beállító függvényeket tartalmaznak, így ezek külön tesztelése nem volt szükséges. Szintén érdemes kiemelni, hogy a több mint 15000 sorból álló projekt kódismétlések aránya 0,0%, amit a SonarQube elemzése is alátámaszt. Ez azt mutatja, hogy az alkalmazásban nem található redundáns vagy többször előforduló kódrészlet, amely különösen fontos a fenntarthatóság és az olvashatóság szempontjából.

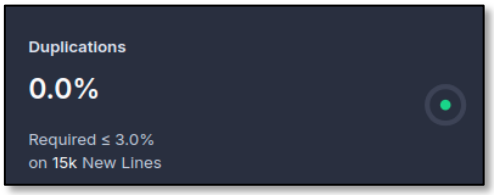
Büszke vagyok az elért eredményekre, hiszen rengeteg időt és energiát fordítottam arra, hogy a tesztek valóban minőségiek legyenek. A tesztek alapja az üzleti

logika stabil működése, amelyet nagy körültekintéssel ellenőriztem. A tesztelés által nyújtott megbízhatóság jelentős mértékben hozzájárult az alkalmazás végső minőségéhez, biztosítva, hogy a rendszer stabil, hibamentes és készen áll a valós környezetben való használatra.



15. ábra: Összes tesztem futási eredménye.

16. ábra: SonarQube által futtatott teszt lefedettségi statisztika.



17. ábra: SonarQube által futtatott kód duplikáció statisztika.

7 Összefoglalás

A dolgozatom során egy modern, több szerepkört támogató raktárkezelő rendszer backendjét terveztem és valósítottam meg, amely számos üzleti logikai funkcióval rendelkezik. A projekt elején még csak alapszintű ismereteim voltak a Spring keretrendszerről, de a munka során hatalmas fejlődést értem el: elmélyítettem tudásomat a Spring Boot, Spring Security, JPA/Hibernate és a CI/CD folyamatok területén. Az ilyen méretű projekt megvalósítása nemcsak szakmai ismereteimet bővítette, hanem önbizalmamat is megerősítette, hiszen egy ekkora volumenű rendszert sikerült a kezdetektől a végéig kiviteleznem.

A fejlesztés során számos technológiai megoldást alkalmaztam, amelyek közül a Spring Boot tette lehetővé a moduláris, könnyen bővíthető architektúra kialakítását. A JPA/Hibernate egyszerűsítette az adatbázis-kezelést, míg a Spring Security és a JWT tokenek a biztonságot garantálták. A H2 in-memory adatbázis gyors és egyszerű tesztelési lehetőségeket kínált, a CI/CD pipeline pedig a folyamat zökkenőmentességét biztosította. A rendszer konténerizációját Docker segítségével valósítottam meg, ami különösen fontos volt a hordozhatóság érdekében. Ahogy haladtam előre a projektben, ráébredtem, hogy a Spring keretrendszer milyen elképesztő rugalmasságot nyújt, és mekkora segítséget jelent a modern alkalmazásfejlesztésben.

A feladat megoldása során számos mérnöki kihívással szembesültem, amelyekkel a való életben is gyakran találkozhat egy fejlesztő. Az új technológiák, fejlesztési és tesztelési módszerek felfedezése és alkalmazása nemcsak szélesítette a látókörömet, hanem jelentős mértékben hozzájárult ahhoz is, hogy magabiztosabb és felkészültebb szakemberré váljak.

8 Irodalomjegyzék

[1] „Baeldung - What Is a Spring Bean?,” [Online]. Available: https://www.baeldung.com/spring-bean. [Hozzáférés dátuma: 12 10 2024].

[2] „Spring - Building REST services with Spring,” [Online]. Available: https://spring.io/guides/tutorials/rest. [Hozzáférés dátuma: 12 10 2024].

[3] „Java – JPA vs Hibernate,” [Online]. Available: https://www.geeksforgeeks.org/java-jpa-vs-hibernate/. [Hozzáférés dátuma: 14 10 2024].

[4] „Introduction to JSON Web Tokens,” [Online]. Available: https://jwt.io/introduction. [Hozzáférés dátuma: 14 10 2024].

[5] „Spring Boot features testing,” [Online]. Available: https://docs.spring.io/spring-boot/docs/1.5.2.RELEASE/reference/html/boot-features-testing.html. [Hozzáférés dátuma: 20 10 2024].

[6] „SonarQube - Java test coverage,” [Online]. Available: https://docs.sonarsource.com/sonarqube-server/latest/analyzing-source-code/test-coverage/java-test-coverage/. [Hozzáférés dátuma: 21 10 2024].

[7] „GitHub Docs - Building and testing Java with Maven,” [Online]. Available: https://docs.github.com/en/actions/use-cases-and-examples/building-and-testing/building-and-testing-java-with-maven.

[8] „IBM - What is Docker?,” [Online]. Available: https://www.ibm.com/topics/docker. [Hozzáférés dátuma: 26 10 2024].

[9] „Postman - Guide to API-first,” [Online]. Available: https://www.postman.com/api-first/. [Hozzáférés dátuma: 4 11 2024].

[10] „GitHub - Spring boot 3 JWT security,” [Online]. Available: https://github.com/ali-bouali/spring-boot-3-jwt-security. [Hozzáférés dátuma: 6 11 2024].

[11] „GeeksForGeeks - Spring Boot scheduling,” [Online]. Available: https://www.geeksforgeeks.org/spring-boot-scheduling/. [Hozzáférés dátuma: 2 12 2024].

[12] „Baeldung - Spring profiles,” [Online]. Available: https://www.baeldung.com/spring-profiles. [Hozzáférés dátuma: 1 12 2024].