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

Kovács Ádám

# RFID AZONOSÍTÁS, ADATGYŰJTÉS IPARI KÖRNYEZETBEN

KONZULENS

Kovács László

BUDAPEST, 2020

# TARTALOM

# HALLGATÓI NYILATKOZAT

Alulírott **Kovács Ádám**, 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(k), 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, 2020. 12. 11.

...…….

Kovács Ádám

# 1 Összefoglaló

A hálózatba kapcsolt eszközök, vagyis a dolgok internete (Internet of Things) alapjaiban változtatja meg az emberek életét, a termelési rendszereket, üzleti modelleket. Az elmúlt években a gyártásban is elterjedtek a hálózatba kapcsolt, egymással kommunikálni tudó, akár döntésképes eszközök: az ipari termelés területén is elindult egy technológiai forradalom. E változásokat hívjuk Ipar 4.0-nak. Az Ipar4 technológiák alkalmazásával lehetőség nyílik akár egyedi termékek előállítására is gazdaságos módon, egy sorozatgyártásra berendezett üzemben – a munkám során egy ennek támogatására alkalmas megoldást készítettem.

A rádiófrekvenciás azonosítás (Radio Frequency IDentification) olyan technológia, amely rádióhullámok segítségével automatikus azonosítást, illetve adatrögzítést tesz lehetővé. Az azonosításhoz un. RFID-taget használunk, ami egy apró eszköz, amely rögzíthető az azonosítani kívánt tárgyon, amivel az RFID író/olvasó segítségével kommunikálni lehet.

A szakdolgozatom során egy olyan rendszer megtervezését és elkészítését tűztem ki célul, amihez az egyetemi éveim alatt összegyűjtött tudásom legjavát és a gyakorlatban előforduló ipari és informatikai technológiák széles palettáját kell felhasználnom. A munkám során elkészített alkalmazás egy ipari környezet által ihletett rádiófrekvenciás azonosítórendszer, amely a termelés közben dinamikus illusztrációk által (egyfajta vizuális összeszerelési útmutatóként) nyújt segítséget egyedi termékek pontos összeszereléséhez.

A (web)alkalmazás a háromrétegű architektúra szoftverfejlesztési minta alapján készült el, rétegenként több komponenssel. A komponensek egymástól függetlenül mikro-szolgáltatásként konténerizált környezetben üzemelnek és HTTP segítségével kommunikálnak. A rendszer egy single-board computer-en fut, és hálózaton keresztül van összeköttetésben az RFID olvasót működtető programozható logikai kontrollerrel (PLC). Az RFID érzékelő a termékek 3D nyomtatott alkatrészein rögzített UHF RFID tag-ekkel képes kommunikálni. A termékek megrendelését tablet, az összeszerelés megjelenítését pedig monitor teszi lehetővé. A kód fordítását modern build keretrendszer könnyíti meg.

# 2 Abstract

The Internet of Things fundamentally change people's lives, production systems, and business models. In recent years certain networked devices, capable of communicating with each other and making decisions became widespread and they are launching one of the technological revolutions in the field of industrial production. These changes are called Industry 4.0.

Radio Frequency Identification (RFID) is a technology that can use radio waves for automatic identification and data recording. An RFID tag is a tiny device that can be attached to the object to be identified and communicate with the RFID reader / writer.

In my thesis, I aimed to design and create such a system that requires using the best of my knowledge gathered during my university year and the wide range of industrial and IT technologies that occur in practice. The application created in the course of my work, is a radio frequency identification system inspired by an industrial environment, which provides dynamic representation (as a kind of visual assembly guide) for the precise assembly of the unique products.

The (web) application is based on the three-tier software architecture pattern, with multiple components per layer. The components operate in a containerized environment as independent microservices and communicate with HTTP. The system runs on a microcontroller and is connected via a network to a programmable logic controller (PLC) that operates the RFID reader. The RFID sensor is able to communicate with the UHF RFID tags attached to the 3D printed parts of the products. The products can be ordered on a tablet and the assembly is displayed on a monitor. Compiling the code is facilitated by a modern build framework.

# 3 Bevezetés

## 3.1 Előzmények

Másodéves hallgatóként lehetőségem nyílt az egyetemen az Ipar 4.0 Technológiai Központban demonstrátorként dolgozni. Munkám során a központban található ipari informatikai berendezések üzemeltetése, karbantartása és bemutatása volt a feladatom. Ezek az eszközök cégek berendezései vagy egyetemi projektek eredményei és egy-egy ipari „szcenárióba" nyújtanak betekintést. Céljuk annak a szemléltetése, hogy olyan technológiák, mint az IoT, 3D nyomtatás, RFID azonosítás, Big Data, milyen előnyökkel járnak gyakorlati alkalmazásuk esetén.

Ebben az időszakban ismerkedtem meg mélyrehatóbban ezekkel a technológiákkal és elhatároztam, hogy én is szeretnék készíteni olyan rendszert, ami a többi Technológiai Központban lévő szcenárióhoz hasonlóan bemutat néhány ipari informatikai megoldást. A kedvenc témáim közé tartozik az RFID azonosítás illetve a 3D nyomtatás, így e technológiák felhasználásával akartam a tervemet megvalósítani, de olyan formában, hogy a fókuszpontban mindenképpen az adatbázist, backend szervert és frontend webalkalmazást tartalmazó szoftver álljon.

## 3.2 Indokoltság

Az iparban jelenleg egyforma termékek sorozatgyártása hatékonyan megoldható modern gyártósorok segítségével, azonban egyedi termékek tömeges előállítása költséghatékony módon továbbra is kihívást jelent, mert az gyártás költsége az egyedi konfigurációk számával arányosan növekszik. Az ipar 4 egyik célja, hogy ipari informatikai technológiák felhasználásával megoldást nyújtson erre a problémára.

„A vizsgált vállalat tapasztalatai megerősítik, hogy az I4.0 csúcstechnológiái összefonódnak: a big data elemzés megfelelő használata feltételezi a dolgok internetét (szenzorokkal felszerelt gépek hálózatát), az adattárolási és számítási kapacitást (felhő), valamint a szoftvermegoldások és a megfelelő tudás egyidejű rendelkezésre állását. Az I4.0 fizikai szférája, mint az okos gépek, RFID rendszerek stb., csak ezekbe az összefonódó digitális hálózatba ágyazhatók be." [1]

Az egyedi gyártás első lépése az adatgyűjtés (jelen esetlen a termékek egyedi azonosítása), a második lépés az adott terméken a szükséges beavatkozás informatikai rendszer által való meghatározása. A harmadik lépés a művelet automatikus elvégzése.

A szakdolgozat projekt keretein belül elkészített rendszer a fenti elvek alapján működik. Az azonosítást egy RFID azonosító egység végzi el, a következő szerelési lépést a rendszer határozza meg, azonban az automatizált szerelés megoldása egy bemutató rendszerrel nem megvalósítható, így a rendszer a következő lépést egy vizuális felületen megjeleníti, a szerelési feladat pedig az „operátorra" hárul.

## 3.3 A demo rendszer

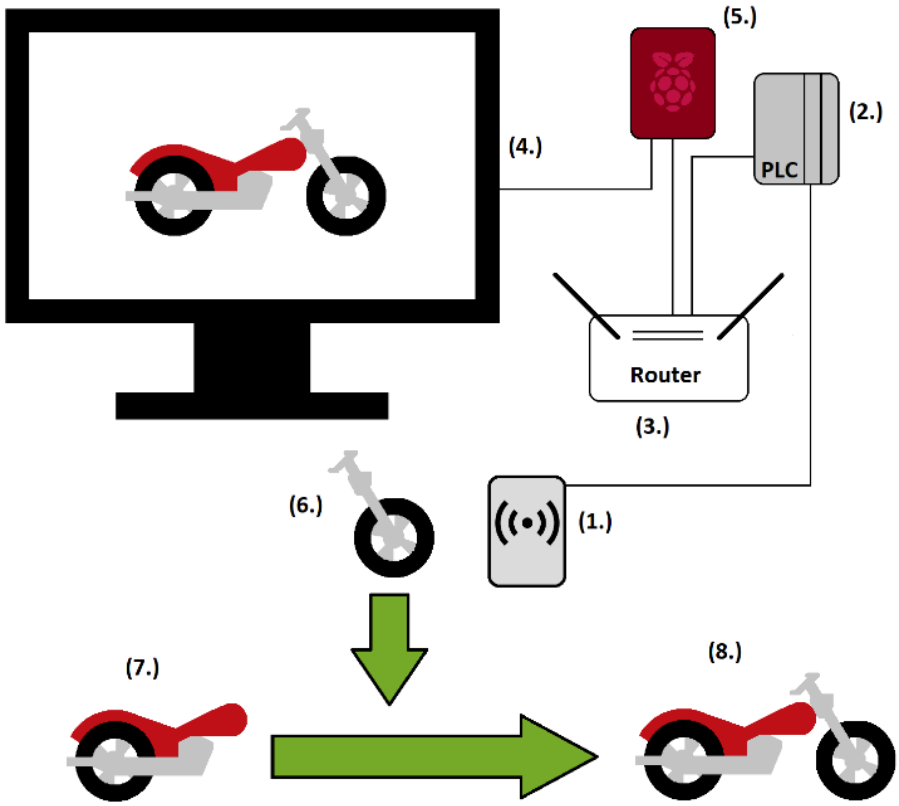
Az ötlet az előző félévben megszületett, és Önálló laboratórium tárgy keretein belül elkezdtem megvalósítani a rendszert, amiből egy „Proof of concept" alkalmazás el is készült a félév végére, azonban felismerve a demó első verziójához választott technológiák korlátait, úgy döntöttem, hogy a szakdolgozat projekt során nem az előzőfélévben elkezdett rendszert folytatom, hanem „tiszta lappal" állok neki az új rendszer megvalósításának. (Az adatmodellen és az RFID azonosításhoz szükséges komponensek egy részén kívül minden mást újra terveztem.)

A projekt során egy interaktív demót akartam elkészíteni, ami egy olyan szerelőállomást mutat be, ahol a gyártott termék egy 3D nyomtatott műanyag alkatrészekből álló motorkerékpár, amelynek egyes alkatrészeit más-más színben, ízlés szerint lehet kiválasztani.

Az állomáshoz tartozik egy RFID olvasó, ami munkadarab alkatrészein található RFID tag-eket vizsgálja, és ezek alapján azonosítja a termékeket. Az operátorral szemben található egy monitor, amin a munkadarabhoz tartozó utasítás látszik, továbbá az alkatrészek is az operátor előtt lévő asztalon találhatók.

A képernyőn az éppen szerelés alatt álló munkadarab aktuális állapota látható, kiegészítve a villogó új alkatrésszel (ezzel jelezve, hogy ez a következő alkatrész, amit fel kell szerelni a termékre). Miután a munkadarabra rá lett illesztve az új elem, az RFID olvasóhoz tartva a terméket, az olvasó azonosítja az összes RFID tag-et és a rendszer ellenőrzi, hogy megfelelően történt-e a szerelés (minden RFID tag megtalálható-e a terméken, ami szükséges). Ha nem, akkor hibaüzenetben figyelmeztet, ha igen, akkor megerősíti a szerelés sikerességét és a képernyőn a következő lépés lesz látható, ahol a

következő alkatrész villog. Amennyiben elkészült a termék, a rendszer azt is jelzi és megjeleníti a soron következő termék első alkatrészét (villogva).




**A rendszer működése és komponensei**

1. RFID olvasó
2. PLC
3. Router
4. Monitor
5. Mikroszámítógép
6. Következő alkatrész
7. Aktuális munkadarab
8. Kész munkadarab

A fentiekben bemutatott termelést segítő szolgáltatás mellett a rendszer tartalmaz egy megrendelői szolgáltatást is. A termékek egyedinek mondhatók abban a tekintetben, hogy a legtöbb alkatrész több színben is rendelkezésre áll. A megrendelői felület célja, hogy lehetőséget teremtsen a megrendelt termék alkatrészei színének kiválasztására.

# 4 Specifikáció

A feladat egy olyan demo rendszer megalkotása, amely az RFID technológiát használva segíti egy munkaállomáson történő műanyag 3D nyomtatott választható színű alkatrészekből álló motorkerékpár makett összeszerelését. A demo rendszert úgy kell megvalósítani, hogy végig kövesse a termék életciklusát a megrendeléstől egészen az összeszerelés végéig.

A feladat része egy megrendelési felület létrehozása, ami megjeleníti a motorkerékpárt és a vizuális felületen lehetőség van az alkatrészek színének kiválasztására. A kiválasztott rendelést lehessen elküldeni, amit tároljon el a rendszer.

Létre kell hozni 3D nyomtató segítségével a motorkerékpár alkatrészeit több színben, az alkatrészeken RFID chipeket kell rögzíteni és az alkatrészek tulajdonságait (RFID chip azonosítója, szín, típus…) el kell tárolni a rendszerben.

El kell készíteni a PLC vezérelt RFID azonosító rendszert, ami elé az RFID tag-ekkel ellátott alkatrészeket tartva érzékeli a chipek azonosítóját és ezt elküldi a szervernek.

El kell készíteni egy vizuális felületet, ami a szerelés közben mutatja, hogy milyen (félkész) termék alkatrészeit azonosította be a rendszer és mi a következő alkatrész, amit rá kell szerelni a munkadarabra.

A rendszernek rendelkeznie kell backend szolgáltatással, ami a rendelést ellenőrzi, például hogy az adott termék típus minden komponenséhez olyan színt választottunk-e, amit az adatbázisban szereplő adatok alapján lehetséges (ne fogadja el a lila hátsó kereket tartalmazó megrendelést, ha az adatbázis szerint csak fekete hátsó kerékkel lehetséges megrendelni a terméket).

A rendszernek rendelkeznie kell backend szolgáltatással, ami ellenőrzi, hogy az azonosított RFID chipekhez tartozó komponensek konzisztensek az adatbázisban lévő komponensekkel (ha az adatbázis szerint a munkadarabra eddig egy szürke váz és egy fekete hátsó kerék van szerelve, akkor a beazonosított chipekhez tartozó alkatrészeknek is egy fekete hátsókeréknek és szürke váznak kell lenniük).

A rendszer egy Raspberry Pi mini számítógépen működjön.

# 5 Előzmények

Mérnökinformatikus hallgatóként a rendszer szoftveres részének megtervezéséhez illetve megvalósításához nem volt szükségem a különösebb kutatást végeznem vagy egyéb ismeretekre szert tennem. Szakmai ismereteim és a felhasznált technológiák dokumentációi alapján el tudtam készíteni az alkalmazás szoftveres részeit.

A rendszer nem „szakmába vágó” részei számomra a 3D nyomtatás, az RFID technológia illetve a PLC programozás voltak, azonban ezek sem voltak teljesen ismeretlenek, mert az Technológiai Központban töltött időm alatt egy alapszintű tudásra már szert tettem az említett technológiákkal kapcsolatban.

## 5.1 RFID technológia

A rádiófrekvenciás azonosítás (Radio Frequency IDentification) olyan technológia, amely rádióhullámok segítségével automatikus azonosítást, illetve adatrögzítést tesz lehetővé, mikrochipek segítségével. Az RFID-chip/tag egy apró tárgy, amely rögzíthető vagy beépíthető az azonosítani kívánt objektumba, ami tulajdonképpen a már jól ismert vonalkód funkcióját tölti be, viszont annál sokkal többre is képes.

Egy RFID technológiát alkalmazó rendszer három fő komponensét különböztethetjük meg. Szükség van egy RFID olvasó egységre (interrogator), RFID tagekre és egy megfelelő informatikai infrastruktúrára. Az RFID olvasó és a tag közötti kommunikáció az eszközök típusa, a kommunikációs protokoll, a további írás/olvasási műveletek és több RFID tag egyszerre történő olvasása miatt igen sokféle lehet, azonban nagyvonalakban a következőképpen történik:

Amint az RFID tag, belép az RFID olvasó által létrehozott rádiófrekvenciás mezőbe, az RFID tag aktiválódik és visszaküldi (szintén rádiófrekvenciás hullámot használva) az azonosítóját, illetve esetleg a memóriájában tárolt adatokat. Ezeket a rádiófrekvenciás hullámokat az RFID olvasó fogadja, majd továbbítja a vele összeköttetésben lévő informatikai rendszerbe, ami meghatározza a szükséges lépéseket, majd visszaküldi az RFID olvasónak az esetleges további módosítási/írási parancsokat. Az olvasó ezeket az utasításokat végrehajtja, majd véget ér a kommunikáció.

A rádiófrekvenciás azonosítást többféleképpen lehet csoportosítani, a chip energiaellátása alapján lehet passzív, fél-passzív vagy aktív. A működési frekvencia lehet

alacsony, magas vagy ultra magas, a tag-ek a memóriakezelés alapján lehetnek csak olvashatók, vagy írhatók is. Több fajta RIFD ISO szabvány létezik, amelyeket különböző célra fejlesztettek ki, néhányat megemlítve:

- ISO 14443 – HF (13,56 MHz) tartományban működő kis hatósugarú (max. 10cm) és általában személyazonosításra, fizetésre, beléptetés használják.
- ISO/IEC 15693 – HF (13,56 MHz) tartományban működő, közepes hatótávolságú (1.5m) és az ütközésgátló algoritmus miatt képes több tag-et is egyszerre olvasni.

ISO 18000-6 – UHF (860 – 960 MHz) tartományban működő, nagy hatótávolságú (max. 10m) és szintén képes egyszerre több tag-et is olvasni.

# 6 Felhasznált technológiák

Ebben a fejezetben ismertetem, hogy a rendszer elkészítéséhez milyen eszközöket, technológiákat használtam. Nem csak a szoftvereket mutatom be, hanem az RFID olvasót, PLC-t és a 3D nyomtatott komponenseket is. Nem célom a technológiák részletes bemutatása, a hangsúlyt inkább a technológia használatának indoklására helyezem.

A háromrétegű architektúra kialakításához szükséges egy adatbázis, egy backend és egy frontend szerver, valamint utóbbi kettőhöz nyilvánvalóan valamilyen keretrendszer alkalmazása is célszerű, hogy megkönnyítse a fejlesztést. A rendszernek egy fizikai szerveren kell futnia, amire egy mikroszámítógép tökéletesen alkalmas lehet. A rendszer több kisebb alkalmazásból áll, így ezeknek az összekötését, egyszerű futtatását is érdemes valamilyen technológiával megoldani.

## 6.1 Perzisztencia

A rendszernek szüksége van egy adatbázisra, hogy a termékekről perzisztensen tárolja az információt. Az adatbázis alkalmazása azért is indokolt, mert több különböző szerver közösen használja az adatbázisban lévő adatokat így az adattárolás mellett az adatok megosztása is feladata.

### 6.1.1 PostgreSQL

A *PostgreSQL* [2] egy ingyenesen használható, nyílt forráskódú relációsadatbázis-kezelő rendszer. Azért választottam ezt az adatbáziskezelőt a projekt elkészítéséhez, mert az egyik legnépszerűbb ilyen alkalmazás, korábbi munkám során már megismerkedtem vele.

## 6.2 Backend

Amennyiben üzleti logikára nem lenne szükség és egyszerű adattárolás lenne a backend feladata, akkor egyes ingyenes backend-as-a-service szolgáltatások segítségével (mint például a firebase) össze lehetne vonni az adatbázist és a backend szervert. Az elképzelt rendszer backendjének azonban az adatok elmentésén kívül komplexebb üzleti logikát is meg kell valósítania: szükséges validálnia a beérkező adatokat és értesítéseket küldenie. Ebből kifolyólag nem hagyható ki ez a réteg sem a rendszer felépítéséből.

### 6.2.1 Java

Szerveroldali alkalmazáskészítéshez két nyelvvel illetve keretrendszerrel ismerkedtem meg mélyrehatóbban, a *.NET* [3] alapú *ASP-vel* [4] és a *Java* [5] alapú Springgel. Az Önálló laboratórium során készített első verzió ASP.NET-tel készítettem el, azonban az új verzióban a kettő közül az utóbbit választottam, mert az elmúlt hónapokban a szakmai gyakorlatom végzése során ezt a technológiát használtam és ebben szerettem volna tovább fejlődni.

### 6.2.2 Spring

Java alapú backend fejlesztéshez két ismertebb keretrendszer is létezik, a Java EE (Enterprise Edition) és a *Spring* [6]. A Java EE-vel nincs tapasztalatom, a Springet viszont mint fentebb említettem, korábban már használtam, így nem volt kérdéses a Spring melletti döntésem. A Springnek egy kiegészítése a Spring Boot, ami nem csak a backend alkalmazáshoz használatos komponenseket kínál, hanem ezekből egy alap konfigurációt készít, hogy már az első fordításkor egy működő alkalmazást kapjunk.

## 6.3 Frontend

A frontendet az Önálló laboratórium során *Universal Windows Platform* [7] segítségével készítettem el, azonban ez egy rendkívül rossz választásnak bizonyult a megvalósítani kívánt rendszer szempontjából ugyanis az UWP-t nem erre tervezték. Az UWP remek keretrendszert biztosít gombok, szövegdobozok megjelenítéséhez, azonban a projekt során az alkatrészek képeinek megfelelő színben történő megjelenítésén, villogtatásán volt a hangsúly és az ilyen grafikus eseményeket bonyolult volt vele elkészíteni.

Mivel az UWP nem tűnt jó megoldásnak, ezért megvizsgáltam, hogy milyen más technológiát érdemes használni. Az UWP-vel szemben egy webes alkalmazásnál a DOM JavaScripttel való manipulálásával az ilyen animációk könnyedén kivitelezhetők. A rendszer második verziójának elkészítésekor a webes technológiák használata tűnt célravezetőnek.

### 6.3.1 TypeScript

A dinamikusan típusos *JavaScriptnél* [8] nagyobb projektek esetén jobb választás a Microsoft által fejlesztett *TypeScriptet* [9]. A JavaScript nyelv a TypeScript nyelv

részhalmaza (tehát ami helyes JavaScriptben az helyes TypeScriptben is) és lehetőséget ad statikus típusellenőrzésre és nagyban megkönnyíti az objektumorientált szemléletmódú fejlesztést. A TypeScript alkalmazása mellett szólt számomra az is, hogy a szakmai gyakorlatom során is TypeScriptet használtam webalkalmazás fejlesztéséhez.

### 6.3.2 React

Manapság ritkán készítenek webalkalmazást bármilyen keretrendszer vagy segédkönyvtár alkalmazása nélkül. Miután elhatároztam, hogy webes frontendet készítek a projekt során, választanom kellett, hogy milyen keretrendszert használjak. Szakmai gyakorlatom során a *React* [10] és az *Angular* [11] technológiákkal ismerkedtem meg, előbbit a Facebook, míg utóbbit a Google fejleszti, mindkettő támogatja a TypeScriptet.

Az Angular (a hivatalos weboldala szerint is) egy keretrendszer, ezért (részben személyes tapasztalataim szerint) viszonylag kötött a felépítése és a konfigurációk miatt eleinte némi többletmunkával jár az alkalmazás elkészítése, ami hosszú távon kifizetődhet, de a projekthez szükséges webes felületek csupán néhány funkcióval bírnak, ezért az Angular által biztosított lehetőségeket nem tudtam volna kihasználni.

Ezzel szemben a React (a hivatalos weboldala szerint is) egy könyvtár, ami csupán a DOM manipulálásáért felel. A React-et a jó skálázódása miatt választottam. A React projektben .tsx kiterjesztésű fájlokban TypeScript (.jsx fájlban JavaScript) és HTML-szerű kód keverékét írhatjuk. A könyvtár fejlesztői a felelősségek szétválasztását komponensek mentén és nem nézet és logika szeparációja mentén képzelték el, így a tsx fájlba imperatívan leírható a komponens logikája, míg a felépítését deklaratívan adhatjuk meg.

## 6.4 Virtualizáció/Konténerizáció

A *Docker* [12] régóta érdekelt, de mélyebben még nem foglalkoztam vele és úgy gondoltam, hogy a szakdolgozat projektem egy remek lehetőség a technológia kipróbálására. Az Ipar4-ben az egyszerű kezelhetőség és a skálázhatóság miatt indokolt is a microservice alapú Ebből kifolyólag a rendszert eleve úgy akartam elkészíteni, hogy több kisebb komponensből épüljön fel és így lehetőség nyílik az alkalmazások külön-külön konténerben való futtatására. Ebben az esetben tulajdonképpen a technológia megismerése volt a célom és úgy alakítottam a projektet, hogy legyen is értelme az alkalmazásának.

### 6.4.1 Docker, docker-compose

A Docker egy olyan szoftver, amely operációsrendszer szintű virtualizáció segítségével teszi lehetővé programok önálló környezetben való futtatását. A használata a következő: Az alkalmazásokat, konfigurációjukat és függőségeiket becsomagolhatjuk egy-egy úgynevezett Docker Image-be, melyek később a Docker konténerek segítségével egymástól elkülönülve futtathatók. A docker-compose egy kiegészítő eszköz több konténerből álló rendszer futtatásához. A docker-compose.yml konfigurációs fájl tartalmazza a futtatni kívánt konténerek Image-eit, a konténerek nyitott portjait, hálózatokat, és az egymás közötti függőségeket. Ennek segítségével az egész rendszer egy parancs kiadásával felállítható.

## 6.5 Build

A szakmai gyakorlatom során *Gradle-t* [13] használtam a projekt buildeléséhez és konfigurálásához. Tapasztalatom szerint minél nagyobb és összetettebb a projekt, annál fontosabb egy build keretrendszer használata, mert segítségével a fejlesztés felgyorsítható.

### 6.5.1 Gradle

A Gradle-t azért választottam a rendelkezésre álló build eszközök közül, mert a függőségek és konfigurációk meghatározása mellett lehetőség van taskok definiálására is, illetve vannak beépített vagy előre elkészített plugin-ok. Ezeknek a magunk vagy mások által definiált taskoknak a célja, hogy automatizáljuk a fejlesztés egyes lépéseit. Például definiálható olyan task, ami törli a projekthez tartozó adatbázis aktuális tartalmát és utána feltölti a kezdeti értékekkel. Én ennek a képességének különösen a docker image-ek buildelésének automatizálásánál vettem hasznát.

## 6.6 Single-board compunter

A Technológiai Központban több demóhoz is használtunk Raspberry Pi mini számítógépet, mert olcsóbb, mint PC, kis helyen is elfér és egy-egy szoftver futtatásához bőven elegendő erőforrással rendelkezik. Ezen előnyöket szerettem volna ennél a projektnél is érvényesíteni.

### 6.6.1 Raspberry Pi 4 Model B

A *Raspberry Pi* [14] egyik legújabb modelljét kölcsön kaptam az Technológiai Központtól a rendszer megvalósításához. A mini számítógép rendelkezik 4x1.5GHz-es processzorral, 4GB memóriával és egy 32GB-os MicroSD kártyával. Az eszközön Raspberry Pi OS (régebben Raspbian) fut. Feladata az alkalmazások futtatása mellett a termelési nézet monitoron való megjelenítése is. Szerencsére a Raspberry Pi 4 (az elődjeihez képest) már stabilan képes vizuális felületek megjelenítésére.

## 6.7 Programozható logikai vezérlő

A Programozható logikai vezérlő (Programmable Logic Controller) egy ipari beágyazott számítógép. Működése alapjaiban tér el az általános felhasználású számítógépektől. A PLC-kkel szemben elvárt fő követelmények – az ipari felhasználás miatt – a megbízhatóság és a determinisztikus működés. A feladat szerinti RFID azonosítást végző komponenst csak ilyen PLC segítségével lehet megvalósítani, így habár nem kapcsolódik szorosan a PLC programozás az informatikus képzés fő témáihoz, ezt is meg kellett tanulnom.

### 6.7.1 Turck

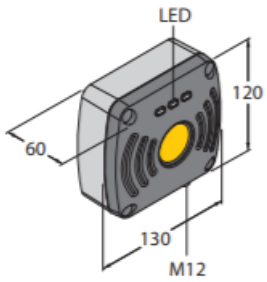
Az alrendszer elkészítéséhez szükségem volt egy PLC-re és egy RFID olvasó egységre. A konzulensem segítségével találtam egy ipari automatizálási berendezéseket gyártó és forgalmazó céget, a Turck Hungary Kft-t, akik biztosítottak számomra az említett berendezések mellett support-ot is, ha elakadtam a program elkészítése során.

#### 6.7.1.1 TBEN-L5-4RFID-8DXP-CDS

A *TBEN-L5-4RFID-8DXP-CDS* [15] egy Turck által gyártott CODESYS 3-ban programozható multiprotocol RFID, Ethernet és egyéb I/O kimenetekkel rendelkező IP67 védettséggel ellátott modul. A projekt szükségleteit bőven kimeríti, sőt, sokkal többre is képes, mert a rendszer elkészítése során csak az Ethernet és a 4 közül az egyik RFID kimenetére van szükség.

#### 6.7.1.2 TN865-Q120L130-H1147

A *TN865-Q120L130-H1147* [16] egy ISO 18000-6C szabványnak megfelelő IP67 védettséggel ellátott RFID UHF író/olvasó fej, ami az energiaellátását és a kommunikációs interfészét is egy M12-es csatlakozó segítségével valósítja meg. A projekt szempontjából fontos, hogy képes egyszerre több RFID tag olvasására is (multitag mode).




**RFID olvasó**

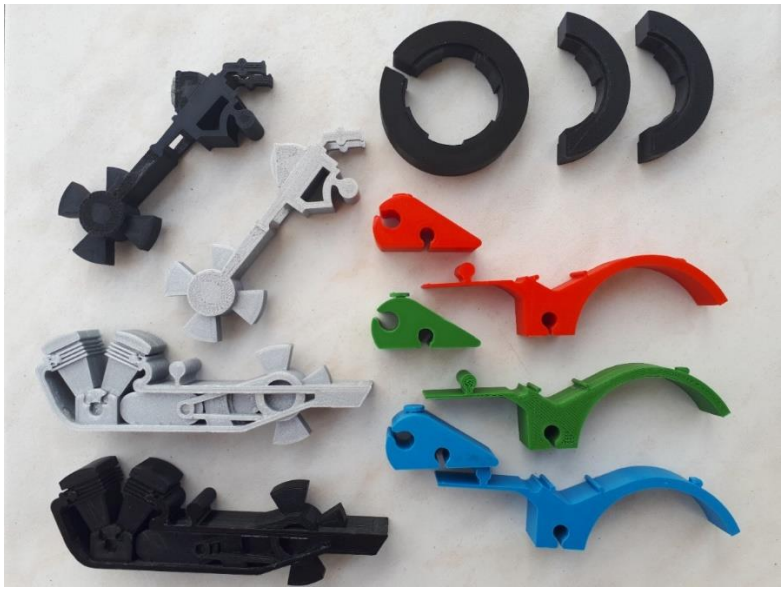
#### 6.7.1.3 RFID-Tag - Impinj Monza B42

A chip kiválasztásánál arra kellett ügyelni, hogy az *ISO 18000-6* [17] szabvány szerint működjön és elférjen a 3D nyomtatott munkadarabokon. A választott chip az Impinj Monza B42 UHF RFID tag lett, aminek a hatótávolsága 20 cm, mérete 26x12 mm, és 860-960MHz-n működnek. Azért erre a típusú tagre esett a választás, mert megfelelő távolságból működik és támogatja egyszerre több tag olvasását. Mivel minden egyes alkatrészre rögzíteni kell egyet, ezért 50 db-ot szereztem be belőle. Kezdetben mindegyik chipnek ugyan az volt az azonosítója. Ezt az RFID (író) olvasó segítségével módosítani kellett, hogy ténylegesen egyedileg lehessen azonosítani mindegyiket.

## 6.8 3D nyomtatás

A projekt lényege, hogy valamit össze kelljen benne szerelni, amivel RFID chipekkel azonosítani lehet az alkatrészeit. A 3D nyomtatás egy olyan technológia, amivel eddigi munkám során is szívesen foglalkoztam, így kézenfekvő volt, hogy a munkadarab alkatrészeit is 3D nyomtassam.

Nem volt egyszerű olyan 3D modellt találni, ami könnyen összeszerelhető alkatrészekből áll, de találtam egy motorkerékpár modellt, ami megfelelt az elvárásoknak. Mivel az alkatrészeknek egymásba kell csúsznia, ezért a nyomtatás során már a néhány tized milliméteres pontatlanság is problémát okoz. A Technológiai Központban volt lehetőségem kinyomtatni az alkatrészeket. A megfelelő nyomtatási konfiguráció megtalálása alkatrészenként is csak sokadik próbanyomtatásra sikerült.



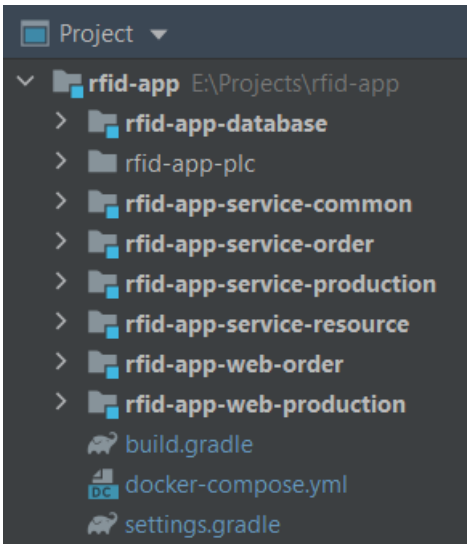
**3D nyomtatott komponensek**

# 7 A rendszer megvalósítása

Ebben a részben bemutatom a rendszer felépítését és működését. Kitérek arra, hogy milyen nehézségekkel kellett megküzdenem az alkalmazás elkészítése során és hogyan oldottam meg ezeket a problémákat. Tartalmilag úgy építettem fel a fejezetet, hogy az alkalmazás komponenseket külön-külön mutassa be, de vannak átfedések (például mivel Docker konténerben fut az összes komponens, ezért a Docker mindegyikhez kapcsolódik) ezért előfordul, hogy az összefüggések miatt olyan komponenst is megemlítek, amiről még nem esett szó.

## 7.1 A Projekt felépítése

A projektet IntelliJ IDEA fejlesztői környezetben készítettem el, rfid-app néven. A projekt több alprojektet tartalmaz, amik majdnem megegyeznek az alkalmazás komponensek felépítésével. Az egyetlen eltérés, hogy a projekt tartalmaz egy extra közös könyvtárat a backend alkalmazásokhoz rfid-app-service-common néven.



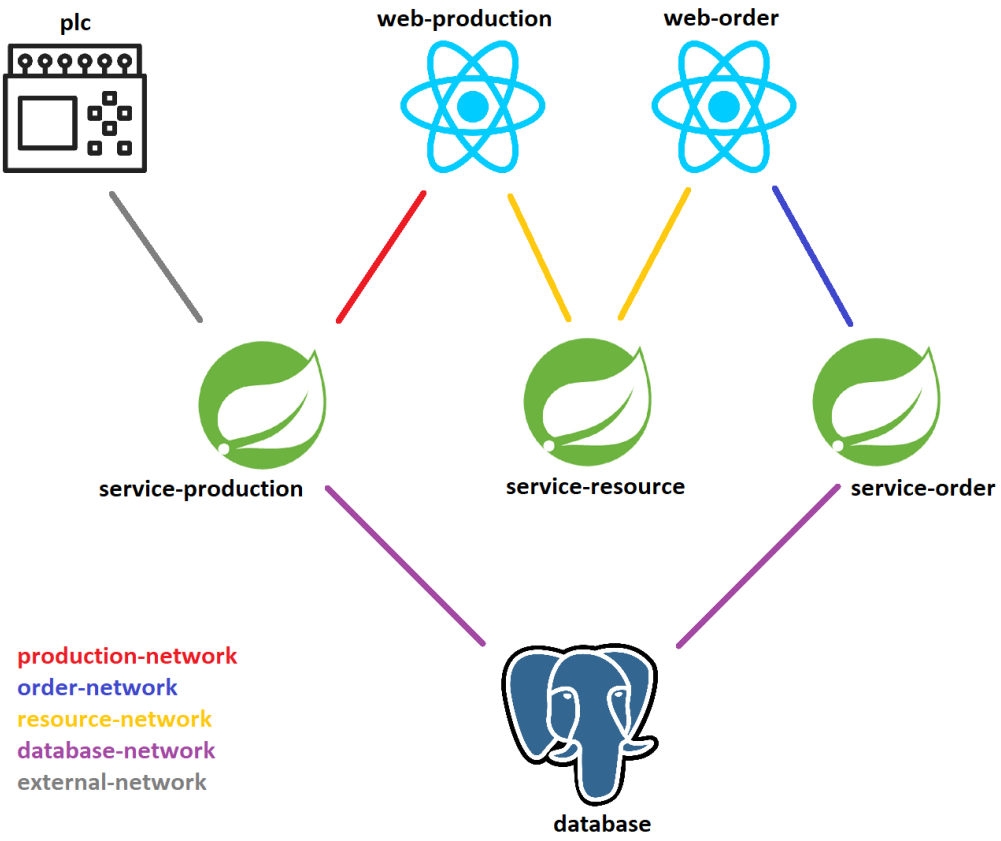

**A projekt felépítése**

Azért volt indokolt ennek a könyvtárnak a létrehozása, mert a backend szolgáltatások adatelérési rétege megegyezik, ezért az ORM model és repository (és néhány egyéb) osztályok ebben a subprojektben találhatók. A többi backend alprojektnek pedig dependenciája van a közös könyvtárra.

Az rfid-app-plc alprojekt az egyetlen, amit nem IntelliJ-ből szerkesztettem, hanem a PLC programozáshoz használt CODESYS-szel.

## 7.2 A Rendszer felépítése

A rendszer 7 elkülöníthető részből áll, egy PostgreSQL adatbázisból, három Spring alapú backend service-ből, két React alapú frontendből és egy PLC programból.




**Alkalmazások és hálózatok**

Az utóbbi értelemszerűen egy PLC-n fut, az előbbiek pedig a Raspberry Pi mini számítógépen Docker konténerekben. Az ábrán látszódnak a komponensek közötti hálózati kapcsolatok. A különböző színek különböző docker alhálózatot jelölnek.

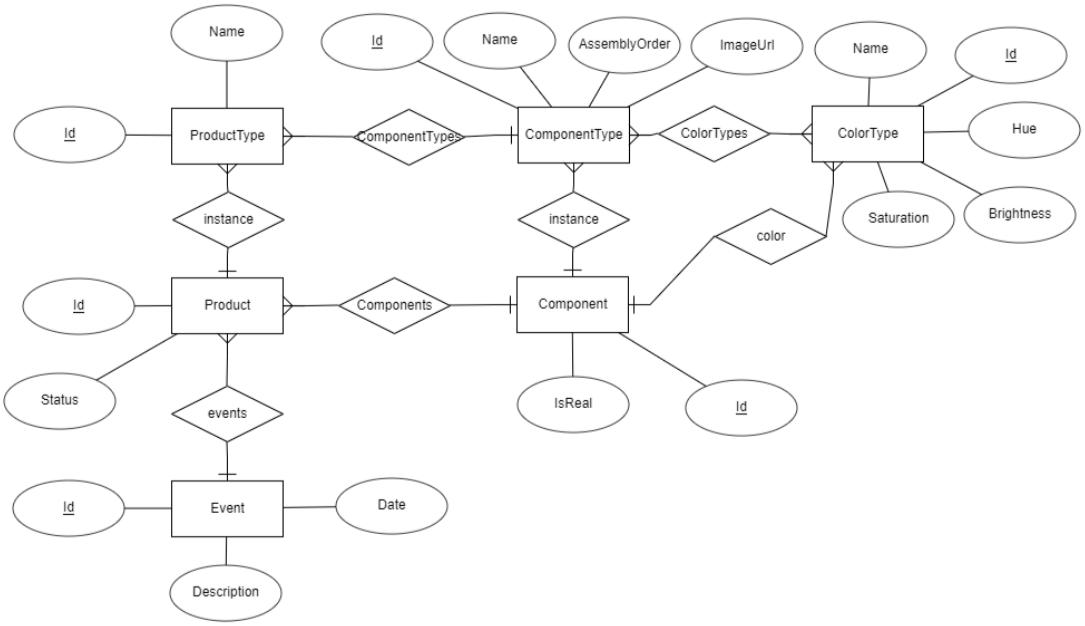
## 7.3 Perzisztencia

Az adattárolást egy Docker konténerben futó PostgreSQL adatbázis segítségével valósítottam meg.

### 7.3.1 Adatmodell

Az adatbázis két absztrakciós szinten tartalmaz adatokat. Egyrészt tárol meta adatokat, amik leírják, hogy egyáltalán milyen termékeket készíthetünk, valamit, konkrét adatokat a gyártásra váró illetve a gyártásban lévő termékekről. Az alábbi ábrán nem

szerepel, mert nem az adatmodell része, de az adatbázis tartalmazza a webes értesítés feliratkozások adatait is. (Erről később beszámolok a frontend kapcsán.)




**Az adatok ER diagramja**

Az adat modell meghatározza a termék típust (**ProductType**), amiknek van neve és azonosítója. A ProductType tartalmazhat több alkatrész típust (**ComponentType**) aminek van neve, azonosítója és meghatározza, hogy hányadik az alkatrész az összeszerelési sorrendben valamint a komponenshez tartozó ábra elérési útvonalát is tartalmazza (ez később, frontendnél lesz fontos).

A ComponentType akár többféle színben (**ColorType**) is létezhet. A ColorType-nak van neve, illetve HSL értékeket is tárolunk az egyes színekről, ugyanis nem tartozik minden színű komponenshez külön kép, hanem a komponenshez tartozó egy darab képet transzformálja a frontend a megfelelő színre. (Erről is bővebben számolok be a frontendhez tartozó fejezetekben.) Konkrét termékeket ezeknek a meta adatoknak a figyelembevételével lehet összeállítani.

A termék (**Product**) egy termék típusból készülhet, azaz tartozik hozzá egy ProductType, amin keresztül van neve és meghatározza, hogy milyen komponensei lehetnek. Ennek a terméknek a komponensei **(Component**) a termék típusának komponens típusai közül kerülnek ki. A rendszer ellenőrzi, hogy ez mindig teljesüljön.

A konkrét komponens két féle lehet: valós komponens, aminek az azonosítója egy fizikailag létező alkatrész RFID azonosítója, vagy „virtuális” komponens, (azonosítója generált). A virtuális mindenképpen egy termékhez tartozik és azt az információt hordozza, hogy milyen valós komponenst kell a termékre illeszteni. Az összes megrendelt termék először virtuális komponensekkel jön létre és ezek az összeszerelés folyamán lecserélődnek valós komponensekre. A konkrét komponens színét a komponenshez tartozó ComponentType színhalmazából lehet kiválasztani.

### 7.3.2 Implementáció

Az adatbázis alprojekt a Dockerfile és a build.gradle fájlon kívül tartalmaz egy sql mappát, amiben két sql script található, a 01_create_tables.sql értelemszerűen az adatbázis tábláinak elkészítéséhez szükséges scriptet, a 02_create_test_data.sql pedig az adatbázis kezdeti adatokkal való feltöltéséhez szükséges scriptet tartalmazza.

```
CREATE TABLE product_type (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL
);

CREATE TABLE component_type (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    assembly_order INTEGER NOT NULL,
    image_url VARCHAR(200) NOT NULL,
    product_type_id INTEGER NOT NULL REFERENCES product_type
);

CREATE TABLE color_type (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    hue INTEGER NOT NULL,
    saturation INTEGER NOT NULL,
    brightness INTEGER NOT NULL
);
```

**Részlet a 01_create_tables.sql scriptből**

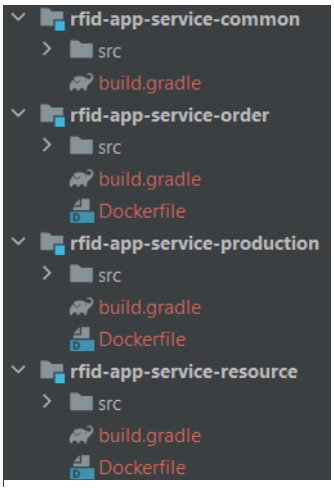
Az előbbi kódrészlet bemutatja, hogy hogyan épülnek fel a ProductType, a ComponentType és a ColorType meta adatokat tartalmazó táblák – amiben semmi szokatlan megoldás nincs – viszont szükséges az alábbi kódrészlet megértéséhez, ami a kezdeti típus meta adatokat tartalmazza (A ColorType és a ComponentType táblák közötti many-to-many join tábla szemléltetése nélkül). Ezek az adatok a program futása során nem változnak, viszont új meta adatok felvételével a „gyártható termékek” köre kibővíthető, tehát nem csak motorkerékpár gyártását képes a rendszer támogatni, hanem bármi hasonlót.

```
INSERT INTO product_type VALUES
(1,'Motor Bike');

INSERT INTO component_type VALUES
(1,'Exhaust Pipe',1,'images/exhaust-pipe.png',1),
(2,'Tyre Back Upper',2,'images/tyre-back-upper.png',1),
(3,'Tyre Back Lower',3,'images/tyre-back-lower.png',1),
(4,'Chassis',4,'images/chassis.png',1),
(5,'Fuel Tank',5,'images/fuel-tank.png',1),
(6,'Saddle',6,'images/saddle.png',1),
(7,'Steering Stem',7,'images/steering-stem.png',1),
(8,'Tyre Front',8,'images/tyre-front.png',1);

INSERT INTO color_type VALUES
(1,'gray',0,0,250),
(2,'black',0,100,0),
(3,'red',0,100,130),
(4,'green',120,100,130),
(5,'blue',240,100,130);
```

**Részlet a 02_create_test_data.sql scriptből**

## 7.4 Backend

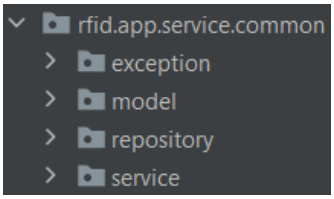
A projekt backendjének működéséért 4 alprojekt felelős (rfid-app-service-*). Az alprojektek közül a közös könyvtáron (rfid-app-service-common) kívül mindegyik egy önálló Docker konténerben futó alkalmazás (Van Dockerfile-juk). Mindegyik alprojekt tartalmaz build.gradle fájlt, ami a függőségek illetve build taskok definiálásáért felelős (később bővebben). Az rfid-app-service-production, az rfid-app-service-order és az rfid-app-service-resource forráskódon kívül egyéb erőforrásokat is tartalmaz.




**A backend alprojektek felépítése**

### 7.4.1 Közös könyvtár

A közös könyvtár nem egy különálló alkalmazás, hanem a projektben közös adatmodellek, repository-k, szolgáltatások és kivétel osztályok gyűjteménye. Az rfid-app-service-order és az rfid-app-service-production használja.




**A közös könyvtár alprojekt felépítése**

#### 7.4.1.1 Exception

Az exception package-ben találhatók azok a kivételosztályok, amik termékek rendelésekor, rfid chipek validációjakor keletkezhetnek.

```
@ResponseStatus(value=HttpStatus.BAD_REQUEST, reason="Component missing.")
public class ComponentMissingException extends RuntimeException {
    public ComponentMissingException(){
        super("Component missing.");
    }
}
```

**Példa egy kivétel osztályra (ComponentMissingException.class fájl)**

#### 7.4.1.2 Model

A model package-ben a perzisztencia fejezetben bemutatott adatmodell objektum relációs leképezése (ORM) található osztályokban, a javax.persistance.* könyvtár annotációival ellátva. Az annotációk segítségével adatbázis kényszereket, az ID generálás módját, a táblák neveit és a táblák (osztályok) közötti kapcsolatokat is be lehet állítani.

```
@Entity
@Table(name = "product_type")
public class ProductType{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Size(max = 50)
    @Column(nullable = false)
    private String name;

    @OneToMany(mappedBy = "productType")
    private Set<ComponentType> componentTypes;

    @JsonIgnore
    @OneToMany(mappedBy = "type")
    private Set<Product> instances;
/*getters, setters…*/
}
```

**Részlet a ProductType.java fájlból**

#### 7.4.1.3 Repository

A repository package-ben találhatók a repository osztályok, amik segítségével hozzáférhetünk java entitásként az adatbázisban tárolt adatokhoz. A repository-t interfészként kell definiálni, és a függvények fejlécéből a keretrendszer legenerálja az implementációt. A felesleges adatok betöltésének elkerülése végett annotációban pontosíthatjuk, hogy milyen adatokat szeretnénk betölteni.

```
@Repository
public interface ComponentRepository extends
CrudRepository<Component,Integer>
{
    @Query(
        "SELECT DISTINCT component FROM Component component " +
        "WHERE component.isReal = true " +
        "AND component.product IS NULL " +
        "AND component.id IN(:ids)"
    )
    List<Component> getRealComponentsFromIds(@Param("ids") Set<Integer> ids);
}
```

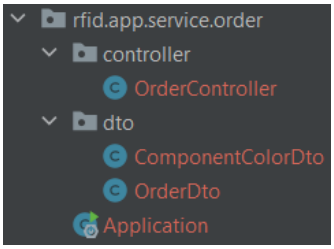
**Példa egy repository interfészre**

#### 7.4.1.4 Service

A service package-ben egyetlen osztály található, a WebPushService. Segítségével értesítéseket küldhetünk a feliratkozott webes klienseknek anélkül, hogy a kliensek kérést intéztek volna a szerver felé a feliratkozáson kívül. (A technológiáról később, a frontend fejezetben lesz szó). Ez azért szükséges, hogy ne a frontendnek kelljen időközönként lekérdeznie a frissítéseket, hanem a szerver értesíti, ha változás történt a rendszer állapotában.

### 7.4.2 Rendelés szolgáltatás

Az rfid-app-service-order alprojektben található a megrendelések kezeléséért felelős backend alkalmazás. A szolgáltatás célja, hogy a JSON formátumban érkező megrendelési adatokat validálja és ha megfelelnek az elvárásoknak akkor létrehozza a megrendelést és elmenti az adatbázisba, majd értesíti a feliratkozott klienseket, hogy új megrendelés érkezett.




**A rendelés backend alprojekt felépítése**

```
{
    "productTypeId":1,
    "components": [
    {
        "componentTypeId": 1,
        "colorTypeId": 1
    },
    …
}
```

**Példa egy megrendelés JSON kérésre.**

```
@Transactional
@PostMapping(value = "order")
public ResponseEntity<String> post(@RequestBody OrderDto orderDto){
    int id = orderDto.getProductTypeId();
    ProductType productType = productTypeRepository.findById(id);
    validateComponents(
        orderDto.getComponents(),
        productType.getComponentTypes()
    );
    Product product = createProduct(orderDto,productType);
    productRepository.save(product);
    webPushService.send("new");
    return ResponseEntity.ok("");
}
```

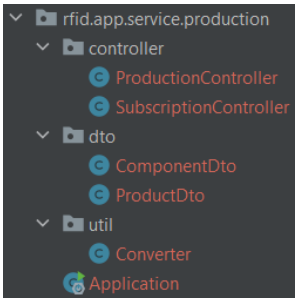
**Az OrderController két endpointja.**

A szolgáltatás működése megrendeléskor:

1. Megkeresi a megrendelt termék típust az adatbázisban.
2. Ellenőrzi, hogy a megrendelésben minden alkatrész megtalálható és a termék adott alkatrészei megrendelhetők az adott színekkel.
3. Elmenti a megrendelést az adatbázisba.
4. Értesíti a feliratkozott klienseket, az új megrendelésről.
5. Visszajelez, hogy sikeres volt a megrendelés.

### 7.4.3 Termelés szolgáltatás

Az rfid-app-service-production alkalmazás az összeszerelés frontendjét szolgálja ki. Az adat és segédosztályokat tartalmazó dto és util package-eken kívül a controller mappában lévő ProductionController és SubscriptionController tartalmazza a backend logikát. Utóbbi




**A termelés alprojekt felépítése**

controller inkább kisegítő endpointokat tartalmaz, a frontend értesítéséhez szükséges feliratkozások kezeléséért felelős.

```
@Transactional
@PostMapping(value = "/subscribe")
public Subscription subscribe(@RequestBody Subscription subscription) {
    subscriptionRepository.deleteAll();
    subscriptionRepository.save(subscription);
    return subscription;
}

@PostMapping(value = "/unsubscribe")
public void unsubscribe(@RequestBody Subscription subscription) {
    subscriptionRepository.deleteById(subscription.getEndPoint());
}
```

**A SubscriptionController endpointjai**

A ProductControllerben található az egész rendszer két legfontosabb végpontja, a getNext és a postIds metódusok. Előbbihez a production frontend intéz kéréseket és a termék összeszereléséhez szükséges komponensek adatait (színét, képének url-jét) kapja válaszul. A következő lépések történnek a lefutása során:

1. Megkeresi azokat a megrendelt termékeket, amiknek még van virtuális komponense, azaz a komponens nem egy fizikai RFID azonosítóval ellátott komponens, hanem csak az adatbázisban létezik.
2. Ha nincs ilyen, az azt jelenti, hogy minden termék, ami az adatbázisban létezik, csak valós komponensekből áll, tehát csak összeszerelt termékeink vannak, nincs még nem összeszerelt termék. (Ekkor hibát jelez a rendszer)
3. A megfelelő termékek közül kiválasztja az elsőt.
4. Ez után kiválasztja azokat a komponenseket, amiket a frontendnek meg kell jelenítenie. (Az összes már felszerelt alkatrész plusz a következő alkatrész a sorban)
5. Visszaküldi az összeállított adatokat a frontendnek.

```
@GetMapping(value = "production/next")
public ResponseEntity<ProductDto> getNext(){
    List<Product> products = productRepository.findHasNotRealComponent();
    if(products.size() <= 0){
        throw new NoAvailableOrderException();
    }
    Product product = products.get(0);
    //number of real components and +1 next component
    long nextInAssembly = product
        .getComponents().stream()
        .filter(Component::isReal).count() + 1;
    Set<Component> visibleComponents = product
        .getComponents().stream()
        .filter(component ->
            component.getType().getAssemblyOrder() <= nextInAssembly
        ).collect(Collectors.toSet());
        product.setComponents(visibleComponents);
    return new ResponseEntity<>(Converter.createProductDto(product),
HttpStatus.OK);
}
```

**A ProductionController getNext metódusa**

A postIds endpoint a PLC-től vár http post üzeneteket, konkrétan a PLC alrendszer által felismert RFID chipek azonosítóit egy JSON formátumú listában. A következő lépések történnek a kérés során:

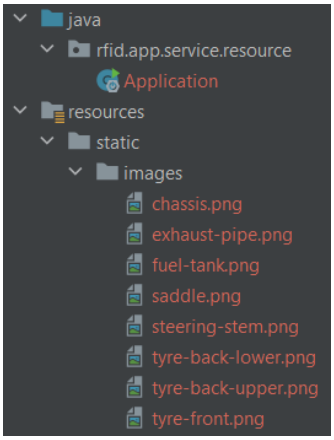
1. Kikeresi az adatbázisból, hogy a detektált komponens id-k alapján melyik félkész terméket azonosította a rendszer.
2. Kikeresi az adatbázisból azt a terméket amelyik éppen szerelés alatt áll (és a frontend az összeszerelését mutatja)
3. Összehasonlítja a két objektumot és ellenőrzi, hogy minden eddig felszerelt komponens azonosítva lett-e, van-e új, még nem felszerelt komponens, ez az új komponens megegyezik-e az elvárt következő alkatrésszel.
4. Ha valamelyik feltételnek nem felelnek meg az azonosított komponensek, hibát jelez a rendszer.
5. Ha nem történik hiba, akkor a virtuális komponenst eltávolítja a termékről a rendszer és beszúrja a helyére a valódi, RFID azonosítóval rendelkező komponenst és elmenti terméket az adatbázisba.
6. Értesíti a feliratkozott klienseket a rendszer, hogy változás történt.

```java
@Transactional
@PostMapping(value = "/production/ids")
public void postIds(@RequestBody Set<Integer> detectedIds){
    Product product = getProductByDetectedIds(detectedIds);
    Component nextRealComponent =
         getNextRealComponentByDetectedIds(detectedIds);
    Component nextVirtualComponent = getNextComponent(product);
    Set<Integer> detectedAttachedIds =
        getDetectedAttachedIds(detectedIds,nextRealComponent.getId());
    Set<Integer> registeredAttachedIds =
        getRegisteredAttachedIds(product.getComponents());
    validateNoComponentMissing(
        detectedAttachedIds,
        registeredAttachedIds
    );
    validateSubstitutability(nextRealComponent, nextVirtualComponent);

    nextRealComponent.setProduct(product);
    product.getComponents().remove(nextVirtualComponent);
    product.getComponents().add(nextRealComponent);
    componentRepository.save(nextRealComponent);
    componentRepository.delete(nextVirtualComponent);
    if(isFinished(product)) {
        webPushService.send("finished");
    }
    else {
        webPushService.send("success");
    }
}
```

**A ProductionController postIds metódusa**

### 7.4.4 Erőforrás szolgáltatás

Az rfid-app-service-resource alkalmazás az alap Springes Application class-on kívül más kódot nem tartalmaz, csak egy szervert indít el, ami azonban a megrendelési és a termelési szolgáltatással ellentétben nem REST api endpointokat definiál, hanem statikus erőforrásokat szolgál ki (képeket).

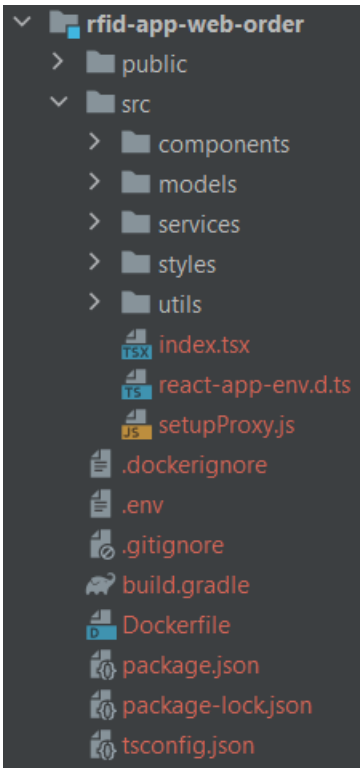



**Erőforrás backend alprojekt felépítése**

Az adatbázisba a komponensekhez felvett image_url-ekben ezeknek a képeknek az elérési útvonala található. Azért volt érdemes ezt a képmegosztó szolgáltatást egy külön alkalmazásba kiszervezni, mert a frontend termelési és szerelési része is felhasználja ezeket az erőforrásokat.

## 7.5 Frontend

A rendszer frontendjéért két react alapú alprojekt felel, az rfid-app-web-order és az rfid app-web-production. Mindegyik egy önálló Docker konténerben futó alkalmazás (Van Dockerfile-juk).




**Frontend alprojektek felépítése**

Mindegyik alprojekt tartalmaz build.gradle fájlt, ami a függőségek illetve build taskok definiálásáért felelős (később bővebben). Ahogy a technológiák részben szó esett róla, mindkét projekt TypeScript React alapú, belső felépítésük hasonló. A React projektek általános felépítésének részletezése nélkül a következő elemeket tartom fontosnak megemlíteni:

A setupProxy.js fájlban vannak definiálva a backend kérések átirányításához szükséges konfigurációk. Biztonsági okokból alapértelmezetten egy web kliens csak a saját webszerveréhez intézhet kéréseket. Ez a működés felülírható CORS (Cross Origin Resource Sharing) segítségével. Mivel a kliensek jelenesetben a spring alapú backendjükkel és a képek kiszolgálásáért felelős resource-service-szel is kommunikálnak, így szükséges volt ezeknek a beállításoknak a megléte.

```
const { createProxyMiddleware } = require('http-proxy-middleware');
module.exports = function(app) {
    app.use('/production', createProxyMiddleware(
              { target: 'http://service-production:8080', changeOrigin: true }
        ));
    app.use('/subscribe', createProxyMiddleware(
              { target: 'http://service-production:8080', changeOrigin: true }
        ));
    app.use('/images', createProxyMiddleware(
              { target: 'http://service-resource:8080', changeOrigin: true }
        ));
};
```

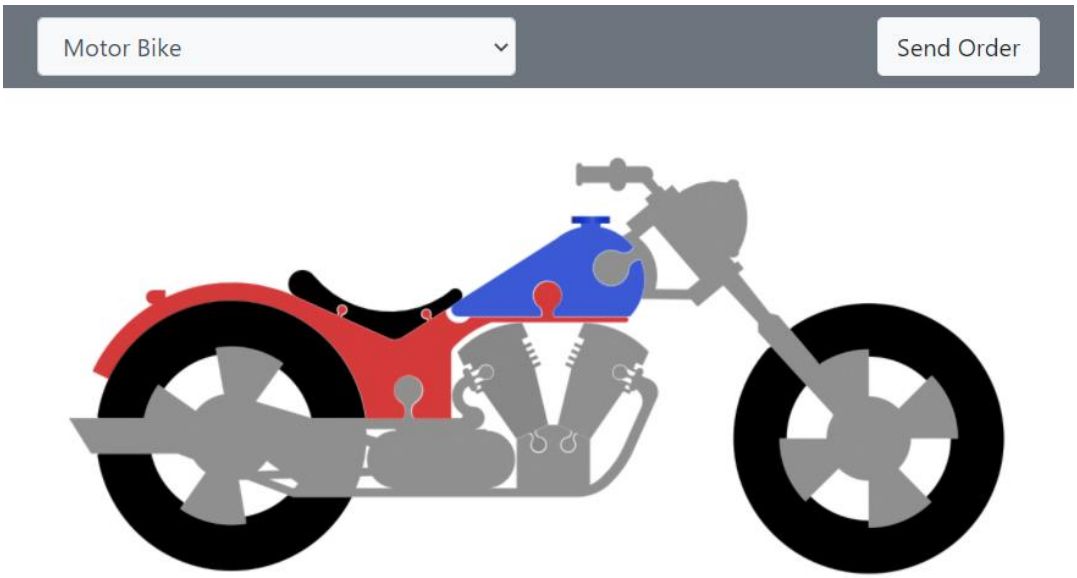
**Példa a proxy konfigurációra az rfid-app-service-production proxySetup.js-ből.**

A projektek src mappájában a components a React komponenseket, a models a backend kérések által visszaadott json objektumok interfészét, a services a backend endpointok

eléréséhez szükséges függvényeket, a styles a frontend css stílusait és az utils segédfüggvényeket tartalmaz.

### 7.5.1 Rendelés webapplikáció

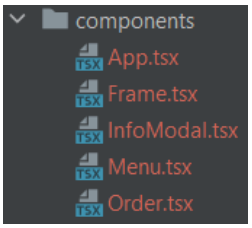
A rendelés webapplikáció (rfid-app-web-order) célja, hogy kiválasztható legyen, hogy milyen termék, milyen színű alkatrészekkel kerüljön legyártásra. Az alábbi képen látható a felület. A listából kiválasztható a megrendelni kívánt termék (jelen esetben csak a játékmotor áll rendelkezésre). A választott termék alkatrészeire kattintva pedig meg lehet változtatni az alkatrész színét. A Send Order gomb lenyomásával elküldhető a megrendelés és egy zöld felugró ablak megerősíti rendelés sikerességét, vagy egy piros felugró ablak hibát jelez, ha hiba történt a termék megrendelése közben.




**A megrendelési felület**

#### 7.5.1.1 Az alkalmazás felépítése

A React projekt az ábrán látható komponensekből épül fel. Mivel egy komponens egy hosszabb függvényből áll, ezért a kód elemzése nélkül, a komponens logikáját, felelősségeit kifejtem, a felépítését pedig rövidítve mutatom be. A {…} jelölés React komponensek attribútumainak helyét jelzi.




**A rendelés frontend komponensei**

Az App a gyökér React komponens, amit renderel az alkalmazás. Az App felelőssége a megrendelhető termék típusok

adatainak lekérése a szervertől és a megrendelések feladása. Az App három React komponenst tartalmaz.

A Menu komponens az alkalmazás tetején látható menü sáv megjelenítéséért felelős. Attribútumként megkapja a választható termék típusok nevét (a legördülő lista megjelenítéséhez) és callback függvényeket a termék típus kiválasztásához a listából és a rendelés elküldéséhez a gomb lenyomásakor.

Az InfoModal egy felugró ablak, ami a sikeres (vagy hibás) termékfeladás megerősítéséért felelős.

```
<div>
   <Menu {…}/>
   <Order {…}/>
   <InfoModal {…}/>
</div>
```

**Az App komponens felépítése**

Az Order komponens a választott termék alkatrészeinek megjelenítéséért, kiválasztásáért felelős. Attribútumként rendelkezésére áll a megjelenítendő komponensek listája és a callback függvény a komponensek színének megváltoztatásának kezelésére. A komponens érzékeli, hogy a képernyőn hova kattintott a felhasználó és továbbítja a koordinátákat (és a komponens adatait) a Frame-nek.

```
<div {…}>
   {components.map(component => <Frame {component,…}/>}
</div>
```

**Az Order komponens felépítése**

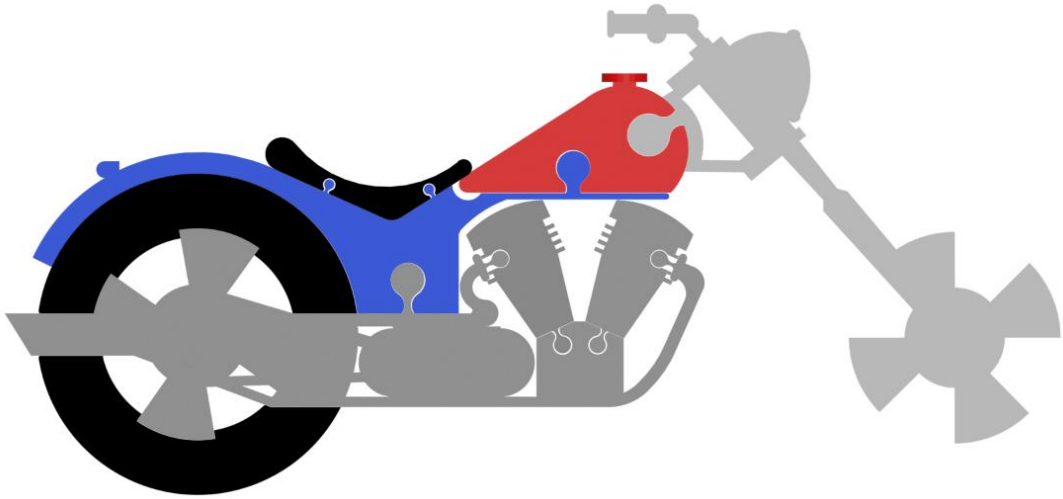
A Frame React komponens egy-egy alkatrész megjelenítéséért felelős. Egy Frame egy html img képet jelenít meg, miután kiszámolta, hogy a kattintások alapján milyen színnel kell az alkatrészt megjeleníteni.

#### 7.5.1.2 Kattintás érzékelése

Az oldal onnan tudja, hogy melyik alkatrészre kattintott a felhasználó, hogy minden kép azonos méretű és az alkatrészen kívül átlátszó. Kattintás eseményre a Frame-k megvizsgálják a saját képüket és megnézik, hogy található-e nem átlátszó pixel a kattintott koordinátán. Ha igen, akkor az adott alkatrészre kattintott a felhasználó.

### 7.5.2 Termelés webapplikáció

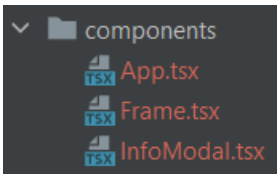
A termelés webapplikáció (rfid-app-web-production) feladata megjeleníteni a szerelésben lévő terméket. Az alábbi képen látszanak a terméken lévő alkatrészek és villog a következő alkatrész. A felülettel a felhasználónak nem kell interakcióba lépnie.



**A szerelési felület**

#### 7.5.2.1 Az alkalmazás felépítése

A React projekt az ábrán látható komponensekből épül fel. A megrendelési felülettel hasonló a felépítése, az komponensek nevei is hasonlóak, de a komponensek által kezelt adat különböző, belső működés pedig nagyrészt eltérő, mert más feladatot szolgál.




**A termelés frontend komponensei**

Az App React komponens kéri el a backendtől az összeszerelendő termék adatait és iratkozik fel a backend értesítéseire. Az InfoModal – hasonlóan az előző alkalmazáshoz – felugró ablak megjelenítéséért felel. Az ablak zöld, ha sikeres volt a szerelés és piros (hibaüzenettel) ha nem. Az előző alkalmazáshoz képest az Order komponens kimarad és rögtön az alkatrészek megjelenítéséért felelős Frame-ket tartalmazza az App komponens.

```
<div>
   {components.map(component => <Frame {component,…}/>)}
   <InfoModal {…}/>
</div>
```

**Az App komponens felépítése**

A Frame React komponens az előző alkalmazáshoz hasonlóan egy-egy alkatrész megjelenítéséért felelős. Egy Frame egy html img képet jelenít meg, de a komponensek színe adott, viszont a villogó komponenst megjelenítésével külön foglalkoznia kell.

#### 7.5.2.2 Értesítések

A rendszerben a PLC csak a production backendnek tud kérést küldeni, ha sikerült RFID tageket érzékelt. Erről viszont értesülnie kell a frontendnek is, hiszen jelezni kell a felhasználónak, hogy a szerelés sikerrel járt (és meg kell jeleníteni a következő alkatrészt a képernyőn) vagy esetleg valamilyen okból meghiúsult.

A production alkalmazás különlegessége, hogy push notification segítségével értesül, ha azonosít tageket a rendszer. A modern böngészőkben a hátterében dolgozó service worker-t értesíteni tudja a backend ha esemény történik, az oldalon működő javascript kódból pedig fel lehet iratkozni a service worker eseményeire. Így megoldható, hogy a kliensként működő webes felület kéréseket fogadjon. Az alkalmazásban a custom-service-worker.js fájl tartalmazza az értesítésekre való feliratkozást.

#### 7.5.2.3 Filter

A JavaScript, css egy hasznos funkciója volt fejlesztés során, hogy a filter css property segítségével képek színét lehet HSL értékekkel transzformálni. Így nem kellett az alkatrész képének minden színben erőforrásként léteznie, elég egy színben is és az adatbázisból kinyert hue, saturation, lightness értékekkel transzformálva megkapható a kívánt színű alkatrész. Amennyiben további színekkel kell bővíteni az alkalmazást, nem kell minden színhez új képet csinálni, csak elég a hsl értékeket felvenni az adatbázisba (és az alkatrész – szín összerendeléseket).

## 7.6 Virtualizáció/Konténerizáció

A projekben a konténerizáció, virtualizáció témával kapcsolatban a célom az volt, hogy könnyen hordozható és könnyen elindítható rendszer legyen a végeredmény. A hordozhatóság azért szükséges, mert Windows operációsrendszeren, PC-n fejlesztettem az alkalmazásokat, de Linux alapú, Raspberry Pi-on is kell futniuk, ami nem csak más operációsrendszert, de más architektúrát is jelent. A könnyű elindíthatóság azért szükséges, mert 6 egymással összekötött alkalmazást, célszerű nem egyesével, hanem egyszerre, automatizált módon elindítani.

A feladat megvalósítása 4 lépésben zajlott:

1. Először keresni kellett minden alkalmazáshoz egy olyan base image-et, ami támogatja az x64 és az arm/v7 architektúrákat.
2. El kellett készíteni a Dockerfile-t minden alkalmazáshoz.
3. Buildelni kellett a Docker Image-eket.
4. Létre kellet hozni a docker-compose.yml file-t.

### 7.6.1 Docker Image-ek

Mivel 3 különböző típusú (3 java, 2 node.js, 1 postgres) alkalmazást kell futtatni, ezért elég volt 3 base image-t keresni (ezek a Dockerfile-ok első sorában láthatók), ráadásul az azonos típusú alkalmazások hasonlósága miatt elég volt 3 Docker Image-t készíteni.

#### 7.6.1.1 Adatbázis image

```
FROM postgres
COPY sql ./docker-entrypoint-initdb.d
ENV POSTGRES_DB rfid
ENV POSTGRES_USER rfid
ENV POSTGRES_PASSWORD dev
```

**adatbázis Dockerfile**

Az adatbázis image buildelése során az adatbázis inicializáló sql scriptek átmásolódnak egy mappába, ahol a rendszer felállása után a scriptek automatikusan lefutnak, illetve, környezeti változók segítségével inicializálódnak az authentikációs adatok. Az adatbázist nem kell külön elindítani, mert az már eleve fut.

#### 7.6.1.2 Backend image

```
FROM openjdk:11-jre-slim-stretch
COPY build/libs/*.jar app.jar
CMD ["java","-jar","/app.jar"]
```

**backend Dockerfile**

A backend image buildelése során átmásolódik a buildelt java jar alkalmazás a Docker Image-be, majd elindul.

#### 7.6.1.3 Frontend image

```
FROM node:12.19.0-buster-slim
WORKDIR /app
ENV PATH /app/node_modules/.bin:$PATH
COPY package.json ./
COPY package-lock.json ./
RUN npm config set unsafe-perm true
RUN npm install --production
RUN npm install react-scripts@3.4.1 -g
COPY . ./
CMD ["npm", "start"]
```

**frontend Dockerfile**

A frontend image buildelése során nem az egész alkalmazás és a dependenciák külön átmásolása történik, mert a node_modules mappa rengeteg kisméretű fájlt tartalmaz, egyszerűbb, ha ennek a mappának a tartalmát nem másoljuk át, helyette a package.json és a package-lock.json segítségével külön települnek a dependeciák. Ezek után átmásolódnak az alkalmazás fájljai és elindul az alkalmazás.

A projekt gyökérmappájában a docker-compose up parancsot kiadva az összes alkalmazás elindul konténerben.

### 7.6.2 Docker-compose

A projekt docker-compose file-ja hosszú, viszont pontosan leírja, hogy hogyan az egyes konténerek milyen image-ből épülnek fel, hogyan állnak kapcsolatban egymással, ezért néhány egyértelmű részlet kivételével (depends_on tulajdonság, a frontendek függnek a velük egy hálózatban lévő backendektől, a backendek függnek az adatbázistól, stdin_open szükséges a frontendekhez) bemásoltam. A hálózatok felépítése a dokumentum elején látható.

A portok nyitása a frontend alkalmazásoknál volt szükséges, hogy a host gépről elérhetők legyenek a weboldalak valamint a production service nem csak a frontenddel kommunikál, hanem a PLC is küld kéréseket, amit csak a host gépen keresztül tud megtenni.

```yaml
version: '3'
services:
  database:
    image: rfidapp/rfid-app-database:latest
    networks:
      - backend

  service-order:
    image: rfidapp/rfid-app-service-order:latest
    networks:
      - backend
      - order

  service-production:
    image: rfidapp/rfid-app-service-production:latest
    networks:
      - backend
      - production
    ports:
    - 8080:8080
  service-resource:
    image: rfidapp/rfid-app-service-resource:latest
    networks:
      - resource
  web-order:
    image: rfidapp/rfid-app-web-order:latest
    networks:
      - order
      - resource
    ports:
      - 80:3000

  web-production:
    image: rfidapp/rfid-app-web-production:latest
    networks:
      - production
      - resource
    ports:
      - 3000:3000

networks:
  backend:
  resource:
  order:
  production:
```

**docker-compose.yml**

## 7.7 Build

A gradle alapú több subprojektből álló buildeléshez létre kell hozni egy settings.gradle file-t, amiben a subprojektek fel vannak sorolva, egy projekt szintű build.gradle fájlt, amiben az összes alprojektre vonatkozó konfigurációkat, dependenciákat tartalmazza, illetve minden alprojektben létre kell hozni az alprojekt specifikus konfigurációkat.

```
rootProject.name = 'rfid-app'
include 'rfid-app-database'
include 'rfid-app-service-common'
include 'rfid-app-service-order'
include 'rfid-app-service-production'
include 'rfid-app-service-resource'
include 'rfid-app-web-order'
include 'rfid-app-web-production'
```

**projekt szintű settings.gradle**

A gradle-lel java alapú projektek buildelése nem kihívás, hiszen azt különböző pluginokkal támogatja, csak a dependenciákat kell hozzáadni a gradle fájlhoz. A valódi célom a gradle használatával a docker image-k projekt szintű buildelése volt.

A projekt szintű build.gradle-ben létrehoztam egy docker nevű task-ot, ami docker buildx paranccsal több architektúrára buildeli az image-ket és feltölti őket a repository-ba.

```
subprojects{
    group = 'rfid-app'
    ext{
        dockerUserName = 'rfidapp'
    }
    repositories {
        mavenCentral()
    }
    def dockerImage = "${project.ext.dockerUserName}/${project.name}"
    def platforms = "linux/arm/v7,linux/amd64"
    task docker(type: Exec){
        workingDir "."
        commandLine "docker", "buildx", "build", "--push", "--platform",
"${platforms}", "--tag", "${dockerImage}", "."
    }
}
```

**projekt szintű gradle.build fájl**

A subprojektek build.gradle fájljaiban a buildelést követően be van állítva, hogy fusson le a docker task. Ennek hatására a build parancsot kiadva lefordul a projekt majd elkészülnek a Docker Image-ek is.

```
build.finalizedBy('docker')
```

**részlet a subprojektek gradle fájljaiból**

Ha változás történik a projektben, a gradle build és a docker-compose up parancsokat kiadva futtathatjuk az frissített alkalmazást.

## 7.8 Single-board computer

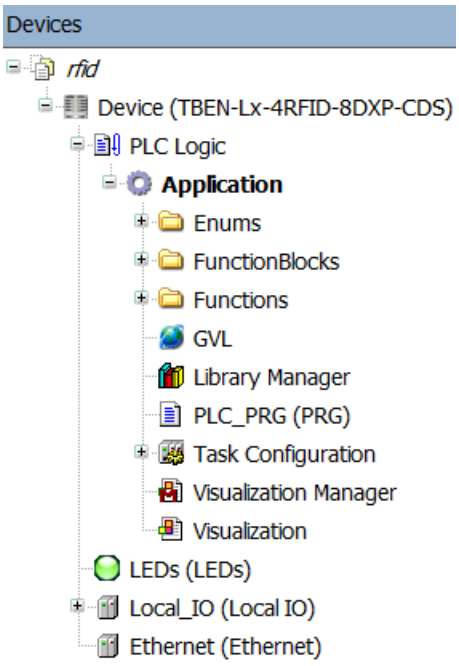
A rendszer futtatásához használt Raspberry Pi 4 mini számítógép a Linux alapú Raspberry Pi OS-t használ. A gépen Docker Engine fut, és a mikroszámítógépre a fentebb bemutatott docker-compose.yml fájlt átmásolva docker-compose up paranccsal elindítható az alkalmazás.

A Pi-t a hálózathoz csatlakoztatva az ip címén elérhető a megrendelési weboldal. A pi-n a böngészőben a localhost:3000 címen található a termelési webalkalmazás. Az oldalon a service-worker-t frissíteni kell, hogy az értesítésekre való feliratkozás megtörténjen. Ehhez az oldalon Ctrl+F5 billentyűkombináció lenyomása szükséges. A böngésző konzolban a rendszer jelzést ad a feliratkozás sikerét illetően.

## 7.9 Programozható logikai kontroller

Az RFID olvasó működtetéséhez a Turcktól kapott TBEN-L8-4RFID-8DXP-CDS PLC-re kellett programot fejlesztenem Structured Text nyelven. Mivel a PLC programozást az informatikusképzés során nem tanultunk, ezért rengeteg segítséget jelentett a Turck mérnökeitől kapott útmutatás a komponens elkészítéséhez.

A PLC Modbus protokoll segítségével áll összeköttetésben az RFID olvasóval (Local_IO), helyi hálózattal pedig TCP/IP segítségével kommunikál.




**CODESYS projekt felépítése**

### 7.9.1 Működés

A PLC program működése a következő: A rendszer bekapcsolja az RFID olvasó antennáját, megpróbál RFID tageket találni. Amennyiben nem talál, megadott idő után kikapcsol az antenna, a PLC várakozik egy ideig, majd ismét bekapcsolja az antennát. Amennyiben talál tageket az RFID olvasó, kiolvassa a tagek azonosítóit, amikhez a program a Local_IO alatt található UHF_extended regisztereiből férhet hozzá. A készülék gyártója, a Turck szoftverkönyvtárként egy RFID interfészt is készített a rendszerhez, ami

„mappeli” a regiszterek címeit és a programból funkcióblokként teszi elérhetővé. Az ID-k kiolvasása után a program egy másik könyvtár funkcióblokkját hívja segítségül a http kommunikációhoz. A raspberry ip-címére a 8080-as porton a /production/ids útvonalra (rfid-app-service-production) egy POST kérésben JSON listában küldi el az azonosított ID-kat a szervernek.

# 8 Értékelés, továbbfejlesztési lehetőségek

Ebben a fejezetben a szakdolgozat projekt elkészítése közben szerzett tapasztalataimról számolok be. Az értékelés során összehasonlítom az elképzelt rendszert a megvalósított rendszerrel, majd a továbbfejlesztési lehetőségek alatt arról ejtek szót, hogy milyen módon lehetne érdemben bővíteni az elkészült rendszert.

## 8.1 Értékelés

Az elkészült rendszer összességében az elvártnak megfelelően működik, csak néhány nemfunkcionális követelményben teljesít alul, azonban más területeken az elvártnál jobban teljesít.

### 8.1.1 Lassú indulás

Az egyik gyengesége a rendszernek a Raspberry Pi. Ugyan a legmodernebb modellt felhasználásával készült el a rendszer, azonban párhuzamosan 6 docker konténer futtatása kihívást jelent a számára, különösen a rendszer elindítása illetve a leállítása idején. Szerencsére miután felállt a rendszer, a lassulás nem jelentős, így a rendszer használható (Több erőforrással rendelkező eszközön pedig gond nélkül működik).

### 8.1.2 RFID technológia alkalmazása

A rendszer elkészítésének egyik célja az RFID eszközök stabil működésének tesztelése volt. A Turck által biztosított RFID olvasó túlteljesítette az elvárásokat, képes volt több RFID chip azonosítására, többnyire probléma nélkül. Időnként azonban túl korán, illetve túl távolról érzékelte a tag-eket és előfordult hogy egy kritikus távolságnál az alkatrészek egy részét már érzékelte, de a másik részét nem. Ez a működés az olvasó fej további konfigurációjával javítható, azonban ritkán fordult elő.

### 8.1.3 Docker

Részben arra számítottam, hogy az alkalmazások Docker konténerben való futtatása további nehézséget fog okozni a rendszer elkészítése során, ez azonban éppen ellenkezően történt. Az egyetlen bonyodalmat az image-k több architektúrára való automatizált buildelése jelentette, azonban ezen kívül mind a fejlesztés közben, mind a kész rendszer futtatásakor remek eszköznek bizonyult. Az egyetlen lépés, amit másképp

csinálnék, az az, hogy nem a félkész rendszer integrációjakor állnék neki a Docker Image-ek létrehozásának, hanem rögtön az új projekt kezdetekor készíteném el a Dockerfile-okat és a docker-compose.yml fájlt is, hogy a kiinduló projekt már egy működő alkalmazás legyen.

## 8.2 Továbbfejlesztési lehetőségek

Miként az Önállólaboratórium tárgy keretein belül készített rendszer esetében is implementáció közben derült ki, hogy egyes problémákat érdemesebb lett volna más technológiák használatával megoldani, hasonló módon, a szakdolgozat projekt elkészítése közben is fényderült ilyen helyzetekre. Mivel implementáció közben már nehéz lett volna a rendszer felépítésén változtatni, ezért azt az eredeti tervek szerint készült el, azonban néhány módosítási, továbbfejlesztési lehetőséget ismertetek.

### 8.2.1 Push Notification vs Web Socket

A rendszer elkészítése után kiderült, hogy a kliensek értesítéséhez használt Push Notification alkalmazása több szempontból is rossz választásnak bizonyult és érdemesebb lett volna helyette Web Socket technológiát használni. A Push Notification-öket nem minden böngésző egyformán támogatja, egyes helyzetekben nem megbízható, az frontend elindítása után mindig frissíteni kell az oldalt a helyes működés érdekében. A Push Notification működéséhez továbbá harmadik fél is szükséges, ugyanis a kliens a feliratkozását a szervernek küldi el, azonban a szerver az értesítést egy harmadik fél által működtetett Push Service-nek továbbítja, amely értesíti a kliens service-worker-ét az eseményekről. Ezzel szemben a Web Socket alkalmazása közvetlen kommunikációt eredményezett volna, ami gyorsabb és megbízhatóbb lett volna és internet elérésre sem lenne szükség a rendszer működéséhez. A rendszer továbbfejlesztése esetén ez lenne az első változtatás amit megvalósítanék.

### 8.2.2 További visszajelzések a felhasználónak

Konzulensem javaslatára kiegészíteném a termelési felület frontendjét egy progress barral, ami jelzi, hogy hány százaléknál tart az összeszerelés.

### 8.2.3 Több termék

Jelenleg csak egy motorkerékpár szerelése lehetséges a rendszer segítségével, azonban a szoftver úgy lett elkészítve, hogy több, különböző termék összeszerelését is támogatja, illetve az alkatrészek színei is könnyedén bővíthetők. További termékek és alkatrészek 3D nyomtatása mindenképpen látványosabbá tenné a rendszert.

# 9 Hivatkozások


[1] Demeter Krisztina, Losonci Dávid, Nagy Judit, Horváth Bálint, „Tapasztalatok az ipar 4.0-val – egy esetalapú elemzés,” 2019. [Online]. Available: http://unipub.lib.uni-corvinus.hu/4058/. [Hozzáférés dátuma: 10 12 2020].

[2] PostgreSQL Global Development Group, „PostgreSQL: Documentation,” PostgreSQL Global Development Group, [Online]. Available: https://www.postgresql.org/docs/. [Hozzáférés dátuma: 4 12 2020].

[3] Microsoft Corporation, „.NET Documentation,” Microsoft Corporation, [Online]. Available: https://docs.microsoft.com/hu-hu/dotnet/. [Hozzáférés dátuma: 4 12 2020].

[4] Microsoft Corporation, „ASP.NET Documentation,” Microsoft Corporation, [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0. [Hozzáférés dátuma: 4 12 2020].

[5] Oracle Corporation, „JDK 15 Documentation,” Oracle Corporation, [Online]. Available: https://docs.oracle.com/en/java/javase/15/. [Hozzáférés dátuma: 4 12 2020].

[6] VMWare Inc., „Spring Framework Documentation,” VMware Inc., [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/. [Hozzáférés dátuma: 4 12 2020].

[7] Microsoft Corporation, „UWP Documentation,” Microsoft Corporation, [Online]. Available: https://docs.microsoft.com/en-us/windows/uwp/. [Hozzáférés dátuma: 4 12 2020].

[8] Mozilla Corporation, „JavaScript | MDN,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Hozzáférés dátuma: 4 12 2020].

[9] Microsoft Corporation, „TypeScript Documentation,” Microsoft Corporation, [Online]. Available: https://www.typescriptlang.org/docs. [Hozzáférés dátuma: 4 12 2020].

[10] Facebook Inc., „React Documentation,” Facebook Inc., [Online]. Available: https://reactjs.org/docs/getting-started.html. [Hozzáférés dátuma: 4 12 2020].

[11] Google LLC, „Angular Documentation,” Google LLC, [Online]. Available: https://angular.io/docs. [Hozzáférés dátuma: 4 12 2020].

[12] Docker Inc., „Docker Documentation,” Docker Inc., [Online]. Available: https://docs.docker.com/. [Hozzáférés dátuma: 4 12 2020].

[13] gradle.com, „Gradle Documentation,” [Online]. Available: https://docs.gradle.org/current/userguide/userguide.html. [Hozzáférés dátuma: 4 12 2020].

[14] The Raspberry Pi Foundation, „Raspberry Pi Documentation,” The Raspberry Pi Foundation, [Online]. Available: https://www.raspberrypi.org/documentation/. [Hozzáférés dátuma: 4 12 2020].

[15] Hans Turck GmbH & Co. KG, „Product TBEN-L5-4RFID-8DXP-CDS,” Hans Turck GmbH & Co. KG, [Online]. Available: https://www.turck.hu/hu/product/00000040000380aa0003003a. [Hozzáférés dátuma: 4 12 2020].

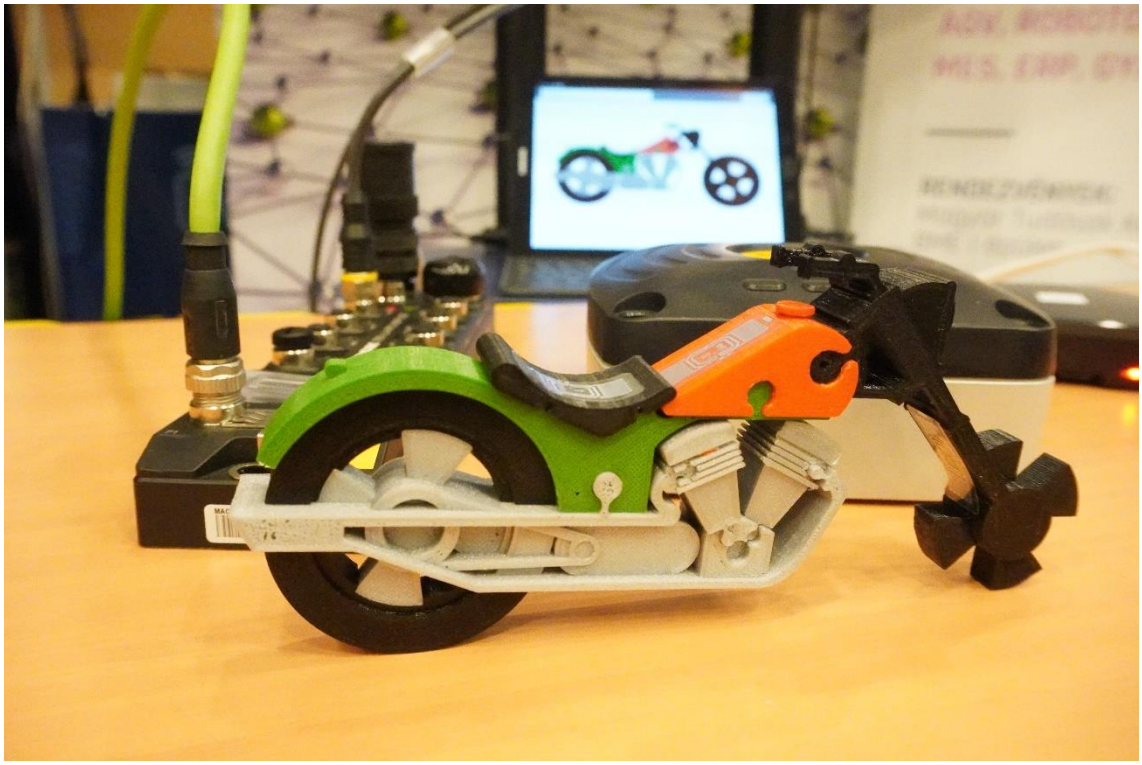
[16] Hans Turck GmbH & Co. KG, „Product TN865-Q120L130-H1147,” Hans Turck GmbH & Co. KG, [Online]. Available: https://www.turck.hu/hu/product/0000001400022645000b003a. [Hozzáférés dátuma: 4 12 2020].

[17] International Organization for Standardization , „ISO,” [Online]. Available: https://www.iso.org/standard/46149.html. [Hozzáférés dátuma: 4 12 2020].

# 10 Függelék


**Az elkészült rendszer**


**Turck PLC**



**3D nyomtatott motorkerékpár**