



SZAKDOLGOZAT FELADAT

Dongó Rajmund

Mérnökinformatikus hallgató részére

Ételrendelő alkalmazás készítése Java Spring és Angular alapokon

A mai világban egyre inkább elterjedt, hogy a különböző ételek rendelése nem telefonhívásban, hanem applikációkon, weboldalakon keresztül történik. A hallgató feladata egy e célra fejlesztett fullstack webalkalmazás megtervezése és megvalósítása.

Backend technológiaként Java Spring használandó, a frontendet pedig Angular alapokon kell megvalósítani. Az alkalmazásnak a felhasználói funkciókon felül adminisztrációs lehetőségeket is kell biztosítania a felkínált ételek kezelésére és a rendelési folyamat támogatására.

A hallgató munkájának a következő követelményeknek kell eleget tennie:

- Röviden mutassa be a megvalósításban használt technológiákat!
- Tervezzen meg és valósítson meg egy ételrendelő weboldalt, a következő képességekkel:
 - Lehessen regisztrálni, bejelentkezni.
 - Elérhető legyen mind a kereskedők, mind a vásárlók nézete.
 - Lehetőség legyen a teljes vásárlási folyamat lebonyolítására.
 - Lehetőség legyen a rendelések állapotát követni és menedzselni a kereskedők által.
 - Lehetőség legyen termékek, hozzáadására.

Tanszéki konzulens: Dr. Kővári Bence András, egyetemi docens

Budapest, 2023. szeptember 21.

Dr. Charaf Hassan
egyetemi tanár
tanszékvezető





Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Dongó Rajmund

**ÉTELRENDELŐ ALKALMAZÁS
KÉSZÍTÉSE JAVA SPRING ÉS
ANGULAR ALAPOKON**

KONZULENS DR. KÖVÁRI BENCE ANDRÁS

BUDAPEST, 2023

Tartalomjegyzék

1 Bevezetés	9
1.1 Létező megoldások.....	10
1.1.1 Foodora - NetPincér	10
1.1.2 Wolt.....	11
1.2 Motiváció	12
2 Felhasznált technológiák.....	13
2.1 Spring keretrendszer.....	13
2.1.1 A Spring általánosan	13
2.1.2 Bean.....	14
2.1.3 Inversion of control és Dependency Injection.....	16
2.1.4 Autowired.....	16
2.2 Angular keretrendszer	17
2.2.1 Komponensek.....	17
2.2.2 Adatkötések	18
2.2.3 Direktívák.....	18
2.2.4 Modulok	19
2.3 Git.....	20
2.4 SonarQube	20
2.5 Docker	20
3 Specifikáció	22
4 Tervezés.....	24
4.1 Frontend	24
4.2 Backend.....	25
4.2.1 Mikroszolgáltatások architektúra	25
4.2.2 Order tracker szolgáltatás tervezése	26
4.2.3 Payment szolgáltatás megtervezése	27
4.2.4 File manager szolgáltatás megtervezése	27
4.2.5 Autentikáció, autorizáció	28

5 Felhasználói nézetek megvalósítása	29
5.1 Általánosan a frontendről	30
5.2 Frontend nézet megjelenítése általánosan	30
5.3 Rendelési folyamat lebonyolítása.....	32
5.4 Főoldal.....	32
5.5 Regisztráció vásárlóknak.....	33
5.6 Regisztráció kereskedőknek	34
5.7 Bejelentkezés.....	34
5.8 Kereskedő oldalának nézete	35
5.9 Rendelés összegzése.....	36
5.10 Fizetés integráció.....	37
5.11 Rendelés állapotának követése.....	38
5.12 Termékek hozzáadása a kereskedőkhöz.....	39
5.13 Rendelések állapotának menedzselése	39
5.14 Profilkép feltöltése	40
5.15 Elfelejtett jelszó.....	41
5.16 Admin felület.....	42
5.17 Autentikáció, autorizáció a frontenden	42
5.18 Frontend technikai megvalósításai	44
5.18.1 Szolgáltatások és feladataik	44
5.18.2 Képek megjelenítése	45
5.18.3 Betöltési animáció	45
6 Backend megvalósítása	47
6.1 Általam választott technológiák	47
6.2 Order tracker	47
6.2.1 Adatbázis, entitások	47
6.2.2 Repository	49
6.2.3 Kérések kiszolgálása	49
6.2.4 Rendelés a backenden	51
6.2.5 Regisztráció, bejelentkezés	52
6.3 Payment service.....	55
6.3.1 Stripe integráció	55

6.3.2 Swagger.....	56
6.4 File upload service	57
6.5 Docker	58
7 Tesztelés.....	60
7.1 Unit tesztek.....	60
7.2 Sonarqube.....	61
8 Projekt összefoglalása	63
8.1 Továbbfejlesztési lehetőségek.....	63
9 Hivatkozások.....	65
10 Ábrajegyzék	67

HALLGATÓI NYILATKOZAT

Alulírott **Dongó Rajmund**, 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 átvettettem, egyértelműen, a forrás megadásával megjelöltetem.

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 eltelt után válik hozzáférhetővé.

Kelt: Budapest, 2023.12.03

.....
Dongó Rajmund

Összefoglaló

A mai világban egyre nagyobb szerepet kap a digitalizáció. Ennek hatására a különböző web applikációk is egyre több szerepet kapnak. Ezen webalkalmazások egyik példája az ételrendelő alkalmazások. A szakdolgozatomban egy ilyen web alkalmazás elkészítését fogom bemutatni és leírni a különböző lépéseket, megoldásokat, amelyek az alkalmazás sikeres elkészítéséhez vezettek.

A szakdolgozatom egy olyan webalkalmazás, amely segítségével nem csak a vásárlók, de a kereskedők oldaláról is lefedjük az alapvető funkciókat amelyeket egy ilyen alkalmazásnál elvárunk. Lehetséges a teljes rendelési folyamaton végighaladni, amely magában foglalja a termékek listázása után a kosárba helyezést és a fizetést is, amelyet Stripe integráció segítségével oldottam meg. A sikeres fizetés után a felhasználó képes követni a rendelést és egy összegzést lát a rendelés adatairól. Eközben a rendelés megérkezik a kereskedőhöz, aki látja ennek részleteit és képes változtatni a rendelés státuszát.

Dolgozatom egyik célja a mai világ népszerű technológiáinak, megoldásainak kipróbálása volt. Ennek okán igyekeztem a legjobban bevált konvenciókat alkalmazni, ahol csak lehet.

Abstract

In today's world, digitalization plays an even bigger part than before. As an effect, web applications are getting more popular. One of these kinds of applications is food ordering applications. In my thesis I will describe various steps to make such a web application, with providing information about the solutions that made this project a success.

My thesis is a web application where not only customers, but merchants have the basic functions that are required from a food ordering application. In this app we have the opportunity to navigate through the full ordering process, which consists of listing the products, putting them into a shopping cart and paying for them. The payment is made with Stripe integration. After successful checkout the customer can track his order and will see a small summary about the order information. Meanwhile the order will arrive to the merchant who will see the details of the order and will have the ability to change its status.

One of my main goals was to try out as many new technologies as I can. For the previous reason I tried to use the most recent development conventions as much as possible.

1 Bevezetés

Egy ember életének egyik elengedhetetlen része az étkezés. Az étek megszerzése az emberiség hajdanán nem volt triviális feladat, sok munka volt az ára minden egyes étkezésnek. Ezt mi sem mutatja jobban, minthogy a Neander-völgyi ember nem csak az elejtett vadakat fogyasztotta el, de képes volt rokonait is elfogyasztani [1]. Az emberiség fejlődésével a vadászatok sikere jelentősen megnőtt, egészen addig, hogy a mai világban inkább sportként tekintünk a vadászatra mintsem fő táplálék forrásunkra.

A vendéglátás története egészen az ókorig vezethető vissza, habár ekkor még nem éttermek, hanem falatozók, utcai kifőzdék töltötték be az étkeztetés szerepét. Ekkortájt jellemzően szükségből keresték fel a vásárlók az ételt kínáló árusokat, mivel nem volt lehetőségük az ételt maguknak elkészíteni.

A 13. században már megjelentek a reggelizők, “éttermek”. Itt már nagyrész kulináris élvezetek reményében térték be a vásárlók, habár ezen éttermek kiszolgálása jellemzően lassú volt és legtöbbször csak melegítették az ételt.

A 18. században elsőként Franciaországban terjedtek el a mai értelemben vett éttermek. Ebben az időben Goethe az évszázad vívmányának nevezte, hogy étlapról rendelhet ételt. Az éttermek elterjedése a választék robbanásszerű növekedését vonta maga után. Mi sem mutatja ezt jobban, minthogy Párizsban az éttermek száma több mint ötvenszeresére nőtt tíz év alatt 1978 és 1988 között. [2]

Eleinte rendelésre mindössze telefonon keresztül, vagy személyesen volt lehetősége az embereknek.

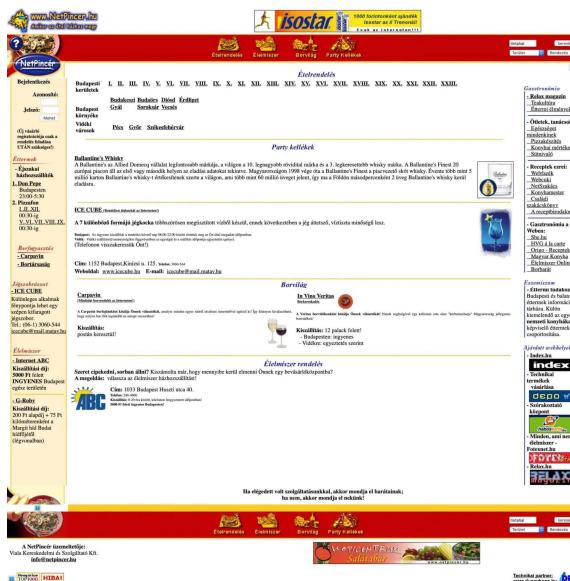
Az online ételrendelés története nem sokkal később kezdődött. A Pizza Hut első rendelését 1994. január 2.-án vette át az interneten keresztül, ráadásul ez volt az első e-kereskedelemben értékesített tárgy.

Habár a folyamat az elején lassan indult, magyarországon is elindult az online ételrendelés 1999-ben. Mivel nem volt elterjedve még a világháló használata sem, ezért ezen alkalmazásoknak csak lassan, 2010 körül sikerült nagyobb népszerűséget szerezni. [2]

1.1 Létező megoldások

1.1.1 Foodora - NetPincér

Magyarországon az online ételrendelés úttörője a NetPincér volt, amelyet 1999-ben alapított Csontos Zoltán és Perger Péter. Az alkalmazás eredeti célja inkább a Schönherz kollégiumbeli rendelések lebonyolítása volt. [3] Később az alkalmazás kinőtte a kollégium kereteit és országosan elkezdte a működését.

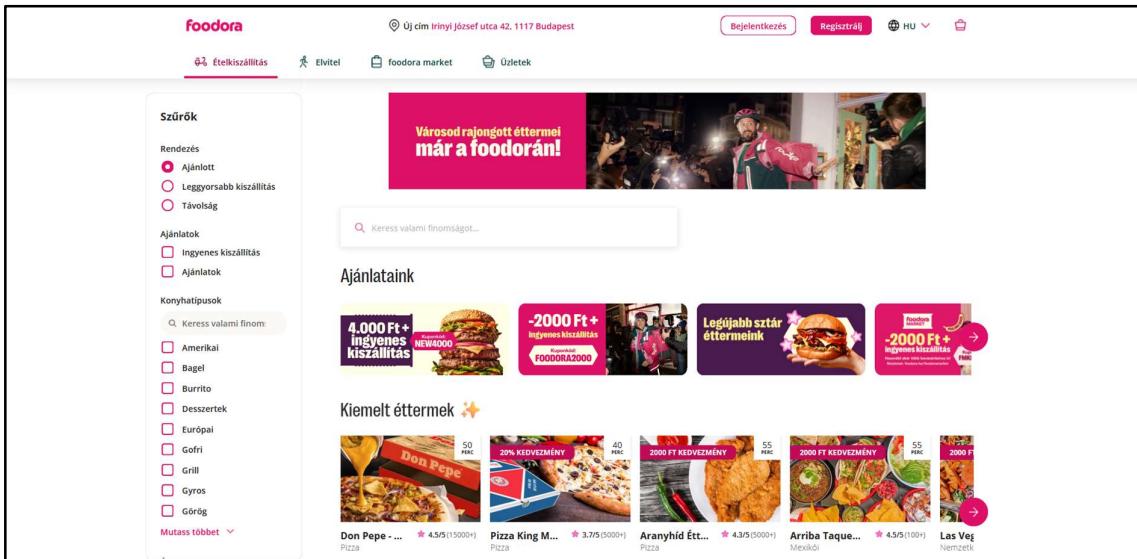


1. ábra Netpincér eredeti oldala

Az applikáció célja ekkortájt mindenki a rendelők és a kereskedők összekötése volt, ebben az időben nem tartoztak saját futárok a márkához.

Időközben az alkalmazás rengeteg változáson esett át, és a tulajdonjog is gazdát cserélt 2016-ban. A mai napon már nem Netpincérként hanem Foodoraként emlegetik az alkalmazást, ezzel pedig bekerült egy nagyobb csoport, a Delivery Hero alá. [4]

Ez az átalakulás rengeteg új funkciót hozott a vásárlóknak és az kereskedőknek egyaránt, a dedikált futárok mellett nem csak éttermekből, de bevásárló központokból is rendelhetünk már nem csak ételeket, de más termékeket is.



2. ábra Foodora mai (2023.11.07) oldala

A mai foodora weboldala rengeteg modern megoldást alkalmaz, többek között integrációt számos fizetési megoldásra, mint például az Apple és Google pay. A weboldal megjelenítését pedig a *React* keretrendszer és *JQuery* alkalmazásával készítették el. Ezen megoldások teszik lehetővé a felhasználók számára a gyors és stabil ételrendelést. [5]

A foodora magyarország legnépszerűbb ételrendelő alkalmazása lett. Sikerének oka például az, hogy több platformot is támogat, nem csak webalkalmazáson keresztül, de telefonon *Android* és *IOS* operációs rendszerekkel is elérhető.

1.1.2 Wolt

Magyarország másik jelentős ételrendelő portálja a wolt. A finn származású cég szolgáltatásai 2018 májusában lettek elérhetőek magyarországon. A fentiek ellenére a portál népszerűsége robbanásszerű sikereket ért el az országban és mára már a legnépszerűbb opciók közé tartozik, ha ételrendélsről van szó. [6]

A wolt által kínált felhasználói élmény egybevág a foodora által nyújtott élményhez, jellemzően minden étterem rendelkezik egy saját oldallal, ahol a termékekből vásárolhat a felhasználó.

The screenshot shows the Wolt website interface. At the top, there's a search bar with the placeholder "Keresés a Wolton...". Below it, there are buttons for "Bejelentkezés" (Login) and "Regisztráció" (Registration). The main content area features a section titled "Leggyorsabb kiszállítás" (Fastest delivery) with four cards:

- McDonald's® | Móricz Zsigmond utca**: Shows a burger and fries, with a note "14 nap kiszállítási idő nélkül". Delivery time: 20-30 perc.
- Hai Nam Pho Bistro**: Shows various Asian dishes, with a note "14 nap kiszállítási idő nélkül". Delivery time: 20-30 perc.
- KFC | Budapest Allee**: Shows fried chicken and sides, with a note "14 nap kiszállítási idő nélkül". Delivery time: 20-30 perc.
- Starbucks® | Allee**: Shows Starbucks coffee cups, with a note "14 nap kiszállítási idő nélkül". Delivery time: 20-30 perc.

Below this, there's a promotional banner for "W+ tagoknak" (W+ members) with a -30% discount offer. It includes images of food and Starbucks cups, and a note "adservice.google.co.in". To the right of the banner is a link "Összes megtekintése" (View all).

3. ábra Wolt mai (2023.10.27) oldala

A wolt sok szempontból hasonló funkciókat lát el, mint a netpincérből lett foodora. Lehetőség van nem csak ételek rendelésére, de elérhető havi díjas előfizetés is, amely segítségével kedvezményes rendelések bonyolíthatóak le.

1.2 Motiváció

A szakdolgozatom témajának olyan alkalmazást akartam választani, amely segítségével kipróbálhatom a webes fejlesztés alapjait és megtanulhatom az iparban használt legújabb technológiákat. Ennek okán igyekeztem a front és backend fejlesztést is magam végezni, ezzel is megtanulva a web alapjait.

Mivel magam is rengeteget használtam ételrendelő alkalmazásokat, megfigyeltem, hogy milyen sok funkcióval rendelkezik egy ilyen alkalmazás és kíváncsi voltam, hogy vajon ezeket hogyan lehetne megvalósítani egy projekt során.

Kézenfekvő megoldásnak tartottam egy ilyen ételrendelő alkalmazás elkészítését, hiszen az alapvető követelményeket, funkciókat ismertem saját tapasztalataim alapján.

Ráadásul nem tartom kizártnak azt sem, hogy az alkalmazás fejlesztése során tanult ismereteket később saját projektekhöz felhasználjam, ha kissé más formában is.

A frontend fejlesztésben a projekt kezdete előtt kevés tapasztalatom volt, ezért szakmai kihívásként tekintettem erre a lehetőségre, hogy megismerjem a kiszemelt technológiák alapjait és betekintést nyerhessek a velük történő fejlesztésbe.

2 Felhasznált technológiák

A következő fejezetben be fogom mutatni a dolgozatomban használt fontosabb technológiákat. Elsőnek a backendhez használt Spring keretrendszer, majd később a frontend technológiákat, végül pedig a felhasznált egyéb technológiákat fogom bemutatni.

2.1 Spring keretrendszer

2.1.1 A Spring általánosan

A Spring egy nyílt forráskódú java alkalmazás keretrendszer. Általában backend fejlesztés céljából használják, viszont különböző függőségek importálásával (például Thymeleaf) a frontend is megvalósítható a keretrendszer segítségével.

A Spring számos funkciót nyújt a felhasználónak, ezen funciókra pár példa:

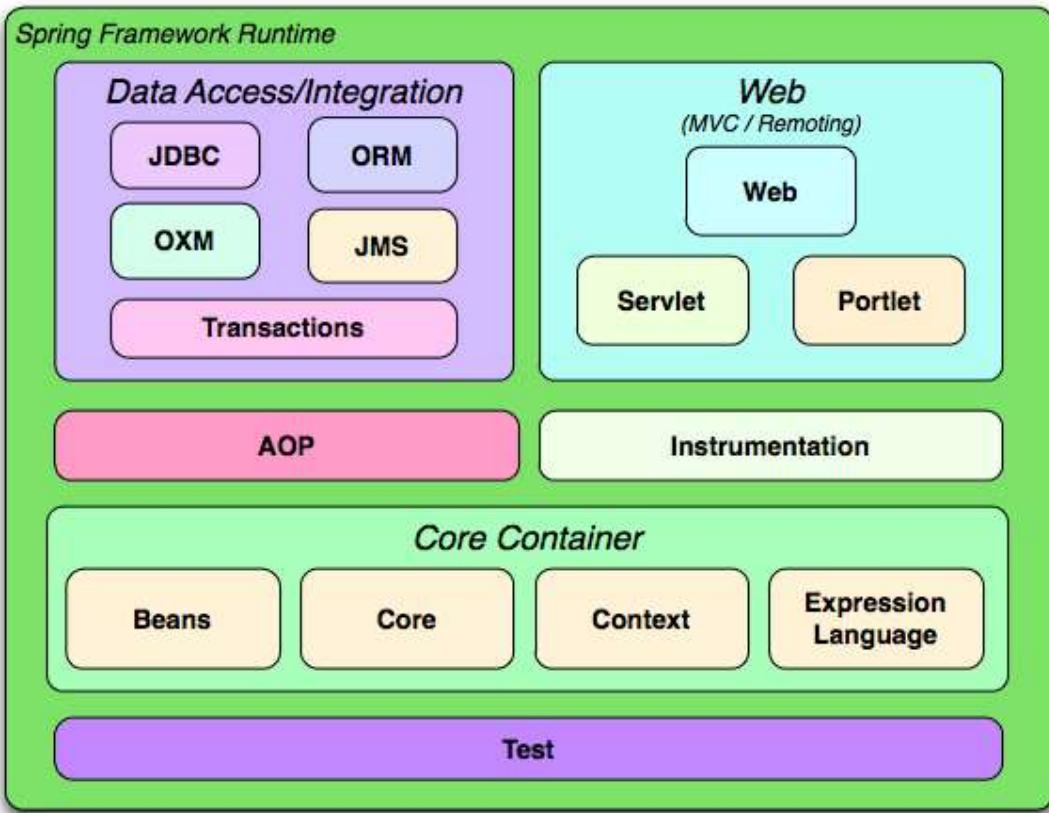
- Osztályok lazán csatolásának támogatása.
- Függőségek menedzselése, objektum életciklusok menedzselése a keretrendszer által.
- Segítségével könnyedén átalakíthatjuk a beérkező Representational State Transfer (REST) vagy Simple Object Access Protocol (SOAP) csomagokat java osztályokra.
- Támogatja Java DataBase Connectivity (JDBC) segítségével a könnyű adatbázis elérést és konfigurációt.
- Erős támogatás az Model View Control (MVC) architektúrára.
- Application Programming Interface (API) készítés támogatása könnyedén.

A fenti pontokból az osztályok lazán csatolásának támogatását szeretném kiemelni, hiszen ennek rengeteg előnye van. Többek között a lazán csatolás növeli a kód tesztelhetőségét, mivel csak egy feladatot ellátó kódrészletet is tudunk tesztelni, ráadásul növeli a kód újrafelhasználhatóságát is.

A Spring keretrendszer funkciói különböző modulokra bonthatóak, amelyek funkció szerint csoportosíthatóak.

Az egyik ilyen csoport az adat elérési csoport, amely tartalmazza többek között a JDBC és Object Relational Mapping (ORM) modulokat. Az ORM azaz objektum relációs leképezés szerepe ahogyan a nevében is szerepel, az adatbázis rekordok objektummá alakítása.

A Spring keretrendszer az alábbi különálló modulokból épül fel:



4. ábra A Spring modulok csoportosítva [7]

2.1.2 Bean

Azon objektumokat, amelyek életciklusát a keretrendszer, azon belül az Inversion of Control (IoC) konténer menedzseli, bean-nek nevezzük. A definícióból kiindulva bármilyen Plain Old Java Object (POJO) lehet bean, mindenkor a Spring konténeren keresztül kell inicializálni az objektumot, amelyhez a konfigurációs metaadatok definiálása szükséges.

Az alkalmazás által használt beaneket három módszerrel definiálhatjuk. [8]

- Leggyakoribb megoldásként annotációval jelölhetjük a keretrendszer számára, hogy egy bean hozunk létre. Ilyen annotációk a `@Service` és `@Component`
- Továbbá a projektben létrehozott Spring configuration xml segítségével megadhatjuk, ha egy osztályt beanként szeretnénk kezelní.
- Végül pedig megadhatjuk egy configuration osztály segítségével, amelyet `@Configuration` annotációval látunk el. Ebben az osztályban definiálhatunk elérési csomagokat, (package) amelyekben a Spring beaneket fog keresni `@ComponentScan` annotációval és itt is definiálhatunk beaneket `@Bean` annotációval.

Az utóbbira egy példa kódot is hoztam a könnyebb átláthatóság érdekében:

```

@Configuration
@ComponentScan(value="aut.bme.rajmundongo.szakdolgozat.utils")
public class MyConfiguration {
    @Bean
    public MyBean getMyBean(){
        return new MyBean();
    }
}

```

A példában látható `@ComponentScan` annotáció segítségével a Spring képes átvizsgálni a megadott csomagokat beaneket keresve. Lehetőség van az annotáció elhagyására amennyiben Spring Boot-ot használ a programozó, amely a Spring egy előre konfigurált változata a könnyebb és gyorsabb fejlesztés támogatásának céljából. Ilyenkor a `@SpringBootApplication` annotációval ellátott osztállyal egy szinten lévő csomagok és alcsmagok mind automatikusan szkennelésre kerülnek beanek felkutatása céljából.

A fenti megoldások közül mára az annotációk a legelterjedtebbek hiszen ez átláthatóbb és letisztultabb kódot eredményez.

Nem csak a beanek definiálására van több lehetőségünk, hanem lehetőség van a beanek hatáskörét is megadni. Ezekből 6 darab van definiálva alapból a keretrendszerben. [9]

- Singleton - Ezekből a beanekból csak egy darab jön létre egy IoC konténerben.
- Prototype - A prototype beanekból minden alkalommal új példány jön létre mikor az alkalmazásnak szüksége van rá.
- Request - Hasonló, mint a prototype, viszont webes applikációra szánva. Itt minden új példány jön létre mikor új http request érkezik.
- Session - minden http sessionre új bean jön létre.
- Application - Itt a ServletContexthez kötjük a beanek egyediségét, ami alapvetően hasonló a Singleton scope-hoz viszont itt lehetőség van arra, hogy több alkalmazás is ugyanazt a ServletContextet használja ezzel a scope-al megosztva a beant.
- WebSocket - A teljes websocket munkamenet alatt ugyanazt a beant fogja visszaadni az alkalmazás.

Az bean scope-ját a `@Scope` annotáció segítségével lehet megadni. Amennyiben számunkra nem megfelelő a Spring által biztosított alapvető scope-ok egyike sem, akkor megadhatunk saját magunk számára custom scope-okat, amelyeket személyre tudunk szabni saját kívánságaink szerint.

2.1.3 Inversion of control és Dependency Injection

A Spring fejlesztés alapja az inversion of control amely segítségével egy biztonságos, pehely súlyú, flexibilis alkalmazást hozhatunk létre. A Spring core konténer segít menedzselni az objektumok életciklusát, applikációnk konfigurálását és a függőségeket is. A feladatot a Spring az *inversion of control* tervezési minta alkalmazásával oldja meg.

Szorosan a témahez kapcsolódik a dependency injection (DI), amelyet egyes oldalakon szinonimaként említenek az IoC tervezési mintával. A két fogalom közötti különbséget a következőkben szeretném kifejteni.

Ameddig az IoC egy tervezési minta, amivel az objektumok életciklusát menedzseljük, addig a DI nem más, mint egy módszer a függőségek injektálására az alkalmazásunkban, tehát egy megoldás, amely segítségével az IoC-t el tudjuk érni.

A *dependency injection* lényege, hogy a létrehozandó objektum a saját függőségeit nem maga definiálja, hanem ezt a feladatot máshova szervezzük ki, így növelte a lazán csatoltságot.

Az alábbi példában látszik, hogy a *MyClass* példányosításakor meg kell adni, hogy milyen *MyDependency* objektum tartozzon a *MyClass* objektumhoz ezzel átadva a tulajdonság létrehozását a konstruktort hívó osztálynak, így ez egy példa a dependencia injektálásra.

```
public class MyClass {  
    private MyDependency dependency;  
    public MyClass(MyDependency dependency) {  
        this.dependency = dependency;  
    }  
}
```

2.1.4 Autowired

Az *@Autowired* annotáció segítségével a Spring képes megtalálni az alkalmazás beaneket, amelyek segítségével ki tudja elégíteni a példányosítandó osztály függőségeit.

Az annotáció használatakor a már scannelt beaneket injektálja a Spring a létrehozandó objektumba. Nem csak az osztály konstruktorában, de a függőségeiben és setttereiben is alkalmazható ez az annotáció.

Amennyiben több beant talál megfelelőnek a keretrendszer egy függőség injektálására, ekkor hiba üzenettel tér vissza az alkalmazás.

2.2 Angular keretrendszer

Az Angular egy nyílt forráskódú keretrendszer typescriptben írva. Létezik egy másik szintén elterjedt javascript alapú keretrendszer is, amelyet AngularJS-nek hívnak viszont én az előbbivel foglalkozom a szakdolgozatomban.

Az Angular keretrendszerben történő fejlesztés során egyoldalas kliens applikációkat hozhatunk létre, (single page client application) amelyeket typescript és html segítségével készíthetünk el.

Elérhetőek továbbá dekorátorok, amelyek, a Spring annotációkhöz hasonlóan metaadatokat tárolnak az osztály, tulajdonság vagy metódusról. Ezen dekorátorok segítségével egyszerűen konfigurálhatjuk a kód bizonyos részeit.

2.2.1 Komponensek

Minden Angular applikáció komponensekből áll. Ezen komponensek pedig modulokba vannak rendezve, tehát az Angular alkalmazás modulok csoportjával van definiálva. minden alkalmazás rendelkezik gyökér modullal, (root module) amely engedélyezi az alkalmazás betöltését.

Egy Angular komponens pedig 3 elemből áll:

- Hyper Text Markup Language (HTML) sablon, amely leírja mit rendereljen a felhasználói interfészre a program
- TypeScript osztály, amely az oldal viselkedését határozza meg
- Cascading Style Sheets (CSS) amely megadja hogyan használjuk a komponenst a sablonban

```
@Component({
  selector: 'my-component',
  templateUrl: './my.component.html',
})
export class MyComponent
```

A fenti typescript fájlban található példán látható, hogy a `@Component` dekorátorral megadom a CSS selectort, amelyre a komponens példányosodik, és az alatta lévő sorban feltűntem a sablonhoz tartozó elérési útvonalat. Egy másik lehetőség a sablon direkt definiálása is a dekorátoron belül.

Minden komponens a felhasználói felület egy részének megjelenítéséért felel. Ezen komponensek sokszor újrahasznosíthatóak, és egy másik jellemzőjük, hogy jól kivehető feladatokat valósítanak meg az alkalmazásokban.

Fontos megjegyezni, hogy a komponens csak felhasználja az alkalmazás üzleti logikáját szolgáltatások (service) formájában, így ez nem a komponensben kerül megírásra.

2.2.2 Adatkötések

Adatkötések segítségével képesek vagyunk HTML elemek és komponensek tulajdonságainak módosítására. Az Angular keretrendszer többféle lehetőséget biztosít az adatkötések létrehozására.

Az Angular 3 eltérő típusú adatkötést különböztet meg:

- Adatkapcsolati rétegből a megjelenítési réteghez
 - Erre példa: `[src] = "myicon"`, ilyenkor az alkalmazás betölti a myicon-t a typescript fájlból.
- Megjelenítési rétegből az adatkapcsolati réteghez
 - Erre példa: `(click) = "logout()"`, ebben az esetben kattintás eseményre meghívódik a *logout()* metódus.
- Kétirányú kapcsolat
 - Erre példa: `[(ngModel)] = "searchQuery"`, ekkor bármelyik réteg képes lesz változtatni a *searchQuery* értékét.

2.2.3 Direktívák

A direktívák segítségével új viselkedést tudunk definiálni a sablon elemeinek vagy módosíthatjuk az eddig viselkedést.

Az Angular két típusú direktívát különböztet meg. Léteznek attribútum és strukturális direktívák. Az attribútum direktívák a már meglévő Document Object Modell-hez (DOM) rendelnek kiegészítő viselkedést, ameddig a strukturális direktívák megváltoztatják a DOM struktúráját.

Attribútum direktívák közé tartozik az `[ngStyle]` amely segítségével módosíthatjuk egy html elem stílusdefinícióját, `[ngClass]` segítségével pedig hozzáadhatunk és

eltávolíthatunk CSS osztályokat a HTML elemről. Továbbá a fentebb már említett `[ngModel]` segítségével kétirányú kötést hozhatunk létre.

Strukturális direktívák között megtalálható az `*ngIf` amely segítségével feltételhez tudjuk kötni az egyes elemek megjelenését a DOM-ban. Van lehetőség `*ngSwitchCase` segítségével több feltételt is megadni, hogy a teljesült feltételnek megfelelő elem jelenjen meg a DOM-ban. Végül pedig megadható akár egy `*ngFor`, amely segítségével egy lista elemeit ismétli meg a DOM-ban az alkalmazás.

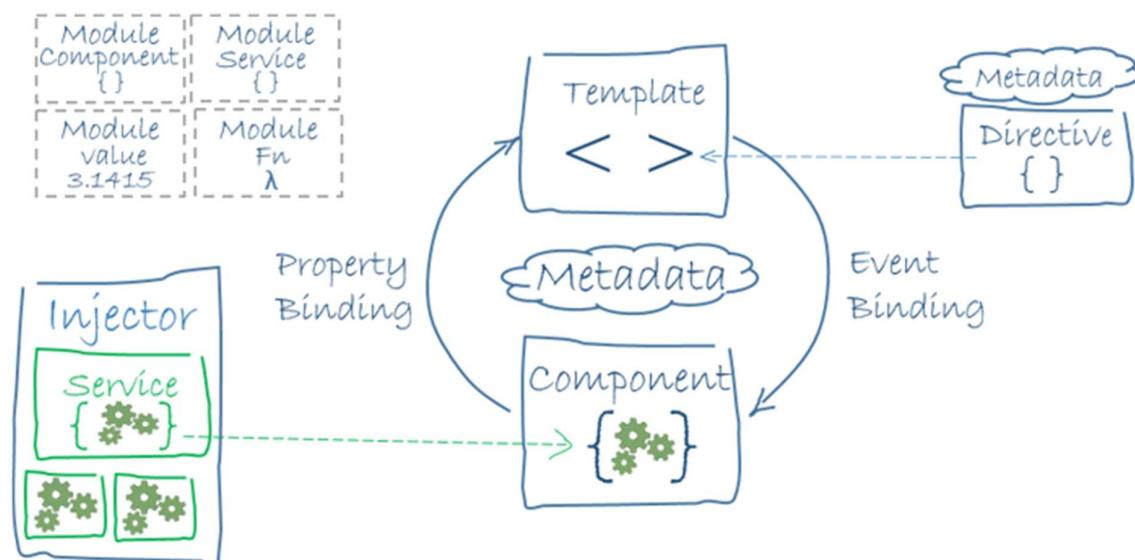
2.2.4 Modulok

Az Angularban szorosan összetartozó kódokat modulokba szervezzük. Szükség esetén pedig ilyen modulokat tudunk a kódba importálni.

Egy modult az `@NgModule` dekorátorral lehet definiálni. Ez egy olyan dekorátor, amely egy metaadat objektumot vár bemenetként és segítségével definiálhatjuk a modul részleteit.

Itt az alábbi tulajdonságokat definiálhatjuk:

- *declarations* - Komponensek, direktívák, amelyek a modulba tartoznak.
- *exports* - Azon részhalmaza a deklarációknak, amely másik modulokból, sablonokból elérhető lesz.
- *imports* - Azon modulok, amelyek exportját felhasználja a modul.
- *providers* - Az üzleti logikát megvalósító osztályok (service) felsorolása.
- *bootstrap* - A fő applikáció nézet, gyökér komponens definiálása. [10]



5. ábra Angular modulok architektúrája [11]

2.3 Git

A git egy ingyenes nyílt forráskódú verziókezelő rendszer, amely segítségével könnyedén tudjuk az alkalmazásunk verziót kezelní. Használata mára már sztenderd a szoftverfejlesztésben, mivel rendkívül gyors és könnyen tanulható eszköz. Népszerű mivel elősegíti a kooperációt a csapatban belül és egy erős eszközt ad a fejlesztőnek az alkalmazása verziókövetésére.

A rendszer segítségével:

- Lehetőség van az alkalmazás verziói perzisztálására
- Alapvető kollaborációra fejlesztők között
- Módosítások összefésülésére
- Módosítás történet visszakeresésére

A szakdolgozatom, és a hozzá tartozó módosítások megtalálhatóak a hozzá tartozó publikus repositoryban, amelyhez az oda vezető URL a hivatkozások pont alatt található. [12]

2.4 SonarQube

A sonarqube egy szintén nyílt forráskódú statikus analízis eszköz, amely segítségével a fejlesztő folyamatosan visszajelzést kaphat a kód minőségéről és esetleges szükséges változtatásokról, hibákról. Rengeteg nyelvet támogat és működése is rendkívül gyors.

Szintén rendkívül elterjedt eszköz, hiszen nem csak a tesztlefedetséget ellenőrzi, de a kódban rejlő biztonsági hibákra is visszajelzést ad. Gyakori megoldás a sonarqube futtatása minden egyes alkalommal mikor az alkalmazás kitelepítésre kerül, vagy új változtatás érkezik.

2.5 Docker

A docker platform szolgáltatásként (PaaS - Platform as a Service) termékek csoportja, amelyek operációs rendszer szintű virtualizációt használnak, hogy úgynevezett konténerekben futtassák azokat. [13]

A konténerizáció fontos tulajdonsága, hogy ezek a konténerek pehelysúlyúak és rendkívül egyszerű őket létrehozni, amennyiben megfelelően vannak konfigurálva. Az operációs rendszer szintű virtualizáció lényege, hogy az alkalmazás a futtató operációs rendszertől függetlenül tudjon működni. Konténerizáció során a programok izolálva futnak egymástól a saját függőségeikkel és az előre megadott csatornákon keresztül érik el egymást.

Megadhatunk úgynevezett docker compose fájlokat is, amelyek segítségével egyszerre több konténert indíthatunk el és definiálhatjuk a konténerek tulajdonságait, mint például:

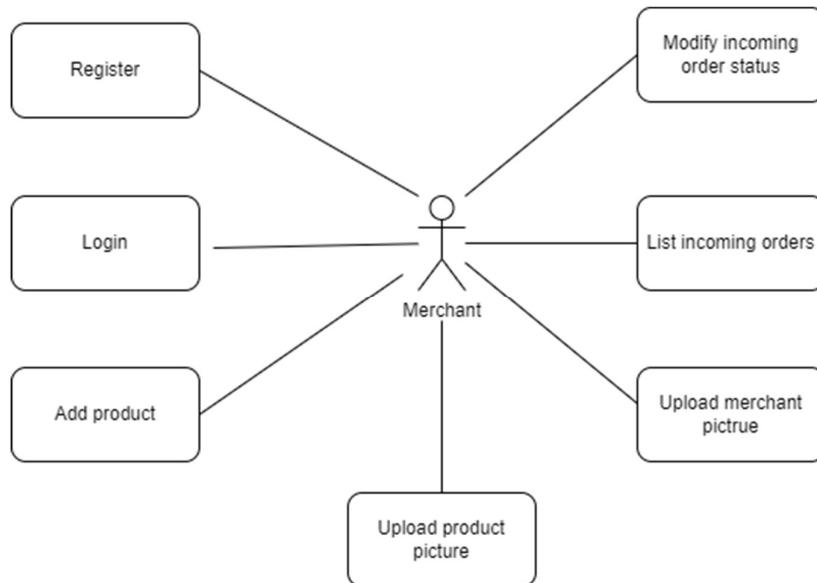
- Egymáshoz való csatlakozáshoz szükséges portokat
- Adatbázis esetén az eléréshez szükséges adatokat, mint URL és azonosító adatok
- Függőségeket egymástól, például ne induljon el a frontend backend nélkül.

3 Specifikáció

A feladatkiírás szerint, a kész applikációnak számos funkcióval rendelkeznie kell. A feladat egy olyan webalkalmazás készítése, amely a felhasználó számára intuitív navigációt tesz lehetővé és képes ellátni egy ételrendelő portál fő funkcióit.

Egyik követelménye a szakdolgozatomnak a regisztráció és bejelentkezés implementációja, itt 3 különböző felhasználói szintet különböztettem meg. Ezek az *ADMIN* (*adminisztrátor*), *CUSTOMER* (*vásárló*) és *MERCHANT* (*kereskedő*).

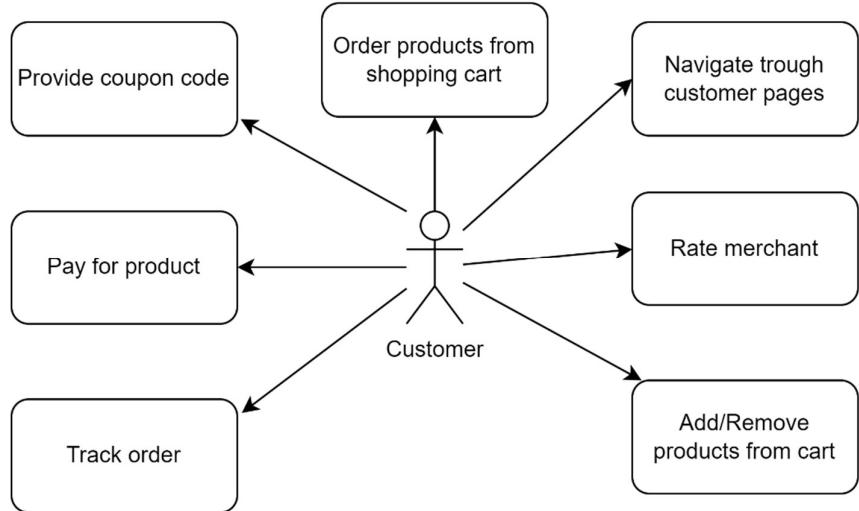
A fenti felhasználóknak különböző műveletekhez lesz jogosultságuk.



6. ábra Kereskedőhöz tartozó használati esetek.

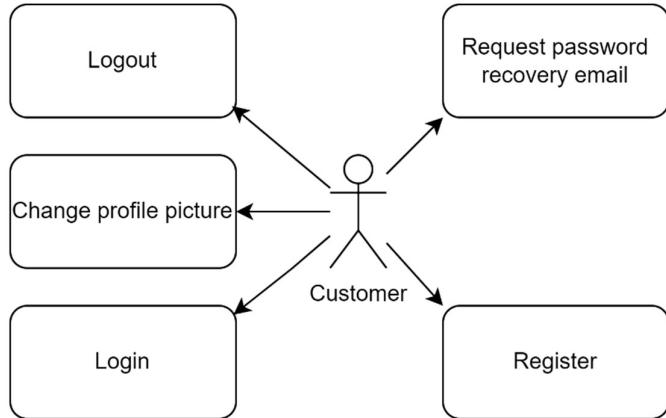
A fenti ábrán a kereskedőkhöz tartozó felhasználási lehetőségeket listázza, amely tartalmazza többek között a termékek hozzáadását és a beérkezett rendelések listázását, majd ezek státuszainak módosítását, ezzel adminisztrációs lehetőséget biztosítva a kereskedőnek a rendelései felett. Eközben az *ADMIN* jogosultságú felhasználóhoz minden össze egy használati eset tartozik, ez pedig az egyes fiókok törlése, így a regisztrált felhasználói fiókok törlésre kerülhetnek amennyiben ez indokolt.

A lenti diagramon a vásárlóhoz tartozó használati esetek vannak felsorolva, ez magában foglalja a kereskedő értékelését, lehetőséget ad többek között kuponok alkalmazására, a kosár tartalmának módosítására és a rendelések követésére is.



7. ábra Vásárlóhoz tartozó felhasználói használati esetek.

A vásárlóknak emellett rendelkeznek a kereskedőnél megszokott profil kezelésére alkalmas funkciókkal is.

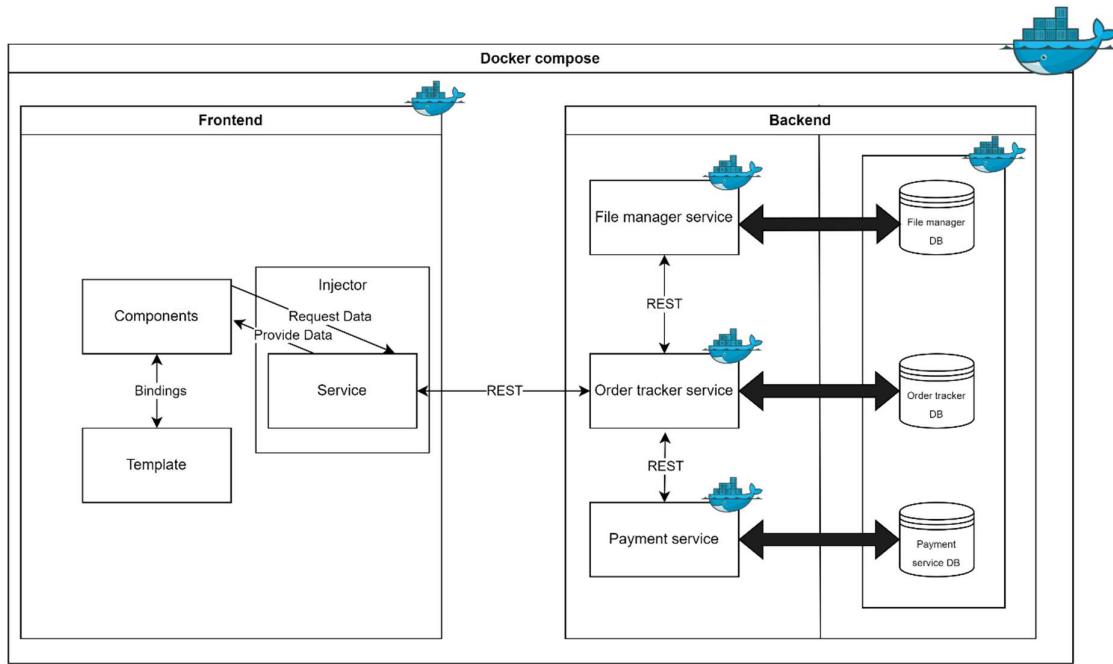


8. ábra Vásárlóhoz tartozó profil kezelésére alkalmas használati esetek

Az applikációban a rendelési folyamat az alábbi lépések soraként valósul meg: A vásárló regisztrál vagy bejelentkezik majd a főoldalra kerül, ahol kilistázónak a kereskedők, itt egy kereskedő kiválasztása után megtekintheti annak termékeit majd ezek közül a bevásárlókosárba adhat hozzá terméket, vagy távolíthat el. Amennyiben kiválasztásra kerültek a termékek átnavigálhat a rendelés összegzésének oldalára, ahol kuponokat aktiválhat a felhasználó kedvezményekért. Az ezt követő oldal a fizetés funkciót valósítja meg, majd a sikeres fizetés után átirányítás történik rendelés követésének oldalára, ahol megtekinthetők a rendelő és rendelés adatai a rendelés státuszával.

4 Tervezés

A következőkben egy részletesebb leírást szeretnék adni az általam elkészített megoldás terveiről. Első körben két nagyobb részre bontottam az alkalmazást, az Angular frontend és a Spring backendre.



9. ábra Alkalmazás architektúrája

4.1 Frontend

A frontend tervezésekor minden nézetet külön komponensként kezeltem. Egyedüli kivétel a *Header komponens*, amely az alkalmazás nézeteinek keretét adja ezért az utóbbi külön komponensként lett létrehozva.

Ezen komponensek külön csomagokban (package) helyezkednek el a hozzájuk tartozó mintával és typescript fájjal együtt, amelyek a technológiák részben említett adatkötésekkel vannak összekapcsolva.

Service csomagban az üzleti logika megvalósítása történik általánosan, amely az alkalmazásomban legtöbbször egy REST kérés formájában valósul meg. Ezen csomagban található szolgáltatásokat (service) használom például, mikor a felhasználó átnavigál egy weboldalra és a backendről elkörem a megjelenítéshez szükséges adatokat.

Model csomagban különböző osztályokat definiáltam, amelyek a ki és bemenő requesteknek megfelelően megkönnyítik az adatok küldését és fogadását azzal, hogy egy struktúrát adnak az utazó adatoknak.

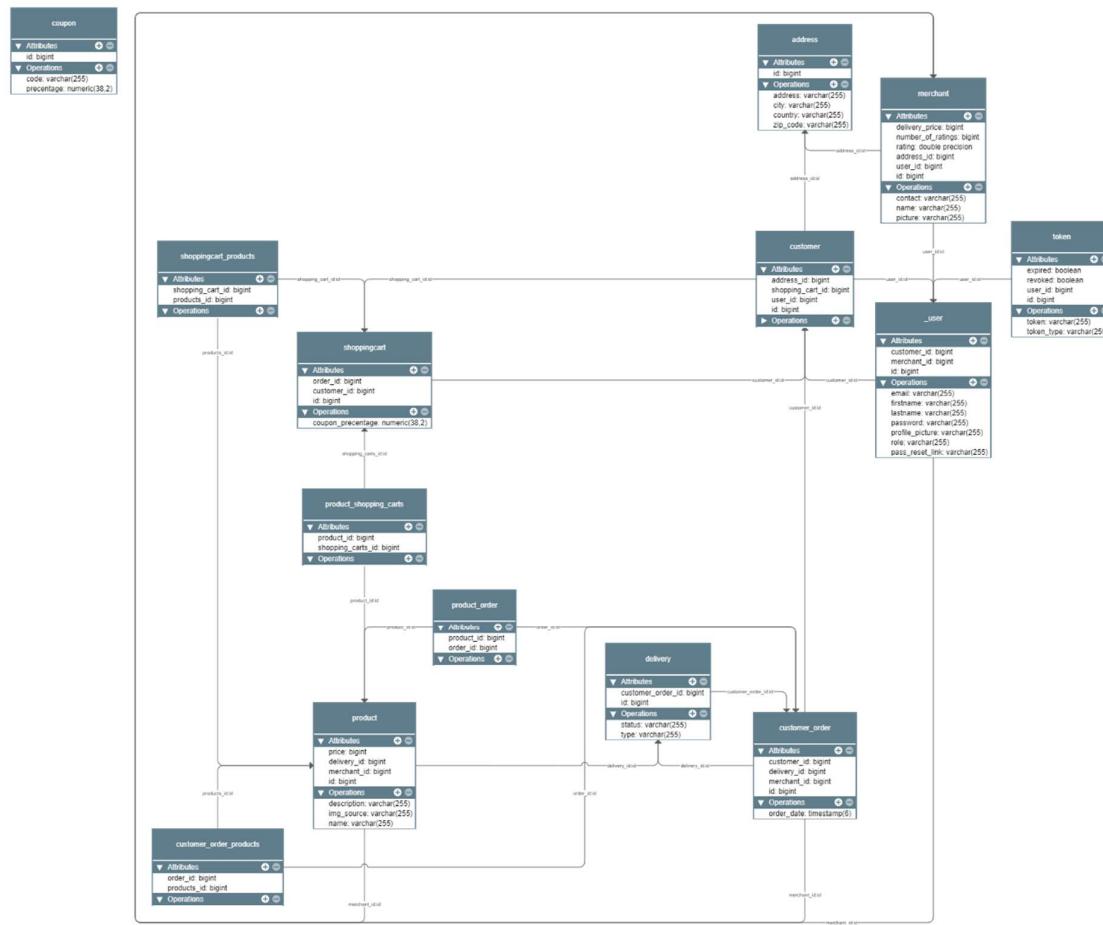
Az alkalmazás elkészültével 15 különböző komponens, 12 service és 9 model osztály jött létre.

4.2 Backend

4.2.1 Mikroszolgáltatások architektúra

A szakdolgozatomban az üzleti logika jelentős részre a backend oldalon kerül megvalósításra, ezért igyekeztem a lehető legátláthatóbb struktúrában kialakítani a szolgáltatásokat, amelyek az alkalmazás adatait fogják biztosítani. Ennek okán és a könnyű átláthatóság, bővíthetőség érdekében nem egy, hanem három darab különböző Spring applikációba szerveztem ki a backend működését. Az alábbi megoldás a mikroszolgáltatás architektúra alá tartozik, hiszen minden alkalmazás egy feladatot lát el és ezen alkalmazások önállóak és saját adatbázissal rendelkeznek. Itt érdemes megjegyezni, hogy az Order Tracker service szigorúan véve nem mikroszolgáltatás hiszen nem egy jól definiált feladatot lát el.

4.2.2 Order tracker szolgáltatás tervezése



10. ábra Order tracker szolgáltatás adatbázis architektúra

Első lépésben az order tracker szolgáltatás adatbázis architektúráját terveztem meg. Itt fontos tervezési szempont volt a rendelés minél könnyebb feldolgozásának támogatása és az is, hogy az egyes táblák elkülöníthető feladatakkal rendelkezzenek, továbbá fontos szempont volt a bővíthetőség támogatása. Az előzőre egy példa, ha akarunk kategóriákat hozzáadni a termékekhez, azt egyszerűen egy Product táblában definiált enum-al megtehetjük.

Igyekeztem továbbá elkülöníteni a rendelési folyamat elemeit külön táblákba, ezért a rendelésnek minden össze egy tulajdonsága a delivery. Ennek köszönhetően nem kell minden a teljes Order objektumot válaszként küldeni a felhasználónak, hanem minden össze elég egy Delivery objektumot küldeni. Természetesen az utóbbi Data Transfer Object (DTO) osztályok használatával is megoldható lenne, viszont ezzel a megoldással jobban sarkallom a fejlesztőt az irányelveim követésére.

Egy kissé különálló tábla a coupon, amely nem kapcsolódik a többi táblához, mivel ez minden össze az adatbázisban való tárolás érdekében jött létre.

A felhasználó kezeléshez a User, Customer, Merchant és a Token táblák vonatkoznak és a rendelés folyamatát az Order, Product, Delivery, Address és a ShoppingCart táblák segítik elő.

Az alkalmazás 52 végpontot tartalmaz, amelyek képesek kiszolgálni a frontend jelenlegi igényeit.

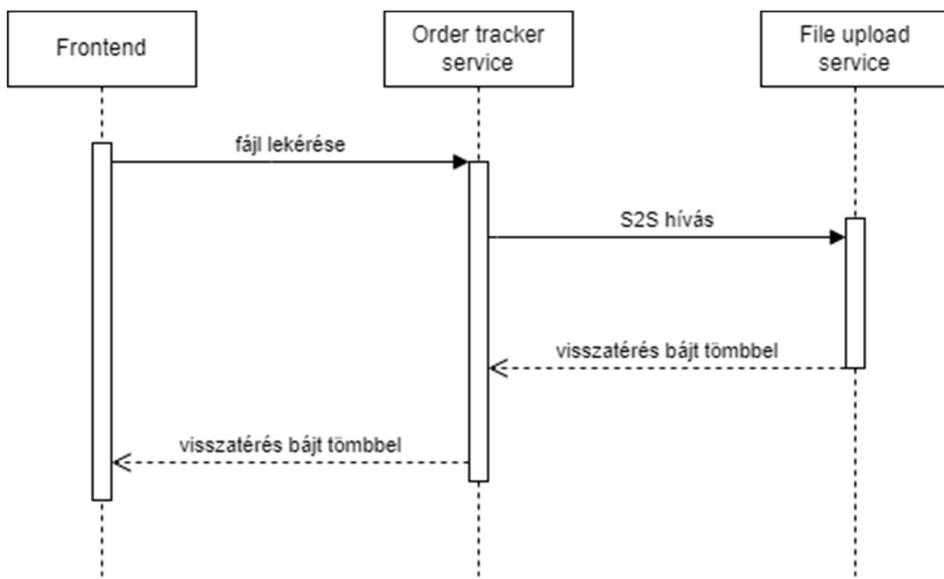
4.2.3 Payment szolgáltatás megtervezése

Itt főleg a Stripe integráció kiszolgálására volt szükség, tehát a tervezési szakasz leginkább a Stripe dokumentáció olvasását és a szükséges endpointok megértését tette ki. A szolgáltatás minden össze 2 táblával rendelkezik, amelyek a Credentials, amely a Stripehoz használt API kulcsot tartalmazza és a Product, amely a Stripe-on belül létrehozott termékek adatainak tárolására szolgál. Az alkalmazás egy végponttal rendelkezik, amely a fizetéshez használt URL-t adja vissza a bemenő adatok függvényében.

A pontos megvalósítást később, a megvalósítás fejezetben mutatom be.

4.2.4 File manager szolgáltatás megtervezése

Ebben a szolgáltatásban minden össze az adatok tárolására volt cél, itt három végpontot terveztem, amelyekből kettő adatok lekérdezésére és egy az adatok feltöltésére szolgál. Egy darab táblát szántam az alkalmazásnak, amely a fájlok tárolására szolgál.



11. ábra Képek lekérdezése a frontendről

Ezt a szolgáltatást nem közvetlenül a frontend fogja hívni a benne megírt szolgáltatásokkal, hanem a frontend egy kérést fog küldeni az ordertracker szolgáltatásnak, amely egy szolgáltatás a szolgáltatásnak („Service to service” vagy S2S) kérés segítségével manipulálja vagy kérdezi le az adatbázisban tárolt képeket.

4.2.5 Autentikáció, autorizáció

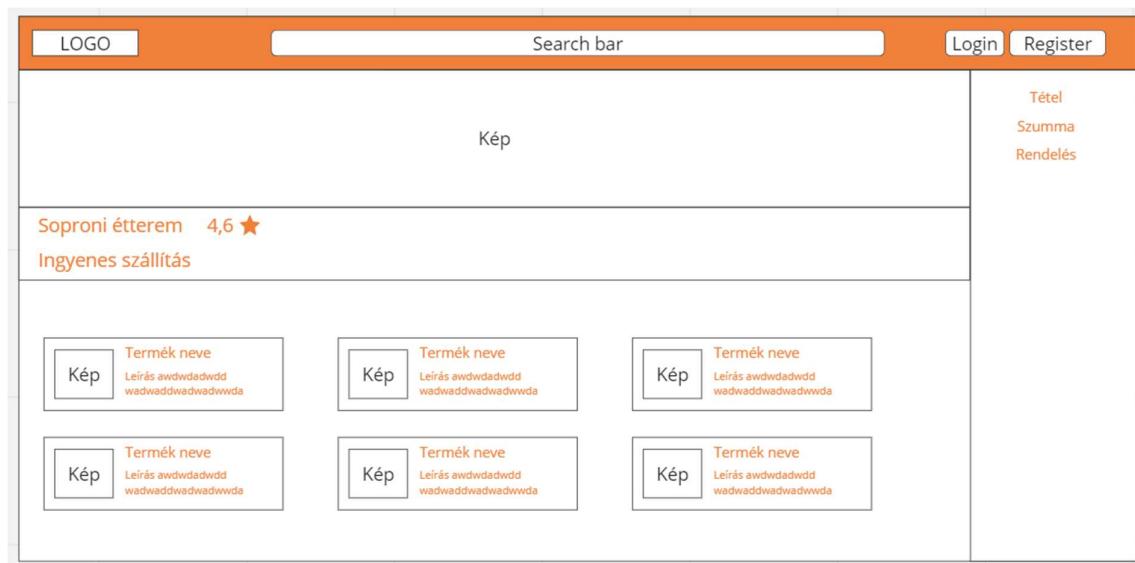
Az autentikáció, autorizáció megvalósítására a legszimpatikusabb technológia számomra a JWT (Json Web Token) használata volt. Alternatívaként felmerült egy külső szolgáltatás (third party) bevonásának ötlete is, viszont az előbbi megoldás rengeteg fejlődési lehetőséget rejtett, mindenkorban növelte a szakdolgozat értékét.

5 Felhasználói nézetek megvalósítása

A megvalósítás első lépéseként, a szükséges nézetek meghatározása után, megterveztem a felhasználói nézeteket.

A tervezéshez itt elsőnek mockup-okat készítettem, amely jelentése "minta", "mintadarab". Ezen látványtervek elsődleges célja az volt, hogy az egyes oldalak kinézetét, funkcionálitását, navigálási lehetőségeit jól kivehető ábrákban megjelenítsem. Természetesen a fejlesztés során igyekeztem a lehető legjobb felhasználói élmény nyújtó megoldások alkalmazására és az intuitív tervezésre, hogy a lehető legjobb felhasználói élmény tudjam nyújtani az oldal látogatóinak.

Tervezéshez a *miro* nevű kollaborációs platformot használtam, amely segítségével elkészítettem a felhasználói nézetek tervezet.



12. ábra Kereskedő oldalának nézete mockup

A fenti ábrához hasonlóan készítettem el a többi nézet tervezet is, amelyek nagyban segítettek a projekt előrehaladásában. Habár az ábrán nem látszik, az egyes nézeteket összekötöttem a hozzájuk tartozó navigációs gombbal, ezzel biztosítva, hogy minden nézet elérhető a felhasználói felületről. Erre egy példa az ábrán látható regisztrációs gomb összekötése a regisztráció nézettel.

5.1 Általánosan a frontendről

A nézetek megtervezése után elkészítettem a weboldal felhasználói felületét, amely megvalósításakor több helyen hasonló elemeket használtam, így ezeket a több helyen előforduló elemeket szeretném most összefoglalni.

A webalkalmazás tervezésekor a frontendnek elsődlegesen a megjelenítés feladataitát szántam, ennek köszönhetően kevés üzleti logika került ide és a szolgáltatások nagy része backenden került megírásra.

Igyekeztem felgyorsítani a fejlesztést, ezért az általam már ismert *TailwindCSS* Cascading Style Sheets keretrendszer alkalmaztam a fejlesztéshez. A döntésem oka nem csak az egyszerűbb szintaxis, de a kód átláthatóságának növelésében és a keretrendszer jól dokumentáltságában rejlik.

5.2 Frontend nézet megjelenítése általánosan

Általánosan a frontenden a nézetek megjelenítését szeretném bemutatni egy példán keresztül. Az általam választott példa a *rendelés követése* nézet, amely implementációja hasonló megoldásokat tartalmaz, mint amit a többi komponensben alkalmaztam.

A megjelenítés 3 fő komponense a minta, amely a HTML kódot tartalmazza, továbbá a Typescript fájl, amely a komponenshez tartozó logikát és a szolgáltatás osztály, amely az üzleti logikát végzi, ami a jelenlegi esetben főleg a backenddel való kommunikációban valósul meg.

Első lépésben az oldal inicializálásakor lefut a Typescript fájlban található *ngOnInit()* metódus, amely segítségével lekérdezésre kerülnek a szolgáltatások által a megjelenítéshez szükséges adatok.

```
this._shoppingCartService.getShoppingCartOrderId(this.customerId).pipe(
  catchError(error => {
    console.error('Error occurred while fetching ShoppingCartOrderId:', error);
    this.navigateBack();
    return throwError(error);
  })
).subscribe(data => {
  if(data==null) {
    this.navigateBack();
  }
})
```

A fenti példakódban található a rendelés azonosítójának lekérdezése, amely hiányában a *navigateBack()* metódus fog meghívódni ami a kezdőlapra navigál. Tehát nem lehet megnyitni ezt az oldalt anélkül, hogy rendelésünk lenne.

A fent hívott szolgáltatás metódus implementációjában a vásárló azonosítója alapján egy GET kérést küld az alkalmazás a backend számára, amely egy Observable azaz „megfigyelhető” számmal tér vissza, amire azért van szükség mivel a backend válaszára várnival azsinkron művelet és így a fordító figyelni fog a kérés állapotának változására.

```
getShoppingCartOrderId(customerId: number): Observable<number> {
    return this.http.get<number>(`http://localhost:8081/api/customer/${customerId}/shoppingcart/orderId`);
}
```

Miután elvégeztem az adatok inicializálását, a következő lépésekben meg is jelenítem azokat az Angularban használt adatkötések használatával.

A fenti példában lekérdezett adatot az „Order #**{orderId}**” egy irányú adatkötés segítségével jelenítem meg a mintában.

Többször előfordul, hogy egy adat tömböt szeretnék megjeleníteni a nézetben, erre egy példa az `ngFor*` strukturális direktíva használata. Erre példa:

```
*ngFor="let product of products"
```

Emellett ezen az oldalon a rendelés állapotához tartozó állapot jelző sáv állapotait az `ngClass` direktíva használatával változtatom meg. A mintához tartozó kód az alábbi:

```
<div [ngClass]="getProgressClass()" class="h-full text-center text-xs text-white bg-ordertracker rounded-full">
  {{statusprec}}
</div>
```

Itt a hozzáadandó CSS módosításokat a `getProgressClass()` metódus segítségével kérem le amely implemetációja az alábbi:

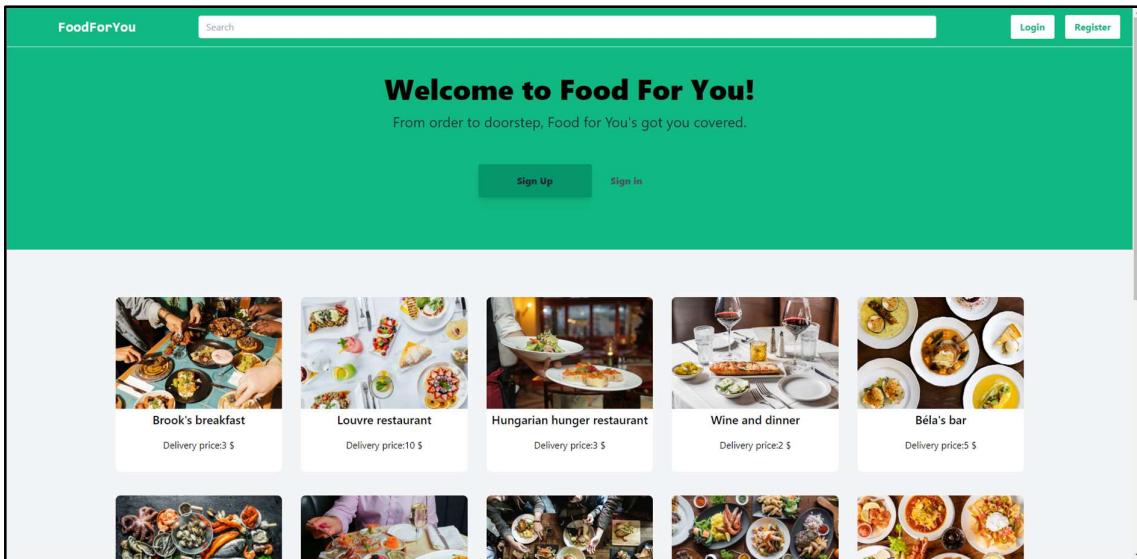
```
getProgressClass(): string {
  let widthClass = 'w-full';
  this.statusprec = "100%";
  if (this.delivery.status == 'Ordered') {
    widthClass = 'w-1/4';
    this.statusprec = "25%";
  }
  if (this.delivery.status == 'Shipped') {
    widthClass = 'w-2/4';
    this.statusprec = "50%";
  }
  if (this.delivery.status == 'On the way') {
    widthClass = 'w-3/4';
    this.statusprec = "75%";
  }
  return widthClass;
}
```

A fenti kód hatására az állapot jelző sáv állapotai feltöltés szerű tendenciát fognak mutatni a rendelés státuszának változása során.

5.3 Rendelési folyamat lebonyolítása

A rendelési folyamat lebonyolítása a főoldalról kezdődik majd a kereskedő oldalán át a rendelés összegzésén túl és a Stripe integrációban történő fizetés után a rendelés összegzése oldalon ér véget. Pontokba szedve az 5.2 ábra majd 5.6. ábrától az 5.9-es ábráig látható a teljes folyamat menete.

5.4 Főoldal



13. ábra Főoldal képernyőkép

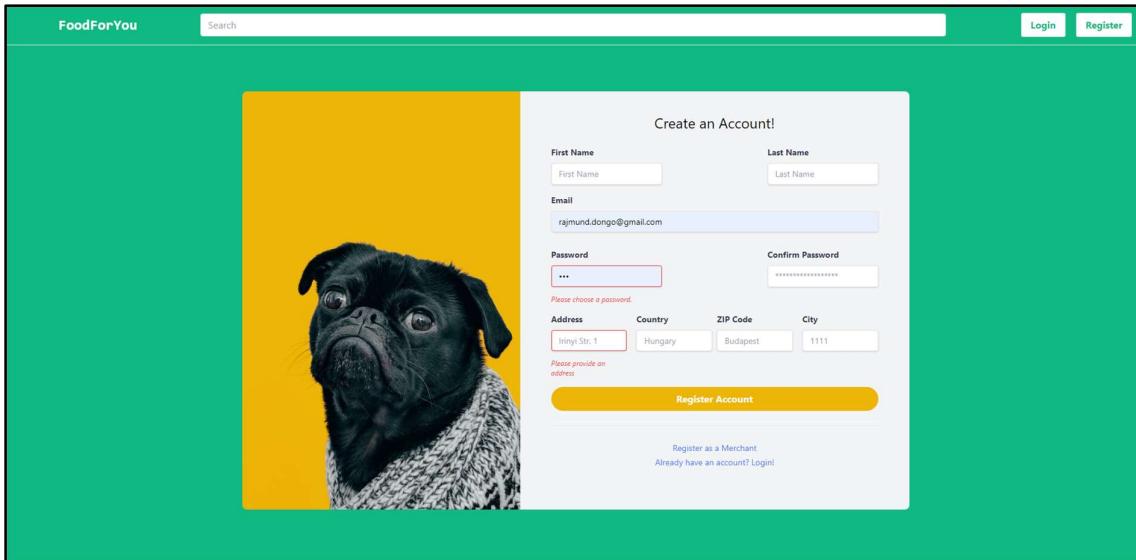
Az itt látható képernyőkép az általam elkészített főoldalt mutatja be, itt lehetősége van a felhasználónak az éttermek között válogatni, kiszállítási áraikat megnézni és átnavigálni többek között a regisztrációs és bejelentkezési nézetekre, vagy a kiválasztott kereskedő oldalára a termékek megtekintéséhez.

Ezen az oldalon megvalósítottam a betöltési animációt, amely szerint "loading..." jelenik meg az éttermek helyett ameddig nem érkezett meg az adat a másik oldalról.

Regisztráció és bejelentkezés után a vásárlókat erre az oldalra navigálja az alkalmazás automatikusan.

Ezen az oldalon az éttermek ablakszerű megjelenítéséhez a HTML mintában külső forrásból vett HTML elemeket használtam. [14]

5.5 Regisztráció vásárlóknak



14. ábra Vásárló regisztráció képernyőkép

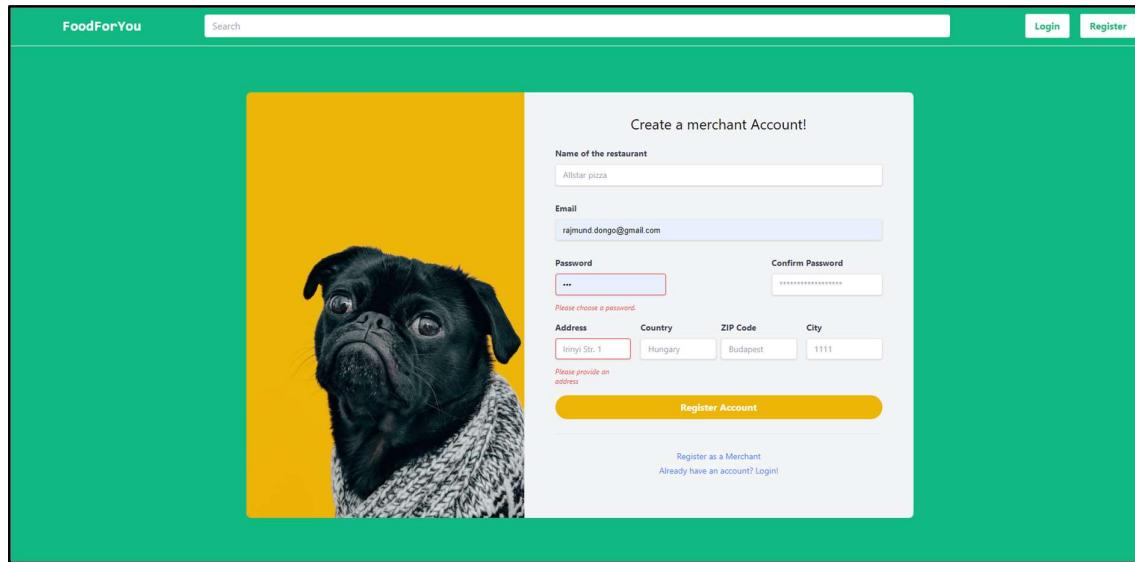
A következő ábrán látható a vásárlók számára szánt regisztrációs felület. Itt név, email, jelszó és cím megadása után a regisztráció gomb megnyomásával regisztrálhat a felhasználó egy vásárlóhoz tartozó fiókot.

Lehetőséget ad az oldal átnavigálni a kereskedői fiók regisztrálására és amennyiben a felhasználó már regisztrált átnavigálhat a bejelentkezéshez fenntartott oldalra is. Ahogyan az ábrán is kivehető, itt figyelmeztetést küld a weboldal amennyiben nem töltötte ki a felhasználó az összes szükséges mezőt.

Továbbá az egyes mezőkhöz tartozó *label* elemekben definiálásra került az elvárt értékek típusa, így például a *Google Chrome* böngésző amennyiben tárol már ilyen jellegű adatot, automatikusan kitölti a mezőket.

Ezen az oldalon a regisztrációs mezők megjelenítéséhez a HTML mintában külső forrásból vett HTML elemeket használtam. [14]

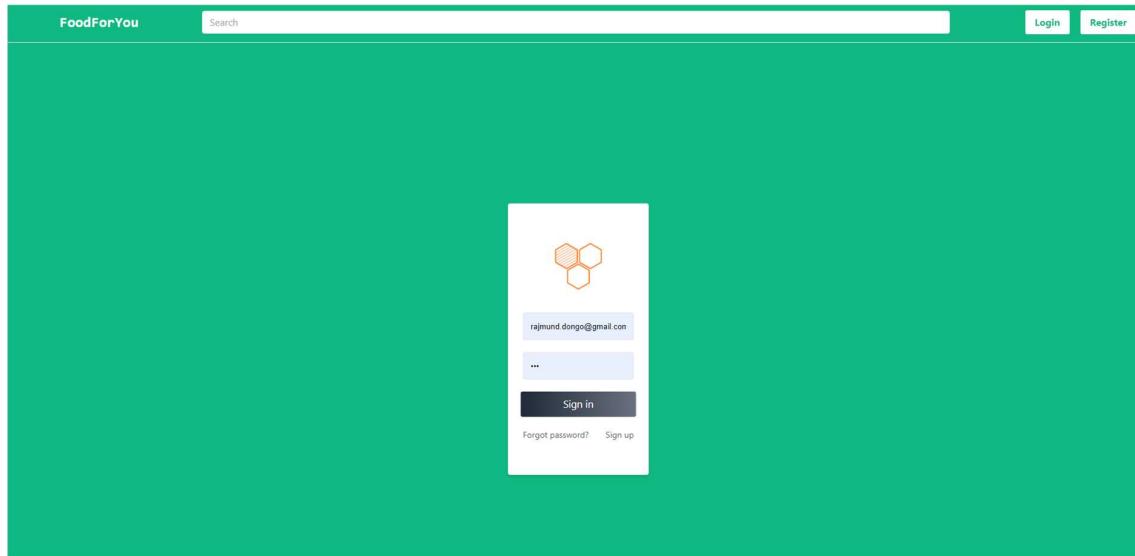
5.6 Regisztráció kereskedőknek



15. ábra Kereskedő regisztráció képernyőkép

Az alábbi oldalon a kereskedők tudnak regisztrálni a kereskedő neve, email, jelszó és lakcím megadásával. Sikeres regisztráció esetén automatikusan bejelentkeznek és át lesznek irányítva a kereskedő termék hozzáadásának oldalára, ahol termékeket tudnak hozzáadni az újonnan létrehozott oldalukhoz.

5.7 Bejelentkezés

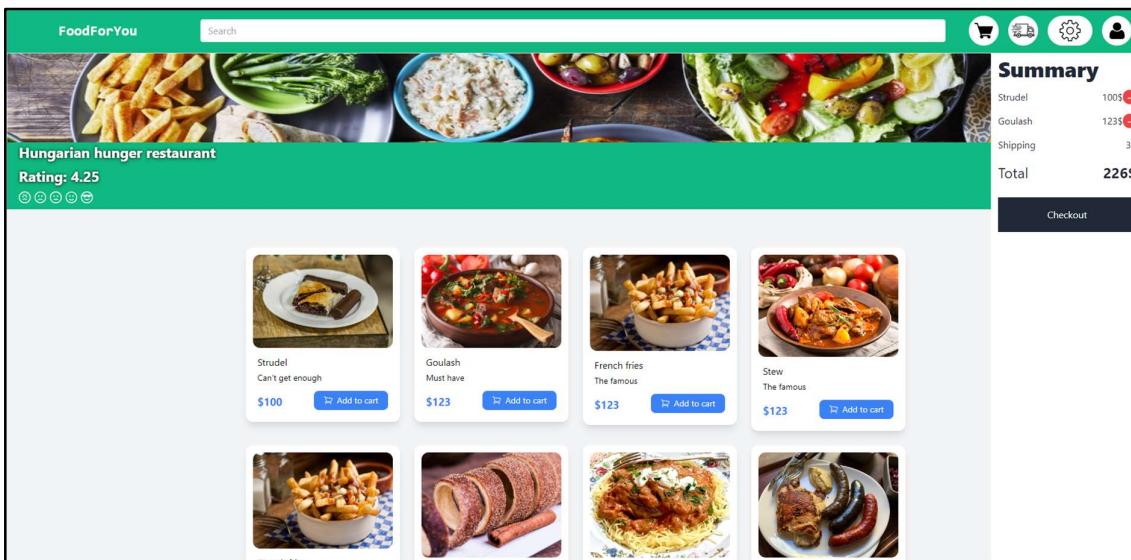


16. ábra Bejelentkezés képernyőkép

Az alábbi nézet bejelentkezés lehetőséget biztosít a felhasználó számára, ez a felület egységes bármilyen jogulságokkal rendelkező felhasználó lép be.

Itt lehetőség van átnavigálni az elfelejtett jelszó oldalra és a regisztrációs oldalra is.

5.8 Kereskedő oldalának nézete



17. ábra Kereskedő oldaláról képernyőkép

Ezen ábra a 12. ábrán megtervezett oldalnézet implementációja, amely segítségével egy vásárló megnézheti a főoldalon kiválasztott kereskedő oldalát. Itt kilistázva megtalálhatóak az adott kereskedőhöz tartozó ételek áraikkal és egy kis leírással együtt, amely tartalmazhat egy, a kereskedő által definiált rövid szöveget.

A nézeten a vásárlónak lehetősége van nem csak az étterem értékelésének (rating) megtekintésére, de annak módosítására is, szavazással. A fent látható hangulatjelek jelzik az értékelés minősítését a szomorú hangulatjelktől a vidámabb hangulatokig, amelyek egy 1 és 5 közötti skálát töltetnek ki, ahol minden boldogabb hangulat 1 értékkal magasabb értékelést jelent.

A nézet fő funkciója, hogy a vásárló számára prezenterálja a kereskedő által biztosított választékot, majd lehetőséget adjon a vásárlónak a termékek hozzáadására a kosárhoz az "Add to cart" gomb megnyomásával. A teljesség érdekében a nézet jobb oldalán megjelenő kosár panel biztosítja a már kosárba helyezett termékek gyors áttekintését, továbbá navigációs lehetőséget biztosít a rendelés összegzése oldalra amennyiben a vásárló megtalálta a számára ideális termékeket.

Ezen az oldalon a termékek ablakszerű megjelenítéséhez a HTML mintában már félkész komponenseket használtam. [14]

5.9 Rendelés összegzése

The screenshot shows the 'FoodForYou' website's checkout process. At the top, there's a navigation bar with icons for search, cart, and account. Below it, the word 'Checkout.' is displayed. The main area shows two items: 'STRUDEL' and 'BRATWURST'. Each item has a small thumbnail image, the name, and a price of '\$100.00' and '\$120.00' respectively. Below the items is a section for a discount code. It contains a text input field with 'STUDENT', a button labeled 'APPLY', and a note that says '\$223' will be deducted. Underneath this, a breakdown of the total price is shown: 'Total product price without coupons' (\$223), 'Delivery Price' (\$3), and 'Total' (\$201). At the bottom of the page is a green 'PAY NOW' button.

18. ábra Rendelés összegzése képernyőkép

A következő ábra a rendelés összegzését mutatja, amely egy utolsó áttekintést nyújt a vásárló számára a vásárlás előtt az általa kiválasztott termékekről.

Ezen az oldalon továbbá lehetőség van az esetleges kuponok megadására, amelyek egy meghatározott kedvezményt biztosítanak a felhasználónak. Jelen beállítások szerint egy kupon egy vásárláskor egyszer használható és maximálisan 90% kedvezmény érhető el.

5.10 Fizetés integráció

The screenshot shows a payment interface. On the left, there's a summary table with the following data:

Kölcsönözött termék	Ár
Strudel	90.00 USD
Bratwurst	108.00 USD
Delivery	3.00 USD
Kifizetés	201,00 USD

On the right, there's a payment form titled "Fizetés: link". It includes fields for:

- E-mail-cím
- Kártyaadatok: Card number (1234 1234 1234 1234), Exp. date (01/24), CVC
- Kártyabirtokos neve: Teljes név
- Ország vagy régió: Magyarország
- Adataim biztonságos mentése az egykattintásos fizetéshez: A checkbox is checked, with the note "Gyorsabban fizethet ezen az oldalon és bárhol, ahol a Link elfogadott." and a "Nem kötelező" button.

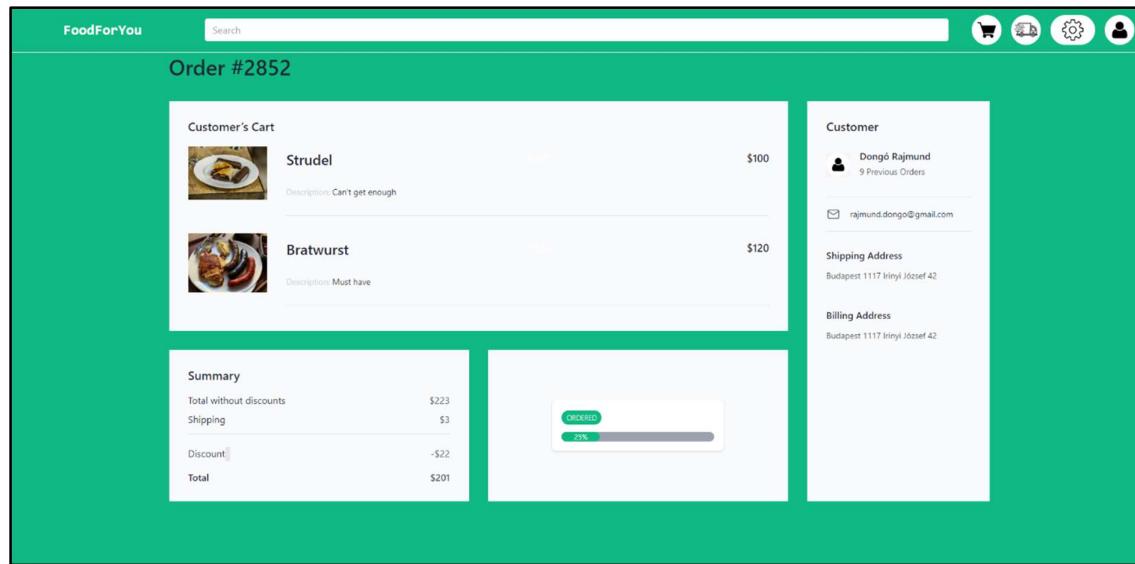
At the bottom, there's a blue "Fizetés" (Pay) button.

19. ábra Fizetés integráció képernyőkép

A fenti képernyőképet a teljesség érdekében jelenítem meg, hiszen ez az oldal nem általam készült. Egy külső szolgáltatás a *Stripe* segítségével integráltam a fizetés lehetőségét az alkalmazásba és a fent említett oldal segítségével biztosítom a lehetőséget a biztonságos fizetésre az alkalmazásban. A sikeres fizetés után a vásárló a rendelés követése oldalra lesz átirányítva.

Ezen az oldalon a kedvezmények az egyes termékekre külön vonatkoznak. Például a képeken látható „Strudel” ára 100\$ helyett 90\$ a képen, mivel a „STUDENT” kuponkód megadásra került.

5.11 Rendelés állapotának követése



20. ábra Rendelés állapotának követése képernyőkép

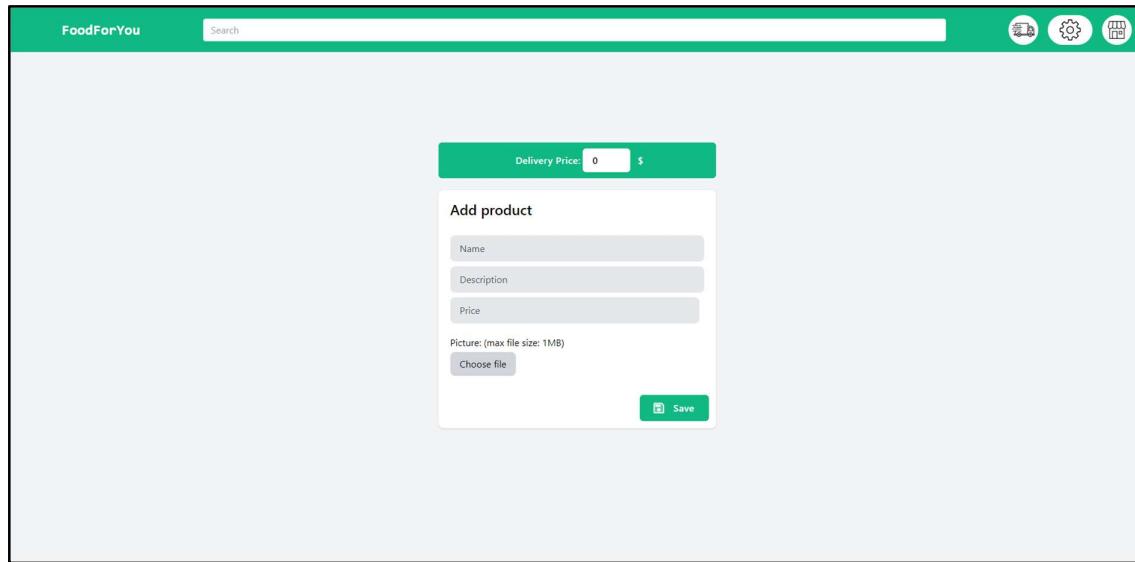
A rendelési folyamat befejeztével a vásárló a rendelés követése felhasználói nézetre érkezik meg. Itt újból áttekinthetővé válik a rendelés. Felsorolva megjelennek a rendelt termékek és megjelenik a rendelés végösszege a kuponok által biztosított kedvezmények végösszegével együtt.

Ezen a nézeten van lehetősége a vásárlónak követnie a rendelés státuszát, ez rendelés után „Ordered”, majd később változtatható a rendelés állapotától függően a kereskedő által. Továbbá megjelenik a rendelés állapotának időrend szerinti reprezentációja is.

Végül megjelenik még pár információ a rendelőről, mint például a rendelés címe, számlázási cím, email cím és az eddigi rendelések száma.

Az oldal ablakszerű megjelenítéséhez a HTML mintában külső forrásból vett HTML elemeket használtam. [14]

5.12 Termékek hozzáadása a kereskedőkhöz

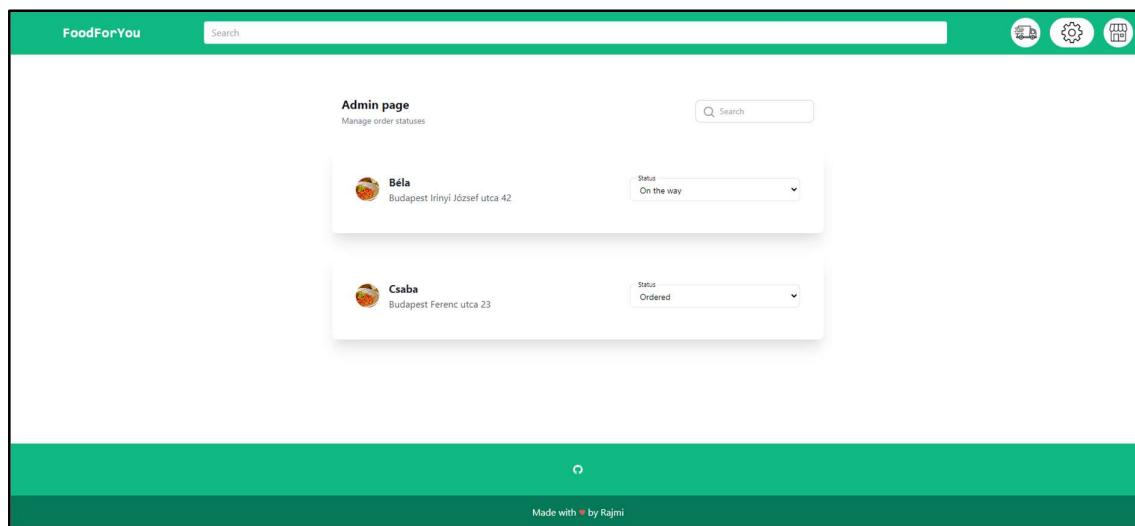


21. ábra Termék hozzáadása a kereskedőhöz képernyőkép

Az alábbi oldalra lesz irányítva minden bejelentkezett kereskedő, ezen az oldalon adható hozzá bármilyen termék a termék nevének, árának, leírásának meghatározása majd a termékhez történő kép feltöltése után.

Továbbá ezen az oldalon változatható a kereskedő kiszállítási ára is.

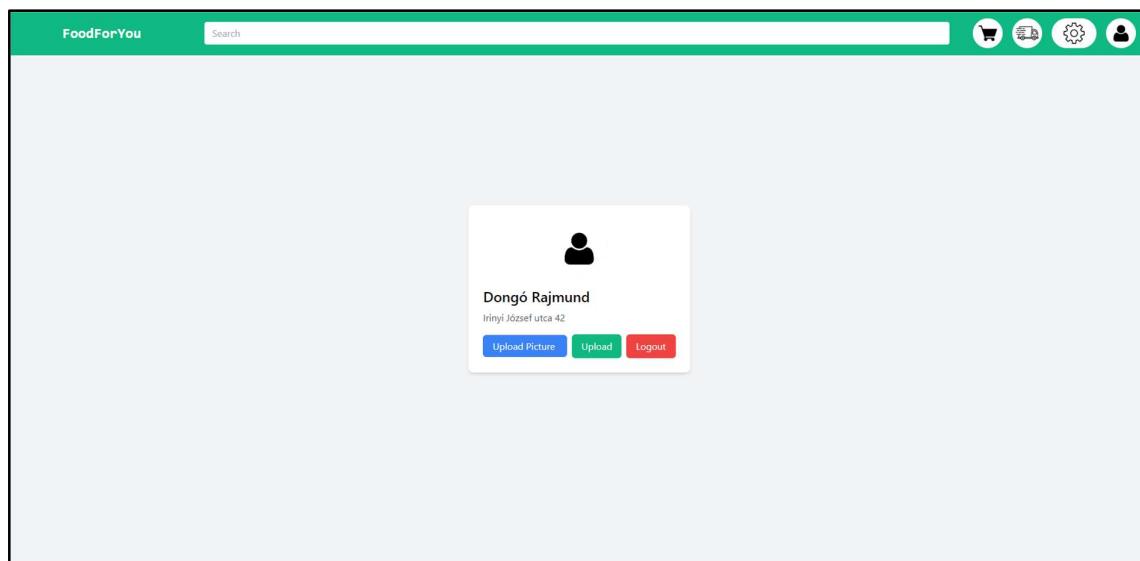
5.13 Rendelések állapotának menedzselése



22. ábra Rendelések állapotának menedzselése képernyőkép

A rendelések státuszának módosítása a kereskedők számára elérhető oldal, amely segítségével a kereskedő számára beérkezett rendelések státusza módosítható. Az oldal a fejlécben található ikonok segítségével érhető el, amely a jelenlegi képen a kis teherautó. A módosítás egy legördülő lista segítségével hajtható végre, ahol a kereskedőnek ki kell választania a rendelés új státuszát a legördülő 4 opción közül. Az opciónák az alábbiak: *Ordered*, *Shipped*, *On the way* és *Delivered*. A módosítás eredménye látható lesz a vásárló számára is a rendelés követése oldalon.

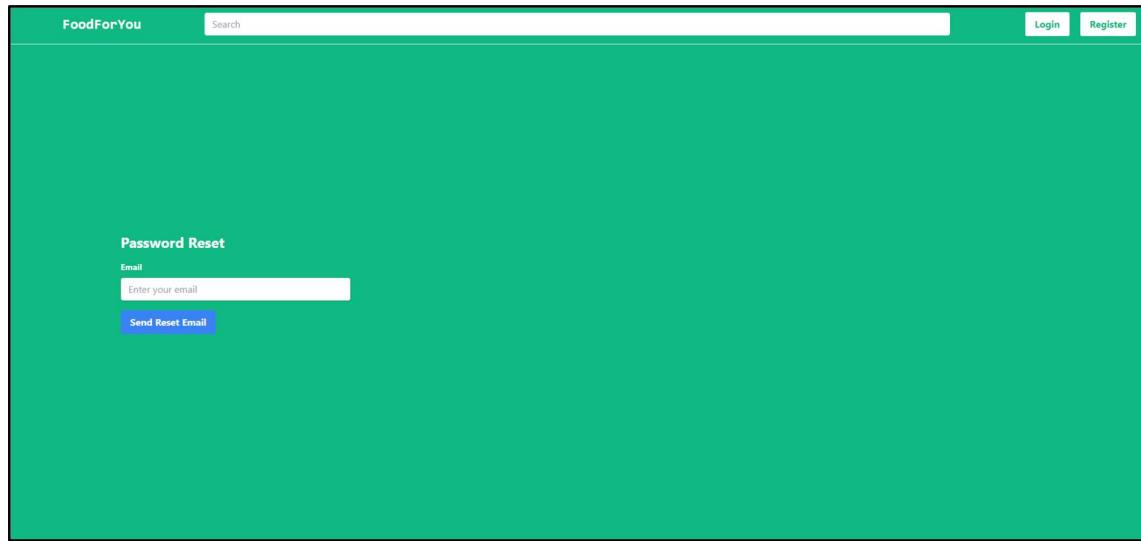
5.14 Profilkép feltöltése



23. ábra Rendelések státuszának módosítása képernyőkép

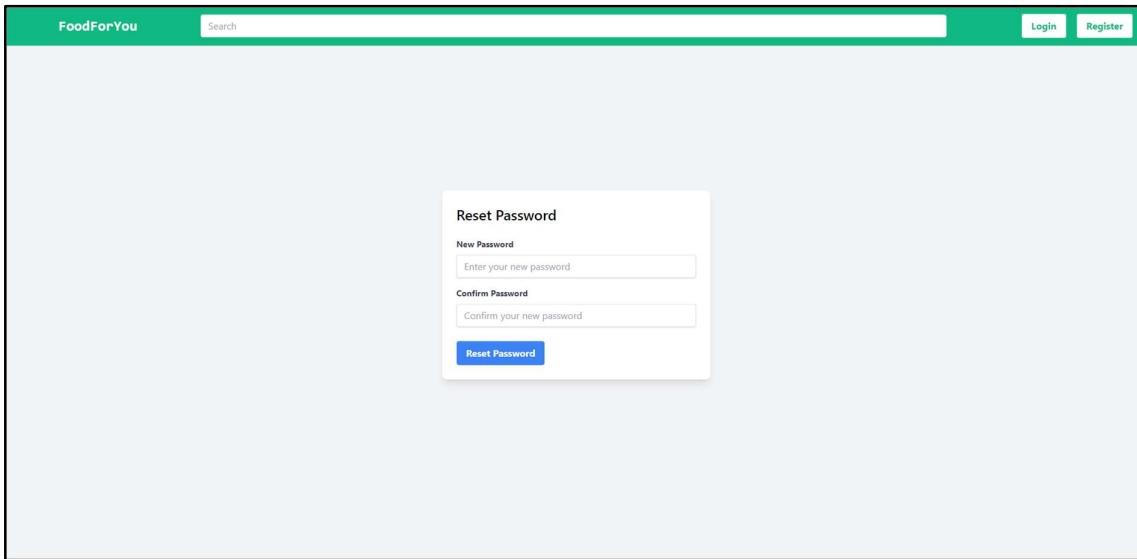
Külön nézet érhető el a profilkép feltöltésére, amely a fejlécben található beállítás gomb megnyomásával érhető el. Itt a felhasználó kiválaszthat egy képet az "Upload picture" gombra kattintva majd feltöltheti azt az upload gomb segítségével. Továbbá opcionálisan lehetőséget biztosít a kijelentkezésre is.

5.15 Elfelejtett jelszó



24. ábra Elfelejtett jelszó felhasználói nézet képernyőkép

A bejelentkezés nézeten az elfelejtett jelszóra kattintva a felhasználó átnavigálhat erre a nézetre, amely segítségével, az email cím megadása után új jelszót kérhet. Az alkalmazás ezek után a megadott email címre elküld egy linket, amely segítségével egy másik nézeten a felhasználó beállíthatja az új jelszavát. Ezek a linkek rövid ideig, 15 percig érvényesek. A jelenlegi megoldás szerint az emailben érkező URL tartalmazza a link lejáratú idejét, viszont ennek a lejáratú időnek a módosítása érvénytelenné teszi a linket, hiszen ez a lejáratú idő a backenden is ellenőrzésre kerül, így ennek a funkcionálitása mindenkor a felhasználó számára a beérkezett link érvényességének jelzése.



25. ábra Új jelszó megadása képernyőkép

5.16 Admin felület

A screenshot of the admin panel. The top navigation bar includes the "FoodForYou" logo, a search bar, and several icons for shopping cart, delivery, settings, and user profile. Below the header, the main content area is titled "Admin page" and "Manage users". It features a table with four rows. Each row contains a small profile picture, the user's name, their role (e.g., MERCHANT or USER), and a red "Delete" button. The users listed are: "Béla's bár" (MERCHANT), "Hungarian Restaurant" (MERCHANT), "Dongó Rajmund" (USER), and "Louvre restaurant" (MERCHANT).

26. ábra Admin felület képernyőkép

A fenti nézet csak admin felhasználók számára elérhető és itt képes az admin törlni az egyes felhasználói fiókokat.

Törlés hatására a fiókok adatai nem törlődnek minden össze inaktívvá válnak tehát a bejelentkezés már nem lesz lehetséges ezen fiókok számára.

5.17 Autentikáció, autorizáció a frontenden

JWT (Json Web Token) alapú autentikáció esetén fontos feladat, hogy a frontend is megfelelő fejlécikkal küldje a kéréseket az autentikációt, autorizációt végző backend számára. A JWT alapú autentikáció esetén a frontend feladata, hogy:

- minden autorizálást igénylő kérésben benne kell, hogy legyen az “Authorization” fejléc a hozzá tartozó értékkel.
- meghatározott időközönként kötelező a frontendnek úgynevezett frissítési (refresh) tokenet küldenie, amely hatására új hozzáférési (access) token és frissítési token keletkezik.
- Bejelentkezés esetén eltárolni a backendről érkező tokeneket amely segítségével később hozzáférhet az erőforrásokhoz.

Az első lépésnek bejelentkezés után az érkezett két tokenet elmentem a böngésző munkamenetéhez tartozó tárolóba.

```
setToken(token: JwtType): void {
  console.log('setToken token:', { token });
  sessionStorage.setItem('access', token.access_token);
  sessionStorage.setItem('refresh', token.refresh_token);
}
```

A következő lépésben egy *Interceptor*-t hoztam létre, amely segítségével minden kérésben elküldöm az *Authorization* fejlécben a hozzáférési tokenet. Ennek az implementációja az alábbi:

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private authService: AuthService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const authToken = this.authService.getToken();
    console.log("Intercepting token:" + authToken?.access_token)
    if (authToken && authToken.refresh_token) {
      const authRequest = request.clone({
        headers: request.headers.set('Authorization', `Bearer
${authToken.access_token}`)
      });
      return next.handle(authRequest);
    } else {
      return next.handle(request);
    }
  }
}
```

A másik feladat a tokenek időnkénti frissítése hiszen ezen hozzáférési tokenek 15 percig érvényesek. Ezért bizonyos oldalak látogatásakor automatikusan küldésre kerül egy frissítési token amelyre válaszul érkezik az újonnal létrehozott hozzáférési és frissítési token. Ezt az eredményt egy új kérés küldésével, majd a munkamenet tárhelyen tárolt adatok felülírásával érem el.

5.18 Frontend technikai megvalósításai

A következőkben a frontendben több helyen használt különböző technikai megoldásokat szeretném ismertetni.

5.18.1 Szolgáltatások és feladataik

Az alkalmazásban az alábbi szolgáltatások működtetik a frontendet:

- ***Auth interceptor service*** – Az JWT tokenek kezelését támogató szolgáltatás, feljebb már kifejtettem a működését
- ***Auth service*** – Autentikációt támogató szolgáltatás, amely a be és kijelentkezésért, tokenek megújításáért és a fiókok jogosultságának lekérdezésére felel.
- ***Customer service*** – A bejelentkezett vásárlót és a vásárló előző rendeléseinek számát kérhetjük le vele a backendtől.
- ***File upload service*** – Fájl feltöltését és lekérését szolgáló végpontokat érhetjük el ezzel a szolgáltatással.
- ***Loading service*** – A betöltés animációt megvalósító szolgáltatás.
- ***Merchant service*** – A kereskedőhöz tartozó végpontok érhetőek el vele, például a hozzá tartozó termékek, rendelések lekérése.
- ***Order service*** – Rendeléshez tartozó szolgáltatás, rendeléseket hozhatunk létre vele és itt alkalmazhatunk kupon kódokat is. Továbbá ez a szolgáltatás felel a rendelés státuszának módosításáért és a rendeléshez tartozó további kérések küldése is ezen szolgáltatás feladata.
- ***Product service*** – Termékek hozzáadását és lekérdezését vezérlő szolgáltatás.
- ***Search service*** – Live keresés megvalósításhoz szükséges szolgáltatás
- ***Shopping cart service*** – A kosár tartalmának lekérdezését és módosítását végző szolgátatás, továbbá itt van lehetőség a Stripe URL lekérdezésére is.
- ***User service*** – A felhasználó profilképének módosítása és a jelszó módosítása, elfejeztett jelszó funkciókat megvalósító szolgáltatás, továbbá ez a szolgáltatás valósítja meg a felhasználó lekérdezését és törlését is.
- ***Xhr interceptor service*** – A szolgáltatás beállítja az *X-Requested-With* header értékét.

Egy példa az Admin felhasználó lekérdezésére az *Auth service* szolgáltatásban:

```
isAdmin(): Observable<boolean> {
    console.log('Checking if user is a merchant');
    return this.http.get<User>('http://localhost:8081/api/auth/whoami').pipe(
        map((user: User) => user.role === "ADMIN"),
        catchError((error: any) => {
            console.error('Error occurred while checking if user is a merchant:', error);
            return of(false);
        })
    );
}
```

A fenti kódban elsőnek a bejelentkezett felhasználó lekérdezése történik meg a backendtől, amely a JWT token alapján visszaküldi a bejelentkezett felhasználó adatait és amennyiben Admin, true-val tér vissza.

5.18.2 Képek megjelenítése

Egy másik szintén általános megoldás a komponensek között a képek megjelenítése, amely backendről érkezik. Itt a backend oldalon egy byte tömb érkezik a frontend-re, amely feladata feldolgozni és megjeleníteni a tartalmát képként. Ezt a feladatot úgy oldottam meg, hogy a *GET request* válasz típusát “text”-re állítottam, majd a beérkezett adatot stringként kezelve áadtam az Angular DOMSanitizer *bypassSecurityTrustURL* függvényének miközben megadtam, hogy a típusa egy png kép. A megoldás így néz ki:

```
this.fileManager.downloadFile(product.imgSource).subscribe(data=>{
    product.imgDataUrl=this.sanitizer.bypassSecurityTrustUrl('data:image/png;base64
    ,'
    + data)
});
```

A fenti függvényhívás eredménye egy *SafeUrl* típusú objektum, amely könnyedén megjeleníthető a felhasználó felületen. Egy példa a megjelenítésére:

```
<img [src]="product.imgDataUrl">
```

5.18.3 Betöltési animáció

Bizonyos esetekben sokáig tarthat ameddig a backend elküld minden szükséges adatot egy-egy komponens vagy nézet megjelenítéséhez, ez idő alatt pedig az alkalmazás üresen jelenik meg. Erre egy példa a főoldal kereskedői az alkalmazásban, hiszen ekkor nem csak a kereskedők adataira, de a kereskedőkhöz tartozó képekre is szükség van.

A fenti problémára találták ki a betöltési animációt, amely segítségével ideiglenesen jeleníthetünk meg tartalmat, ameddig nem érkeznek meg a megjelenítéshez szükséges adatok.

A megoldásomban egy külön szolgáltatást (service) hoztam létre, amely segítségével irányíthatom az egyes oldalak betöltését. A betöltés szolgáltatás (loading service) rendelkezik 2 attribútummal és 2 metódussal. A metódusok segítségével jelezhetjük, hogy megérkezett a szükséges adat és ennek megfelelően változik a kettő attribútum. Két attribútumra azért van szükség mert az egyik egy *BehaviourSubject<boolean>* típusú változó, amely értéke a *next()* metódussal állítható és a másik attribútum pedig az előző attribútum megfigyelhetővé (Observable) alakítása.

Az előbb említett függvényeket pedig úgy alkalmazom, hogy mikor a nézet megnyílik akkor a szolgáltatáshoz tartozó attribútum értéke *false*, majd minden sikeres kéréssel növelek egy számlálót és mikor a sikeres kérések száma eléri az összes kérés számát, akkor a szolgáltatás függvényét hívva megváltoztatom a hozzá tartozó attribútum *true*-ra, amely a nézeten az adatok megjelenítését eredményezi.

A feltételesen megjelenítendő adatok megjelenítését pedig **ngIf* strukturális direktíva használatával feltételhez kötöm. Erre egy példa:

```
*ngIf="(loadingService.isLoading$ | async) === false"
```

6 Backend megvalósítása

6.1 Általam választott technológiák

A feladatkiírásom habár meghatározta a backend és frontendhez használt keretrendszerét, még szükséges volt eldönteni például, hogy szeretnék-e plusz eszközöket használni a fejlesztéshez.

Egy másik jelentősebb hatással járó döntés volt, hogy a Spring mellé egy build tool-t kellett választanom. Itt két opciót fontoltam meg, az egyik a *maven* a másik pedig a *gradle*. Mindegyik eszköz rendelkezik a szükséges funkcionálisokkal, minden össze a szintaxis más a kettő között. A *gradle*-ben *groovy* ameddig *maven*-ben Extensible Markup Language (XML) nyelven íródnak az utasítások. Végül a *maven* mellett döntöttem mert kevesebb tapasztalattal rendelkezem a technológiában és hasznosnak találtam az eszköz mélyebb elsajátítását.

6.2 Order tracker

Az első és egyben legnagyobb szolgáltatás az alkalmazásban az Order Tracker nevű szolgáltatás. Feladata a teljes frontend kiszolgálása, amely néhol a másik kettő mikro szolgáltatás bevonásával valósul meg.

6.2.1 Adatbázis, entitások

A backendhez tartozó első feladat az adatbázis létrehozása, majd az ehhez történő kapcsolat kialakítása.

Adatbázisnak *PostgreSQL* adatbázist választottam könnyen kezelhetősége és jól dokumentáltsága miatt. Az alkalmazást egy Java Persistence API (JPA) implementáció, a Hibernate segítségével kapcsolom az adatbázishoz, amely képes generálni az adatbázishoz szükséges táblákat az entitásokban megadott annotációknak köszönhetően. Az alkalmazás megfelelő működéséhez létrehoztam egy ordertracker nevű adatbázist és sémát, valamint megadtam a szükséges hitelesítési adatokat a kapcsolódáshoz az application.properties fájlban.

A teljes adatbázishoz kapcsolódó *yml* kód így néz ki:

```
server.port=8081
Spring.jpa.show-sql=true
Spring.jpa.properties.hibernate.format_sql=true
Spring.jpa.hibernate.ddl-auto=update
Spring.jpa.properties.hibernate.dialect
org.hibernate.dialect.PostgreSQLDialect =
```

```
Spring.datasource.url=jdbc:postgresql://localhost:5432/ordertracker
Spring.datasource.username=postgres
Spring.datasource.password=postgres
Spring.datasource.driver-class-name=org.postgresql.Driver
```

Az adatbázishoz kapcsolódás után, az entitásokról szeretnénk beszélni. Az entitás nem más mint, olyan osztályok összessége, amelyeket perzisztálni kívánunk az adatbázisba valamilyen objektum relációs leképezést alkalmazó technológia segítségével. Az entitásokat az `@Entity` annotáció jelöli.

Az entityk közötti kapcsolatokat különböző annotációkkal tudjuk leírni, például vegyük az alábbi kódot:

```
@ManyToMany
@JsonIgnore
@JoinColumn(name = "product_id")
private List products;
```

Itt a `@ManyToMany` annotáció megadja, hogy az osztály több-több kapcsolatban áll a *Product* entitással és `@JsonIgnore` annotációval megadtam, hogy ne szerializálja az alkalmazás mikor megpróbálja az objektumot Json formátumba alakítani, ezt azért tettem, mert ellenkező esetben Json szerializációs hiba lépne fel, ha minden oldalon hagyjuk megjeleníteni a másik oldal objektumait, akkor végtelen rekurzív ciklusba fullad az alkalmazásunk és végül hibaüzenetet fogunk kapni.

Egy alternatív megoldás, ha megadjuk, hogy csak a másik objektum ID-ját jelenítsük meg. Az előbbi így lehet megtenni:

```
@JsonIdentityInfo(generator      = ObjectIdGenerators.PropertyGenerator.class,
property = "id")
```

A fenti, kapcsolatok létrehozását leíró példát folytatva a `@JoinColumn` annotációval megadtam, hogy mely oszlopokat szeretném összekötni. Továbbá szeretném megjegyezni, hogy a `@Column` annotáció használatakor, a JPA minden attribútum nevét camel case-ről snake case-re alakítja át és azt az értéket veszi alapul amikor keresi a táblában az oszlop nevét, ezért a lenti esetben például a `@Column` annotáció el is hagyható:

```
@Column(name = "order_date")
LocalDateTime orderDate;
```

6.2.2 Repository

Miután definiáltam az entitásokat, szükségem volt egy megoldásra, amely segítségével el tudom érni az adatbázisban tárolt adatokat és menedzselni tudom őket. Erre tökéletes megoldás a Springben használható adattároló, (repository) amely segítségével rendkívül könnyen érhetünk el adatokat a beépített függvények segítségével, mint például a *findAll()* és kevés kód írásával új függvényeket hozhatunk létre.

Minden entitáshoz létrehoztam egy hozzá tartozó adattárat, amely segítségével manipulálni tudom az entitásokat az adatbázisban.

Egy adattár minden adattárat bővíti ki, amely legtöbbször a CrudRepository vagy JpaRepository. Az adattárakat a *@Repository* annotációval láttam el, ezzel jelezve a Spring számára, hogy adattárként kezelje a Beant.

Amennyiben nincsen a számunkra szükséges metódus alapból definiálva az adattárban, és az adattár metódus nevével leírható az óhajtott metódus funkcionalitása, akkor elég megadnunk a metódus nevét és a Spring elkészíti hozzá az implementációt. Példa egy *Token* objektum lekérdezésére a String típusú *token* tulajdonság alapján:

```
Optional<Token> findByToken(String token);
```

Továbbá megadhatunk saját magunk által definiált sql lekérdezést is:

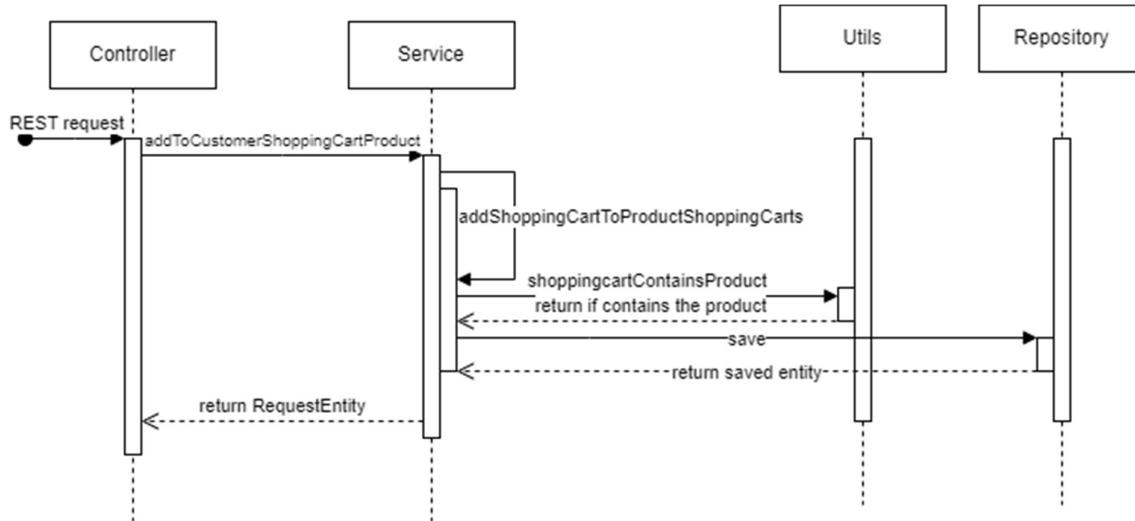
```
@Query("select t from Token t where id = :1")
List<Token> findTokensWhereIdIsOne();
```

6.2.3 Kérések kiszolgálása

A backenden definiált osztályok különböző csomagokban vannak definiálva, ilyen például a fent említett entitás (entity) és adattár (repository). Ezen csomagok együttműködése biztosítja az alkalmazás működését és segítségükkel jól elkülöníthető az egyes osztályok feladatának jellege is.

Az alkalmazás legtöbbször POST és GET kéréseket szolgál ki, POST requestet használok a backenden történő mentésre, módosításra GET kérést pedig adatok leképezésére.

A következőkben bemutatom hogyan néz ki egy kérés kiszolgálása a backenden megírt csomagok segítségével. A bemutatásra a termék hozzáadását választottam a vásárló kosarához.



27. ábra Termék mentése a vásárló kosarába

A 27.ábrán látható, hogy milyen módon hívják egymás metódusait a csomagokban lévő osztályok. A fenti példában elsőnek a kontroller (controller) fogadja a beérkező Representational state transfer (REST) kérést majd a kontroller meghívja az *addCustomerShoppingCartProduct* metódust a hozzá tartozó szolgáltatás osztályban, amely feladata az üzleti logika, azaz a hozzáadás megvalósítása.

```

@PostMapping("/customer/{id}/shoppingcart/product")
public ResponseEntity<HttpStatus>
addCustomerShoppingCartProduct(@PathVariable("id") Long id, @RequestBody
Product product)
{return customerService.addCustomerShoppingCartProduct(product, id);}

```

A kontrollerben megírt kód az alábbi módon néz ki, itt *@PostMapping* annotációval definiálom, hogy milyen elérési útvonalon várja az alkalmazás a kérést és a *@PathVariable* annotációval jelölöm a Spring számára, hogy az elérési útvonalban az *{id}* helyére megadott értéket a kérésben *Long* típusú váltózóként kezelje. A *@RequestBody* annotáció segítségével megadtam, hogy a kérés adatcsomagját alakítsa át a Spring *Product* objektummá.

A szolgáltatáson belül az olvashatóság érdekében a kód egy részét kiszerveztem egy külön privát metódusba az osztályon belül, amelyet *addShoppingCartToProductShoppingCarts*-nak hívnak és magát a hozzáadást végzi az ellenőrzések után. Mivel többször előfordul a kódban a szükség a kosár tartalmának ellenőrzésére, főleg arra, hogy egy termék megtalálható-e benne, ezért ezt egy külön *utils* csomagban található osztályban valósítottam meg, amely szerepe, hogy a programban több osztály által felhasznált funkciókat valósítsanak meg.

Miután megtörténnek a változtatások, már csak a perzisztálás maradt hátra, ezt pedig a már fentebb megismert adattár azaz Repository segítségével teszem meg. minden ilyen adattár

rendelkezik egy `save()` metódussal, amely segítségével az adattár lementi a hozzá tartozó entitást és visszaadja a mentett adatot.

6.2.4 Rendelés a backenden

A frontend oldalról bemutattam már hogyan is néz ki a rendelés folyamata a vásárló oldaláról. Viszont ennek az alapjait a backenden fektettem le. Itt különböző végpontok segítségével biztosítom a vásárló számára a rendelés lehetőségét.

Sorban az alábbi oldalakon a fájl letöltést leszámítva az alábbi végpontok hívódnak meg abban az esetben, ha a felhasználó csak alapműveleteket végez:

1. Kereskedő oldala
 1. GET „`api/auth/merchant/{id}/products`” Lekérdezem a kereskedőhöz tartozó termékeket
 2. GET „`api/customer/{id}/shoppingcart/products`” Lekérdezem a vásárló kosarában lévő termékeket
 3. GET „`api/merchant/{id}`” Lekérem a kereskedő adatait
 4. POST „`api/customer/{id}/shoppingcart/product`” Hozzáadom a terméket a vásárló kosarához.
 5. DELETE „`api/customer/{id}/shoppingcart/product/{productid}`” Törlöm vele a terméket a vásárló kosárából.
2. Rendelés összegzése oldal
 1. GET „`api/customer/{id}/shoppingcart/products`” Lekérdezem a vásárló kosárában lévő termékeket
 2. POST „`api/order/customer/{cid}/merchant/{mid}`” Létrehozom a rendelést a felhasználóhoz
3. Stripe integráció
 1. Itt a payment service-ben bemutatott hívások történnek meg
4. Rendelés összegzése oldal
 1. GET „`api/customer/{id}/shoppingcart/orderId`” Segítségével lekérdezem a rendeléshez tartozó azonosítót

2. GET „*api/customer/{id}/previousorders*” Lekérdezem az előző rendelések számát
3. GET „*api/order/{id}*” Lekérdezem a rendelés adatait és megjelenítem a felhasználó felületen
4. GET „*api/order/{id}/delivery*” Lekérdezem a kiszállítás adatait és megjelenítem a felhasználó felületen

6.2.5 Regisztráció, bejelentkezés

Az alkalmazásban megvalósuló bejelentkezés és regisztrálás lehetőségének megvalósítása a backend oldalon történik az order tracker szolgáltatásban. Ez a funkció manapság már minden modern alkalmazás elengedhetetlen része.

Viszont elsőnek a sikeres kérések küldésének érdekében át kellett állítanom az alkalmazásban a Cross-origin resource sharing-ot (CORS) amely lényege, hogy az alkalmazást megvédje a Cross Site Request Forgery-től (CSRF) amely egy népszerű kibertámadási módszer, amely segítségével a támadó eléri, hogy a felhasználó nem szándékos műveleteket hajtson létre a már bejelentkezett fiókjában. A CORS meghatározza, hogy milyen eredetű csomagokat fogadjon az alkalmazás a saját internetes tartományán (domain) kívül. Az alábbi kód segítségével felüldefiniáltam a corsfiltert, amely feladata CORS kérések szűrése. Fontos megjegyezni, hogy az alábbi implementáció célja a fejlesztés megkönnyítése, mivel valós környezetben azonos eredettel (Origin) rendelkezik a front és backend, ezért az alábbi kódot el kell távolítani, vagy egy kapcsoló implementálása is megoldás lehet.

```
@Bean
public CorsFilter corsFilter() {
    UrlBasedCorsConfigurationSource source = new
    UrlBasedCorsConfigurationSource();
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowCredentials(true);
    config.addAllowedOriginPattern("*");
    config.addAllowedHeader("*");
    config.addAllowedMethod("*");
    source.registerCorsConfiguration("/**", config);
    config.addExposedHeader("Access-Control-Allow-Origin");

    return new CorsFilter(source);
}
```

A fenti kódban látható, hogy bármilyen eredetű minta megengedett bármilyen headerrel és bármilyen http módszert alkalmazva.

Ahogyan említettem már, az autentikáció a megoldásához JWT alapú autentikációt választottam, amely egy iparbeli standard RFC 7591 megoldás [15]. Ennek a megoldásnak a lényege az, hogy bejelentkezéskor keletkezik kettő token, egy hozzáférési (access) és egy frissítési (refresh) token, amelyek szükségesek lesznek a sikeres kérés küldéséhez.

A JWT tokenek 3 részből állnak:

- Fejrész: Megadja a token típusát és a titkosításhoz használt algoritmust, amely nálam Hmac Sha256
- Törzs rész, azaz adatcsomagot tartalmazó rész: Ez állításokat, úgy nevezett “claim”-eket tartalmaz általában a felhasználóról, akihez a token tartozik. Nálam ezekre példa a hozzárendelt felhasználó neve, lejárat dátuma és a kiállítás dátuma. Ez a része BASE64URL kódolással van ellátva.
- Aláírás: A token utolsó része, az előző részeket írja alá a titkosított kulccsal, amelyet az alkalmazás ismer csak. A felépítése az alábbi:

```
HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

Első lépésnek létrehoztam a JWT tokenek létrehozásához szükséges függvényeket a *JwtService* nevű java osztályban, ennek az osztálynak a létrehozásához egy már létező github projektből vettet segítséget [16]. Itt találhatóak a JWT tokenek létrehozására, Claimek kinyerésére és a lejárat ellenőrzésére készült függvények.

A JWT függvények könnyebb kezelésére az *io.jsonwebtoken* csoport alá tartozó csomagokból importáltam függőségeket a *maven* segítségével. Ezek nagyban megkönnyítették a tokenek kezelését.

Miután létrehoztam a tokenek kezelésének alapjait, meg kellett valósítanom a regisztráció és bejelentkezéshez használt függvényeket. Ezek implementációjára az *AuthenticationService* osztályban került sor.

A bejelentkezés megvalósítása az alábbi kóddal történik:

```

public AuthenticationResponse login(AuthenticationRequest request) {
    authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(
            request.getEmail(),
            request.getPassword()
        )
    );
    User user = repository.findByEmail(request.getEmail())
        .orElseThrow();
    String jwtToken = jwtService.createAccessToken(user);
    String refreshToken = jwtService.createRefreshToken(user);
    revokeAllTokensForUser(user);
    saveTokensForUser(user, jwtToken);
    return AuthenticationResponse.builder()
        .accessToken(jwtToken)
        .refreshToken(refreshToken)
        .build();
}

```

A fenti példakódban a felhasználói bejelentkezést valósítottam meg. Itt elsőnek a *AppConfig*-ban definiált *AuthenticationManager* bean segítségével autentikálom a felhasználót, amely kivételt fog dobni amennyiben helytelen hitelesítési adatokkal hívom meg a függvényt. Utána az adatbázisban tárolt felhasználói adatok segítségével új hozzáférési és frissítési tokeneket generálok majd visszavonom a felhasználó számára kiadott régi tokeneket és elmentem az új tokent majd visszaadom az új tokeneket amelyet a kontroller az *AuthenticationResponse* domain osztály segítségével továbbít majd a frontend számára.

A regisztráció és a bejelentkezés elkészítése után a következő feladat a kérések szűrése és az illetéktelen kérések visszautasításának feladata. Ez a *SecurityFilterChain* felüldefiniálásával értem el. Az alkalmazáshoz használt *SecurityFilterChain* az alábbi módon néz ki:

```

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
    http
        .addFilterBefore(corsFilter(), CorsFilter.class)
        .csrf().disable()
        .authorizeHttpRequests()
        .requestMatchers("/api/auth/**").permitAll()
        .anyRequest()
        .authenticated().and().sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .authenticationProvider(authenticationProvider)
        .addFilterBefore(jwtAuthFilter,
            UsernamePasswordAuthenticationFilter.class)
        .logout()
        .logoutUrl("/api/v1/auth/logout")
        .addLogoutHandler(logoutHandler)
        .logoutSuccessHandler((request, response, authentication) ->
    SecurityContextHolder.clearContext());
    return http.build();
}

```

A fenti beant elemezve látható, hogy a feljebb bemutatott *CorsFiltert* hozzáadom a szűrésláncot majd kikapcsolom a CSRF védelmet. Megadom, hogy autorizáció lesz szükséges a http kérések fogadásakor, viszont a „/api/auth/**” reguláris kifejezésre illeszkedő kéréseket engedélyezem ellenőrzés nélkül. A *sessionCreationPolicy* definiálásával meghatározom, hogy a Spring ne hozzon létre munkameneteket. Ezt követően megadtam az alkalmazáshoz tartozó *AuthenticationProvider-t* és az általam elkészített JWT authentikációs filtert. Továbbá definiáltam egy kijelentkezéshez használt urlt és megadtam, hogy sikeres kijelentkezés esetén törlődjön a tárolt *SecurityContext*.

6.3 Payment service

Ez a mikroszolgáltatás a Stripe-al történő integrációért felel. A Stripe segítségével megvalósított fizetésért felel az alkalmazásban.

A szolgáltatás minden össze egy darab Application Programming Interface-el (API) rendelkezik, amely a termékek listáját és egy valutát kér majd egy URL-t (Uniform Resource Locator) ad vissza, amelyet megnyitva elérhetővé válik a Stripe oldala, amelyen a megadott termékek kifizetése elvégzhető.

6.3.1 Stripe integráció

A Stripe lehetőséget ad a programozóknak, hogy egyszerűen integrálhassanak fizetési opciókat az alkalmazásba. Rengeteg endpoint elérhető, amely segítségével személyre szabható a vásárlók számára tervezett fizetési folyamat, viszont én ezek körül csak párat használtam az alkalmazásban.

A Stripe lehetőséget ad teszt környezet alkalmazására, amelyhez nem kell mást tenni minden össze teszt API kulcsot kell alkalmazni a kérés küldésekor.

Az eredmény eléréséhez a szolgáltatásnak 3 kérést kell küldenie a Stripe számára, hogy végül visszakapja a szükséges URL-t:

- Az első kérésben a Stripe számára elküldöm a termékeket, amelyeket később ki szeretnék fiztetni a vásárlóval. Ezt minden egyes termékre meg kell tenni külön a listában. minden egyes termékre, amit elküldök a Stripe ad vissza egy választ, amely tartalmazza a termék adatait, amelyek között ott van a stripe által létrehozott azonosító a termékhez.

- A második kérés során beárazom a terméket, az adatcsomagnak ebben a kérésben tartalmaznia kell az első kérés során küldött azonosítót és mellette az árat és a valuta nevét a *unit_amount* és *currency* mezőkben.
- A harmadik kérésben a Stripe által biztosított módon sorolom fel a termékeket a rendelési mennyiséggel együtt és mivel elvárt, hogy a sikeres fizetés befejeztével átirányítás történjen, ezért meg kell adni az *after_completion[type]* és *after_completion[redirect][url]* mezőket is.

A harmadik kérésre egy minta az alábbi módon néz ki:

```
line_items[0][quantity]:1
line_items[0][price]:price_1NoniVLPnGf2wGgP9hLzQftD
after_completion[type]:redirect
after_completion[redirect][url]:http://localhost:4200/
```

Ahogyan az a fenti kérésben is látszik, nem Json hanem x-www-form-urlencoded formátumban szükséges a kérés adatcsomagját küldeni.

6.3.2 Swagger

Minden alkalmazás alapköve a jól dokumentáltság és hogy az alkalmazás a leírtaknak megfelelően működjön. Erre a célra tökéletes megoldásnak találtam a Open API használatát, amely segítségével az alkalmazás meg tud felelni az előbb említett követelményeknek. Az open api megoldása az, hogy az alkalmazás dokumentációjából generálja a kódot, amely az alkalmazásban fut, ezzel szó szerint a dokumentáció lesz a megvalósítás definíciója.

Az implementáció kettő jelentős lépésből áll. Elsőnek létre hoztam egy *yaml* fájlt, amelyben az openapi konvenciónak megfelelően definiáltam a számomra szükséges model fájlokat. Itt számos lehetősége van a programozónak, megadható leírás, példa vagy akármilyen metaadat az egyes objektumokról, ezzel megkönnyítve a dokumentáció olvasása során az olvasó számára a szöveg megértését.

Az megvalósítás következő lépése a *maven* számára megadni, hogy a projekt építése (build) során az openapi generátort használja az alkalmazás az új fájlok létrehozására.

```

<plugin>
    <groupId>org.openapitools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <version>4.2.3</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <inputSpec>${project.basedir}/src/main/resources/openapi.yaml</inputSpec>
                <generatorName>java</generatorName>
                <configOptions>
                    <additionalModelTypeAnnotations>@lombok.Data
                    @lombok.NoArgsConstructor      @lombok.AllArgsConstructor      @lombok.Getter
                    @lombok.Setter</additionalModelTypeAnnotations>
                </configOptions>
                <generateApis>false</generateApis>
            </configuration>
            <generateSupportingFiles>false</generateSupportingFiles>
            <generateApiDocumentation>false</generateApiDocumentation>
        </execution>
    </executions>
</plugin>

```

A fenti példakódban a látható ahogyan definiálom azt a *maven* plugint, amely segítségével az osztályok létre fognak jönni. Itt a függőség elérhetőségei után megadom, hogy milyen műveleteket végezzen el, amikor az alkalmazás felépül. Elsőnek megadom, hogy a művelet célja generálás lesz és megadom az elérési útvonalat a yaml fájlhoz, amelyben a generálandó osztályok leírása található. Mivel szükségem van pár osztály szintű annotációra is ezért ezeket az *<additionalModelTypeAnnotations>*-ben definiálom.

A fenti kód ellenőrzése a *maven clean install* parancs kiadásával tehető meg, amely újra fogja építeni az alkalmazást ezzel együtt elkészítve az új osztályokat.

6.4 File upload service

A file upload azaz fájl feltöltés szolgáltatás célja, hogy az alkalmazásban tárolandó képeket kezelje és egy egységes API-t biztosítson az alkalmazásoknak.

Ebben a szolgáltatásban nem a megszokott módon, id szerint, hanem név szerint kérem le az egyes képeket. Egy feltöltött fájlról az adatbázisban eltárolom a nevét, típusát, egy azonosítót hozzá és magát az adatot, amely egy bájt tömb *@Lob* annotációval, amely azt jelöli, hogy ez egy nagy objektum (Large Object).

6.5 Docker

Az alkalmazás futtatását szerettem volna egységesen könnyűvé tenni minden környezeten, ezért dockerizáltam az alkalmazást a következőkben leírt módszerrel.

Létrehoztam különböző docker fájlokat a frontendnek és a három darab backend szolgáltatásnak, amelyek segítségével a szolgáltatásokat külön, külön dockerizálni lehet. A docker fájlok összekapsolására létrehoztam továbbá egy docker compose fájlt, amelyben megadtam, hogy az előbb létrehozott docker fájlokat használjam az egyes komponensek dockerizálására.

```
postgres:
  image: postgres:13-alpine
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
  ports:
    - "5432:5432"
  command: ["postgres", "-c", "max_connections=300", "-c",
    "shared_buffers=512MB"]
  volumes:
    - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

A fenti kód segítségével a docker-compose fájlon belül létrehozom az adatbázist, amelynek a postgres nevet adom és alapjául a postgres:13-alpine docker képet(docker image) adom meg.

A következő sorokban meghatározom az adatbás eléréséhez szükséges hitelesítési adatokat, amelyre a többi docker alkalmazásnak szüksége lesz. A rákötő sorban pedig megadom a portokat amelyeken az alkalmazás elérhető lesz, majd az indítási paramétereket a command sorban definíálom. Végül pedig az inicializáláshoz használt sql fájl elérési útvonalát adom meg, amely létrehozza az általam használt adatbázis sémákat.

Az adatbázis definiálása után egyesével definiáltam a tulajdonságokat az egyes szolgáltatások docker fájljainak összekapsolásához.

```
app:
  build: .
  ports:
    - "8081:8081"
  depends_on:
    - postgres
```

Itt az order tracker alkalmazást integrálom a docker komponálásba. Elsőnek megadom a szolgáltatáshoz tartozó dokcer fájl elérési útvonalát, mivel egy mappában van a compose fájal

ezért egy pontot adtam meg. A *depends_on* tulajdonság segítségével pedig megadtam, hogy futtatás előtt elsőnek feljebb létrehozott postgres adatbázist futtassa az alkalmazás.

7 Tesztelés

Minden megírt szoftver számára fontos szempont a tesztelés hiszen ezzel validáljuk az elkészült funkciók helyességét és visszajelzést ad arról, hogy jól dolgoztunk. Emellett a segíthet a későbbi módosítások során ellenőrizni, hogy az alkalmazásban megírt függvények megfelelően működjenek és véletlenül ne módosítsuk a funkcionalitásokat.

7.1 Unit tesztek

Az alkalmazás tesztelése során az üzleti logika tesztelésére tértem ki jelentősen, ezért a tesztjeimet a backenden készítettem el egység tesztek formájában. Az egység (Unit) tesztelés lényege, hogy minden egy funkciót tesztelek és nem a funkciók összességét. Jellemzően egy ideálisan megírt alkalmazás könnyedén egység tesztelhető hiszen minden metódus jól definiálható külön feladatot lát el.

Teszteléshez az elterjedt JUnit-ot használtam, amely segítségével a backend üzleti logikájának szinte minden metódusát leteszteltem. Ennél nagyobb teszt lefedettséget akkor tudtam volna elérni, ha integrációs teszteket is alkalmazom, viszont ezek minden controller hívást tudták volna ellenőrizni a teszteletlen funkciók közül, ezért úgy döntöttem, hogy ezt nem implementálom. Természetesen ez is egy bővítési lehetőség az alkalmazás számára.

Az egység tesztek megírásakor sokszor kellett mock azaz látszat osztályokat létrehoznom, amelyek segítségével a tesztek elkerülhetik a valós adatbázis módosítását. A *Mockito* nevű teszteléshez használt keretrendszert alkalmaztam a feladatra, amely széles eszköztárának köszönhetően rendkívül elterjedt a fejlesztők körében.

A tesztek írásakor a *Mockito* technológiát használva több különböző feladattal kellett szembe néznem, a fentebb említett adatbázis módosítások elkerülése mellett sokszor az adatbázisból való kiolvasás is része volt a tesztelendő kódrészleteknek, ilyenkor lehetőségem volt a *when().thenReturn()* metódusok használatára, amely a *when()* metódusban megadott függvényhívás eredményét a *thenReturn()* paraméterére cseréli le. Fontos megjegyeznem, hogy ezek a függvények csak látszat osztályokon működnek ezért a *@Mock* annotációval jelölöm őket, amely eléri, hogy az osztály példányosítása során ne egy valós funkcionalitással rendelkező példány, hanem egy látszat példány jöjjön létre, amelynek függvényhívásai az általam megadott eredményeket adják vissza.

A megoldásomban a `@Mock` annotáció mellett megadom, hogy Mockito tesztkörnyezetet használok a `@ExtendWith(MockitoExtension.class)` annotáció segítségével és `@InjectMocks` annotációval pedig megadom, hogy melyik beanbe szükséges a látszat osztályokat létrehozni. Erre példa:

```
@TestComponent
@ExtendWith(MockitoExtension.class)
public class AuthenticationServiceTest {
    @InjectMocks
    AuthenticationService service;
    @Mock
    UserRepository userRepository;
    @Mock
    TokenRepository tokenRepository;
```

A regisztrációt fogadó és jelszó változtatáskor küldött email küldésének ellenőrzése volt egy számomra új feladat. A feladatot egy `ArgumentCaptor` létrehozásával valósítottam meg amely segítségével ellenőrizni tudtam az elküldött email tartalmát a következő módon.

```
@Test
void sendWelcomeMailTest() {
    service.sendWelcomeMail("email", "username");
    verify(javaMailSender).send(messageCaptor.capture());
    SimpleMailMessage mailMessage = messageCaptor.getValue();
    assertEquals("Welcome to Order tracker!", mailMessage.getSubject());
}
```

A fent látható kódban a `messageCaptor` segítségével lekérdezem az elküldött emailt és ellenőrzöm az email tárgyát.

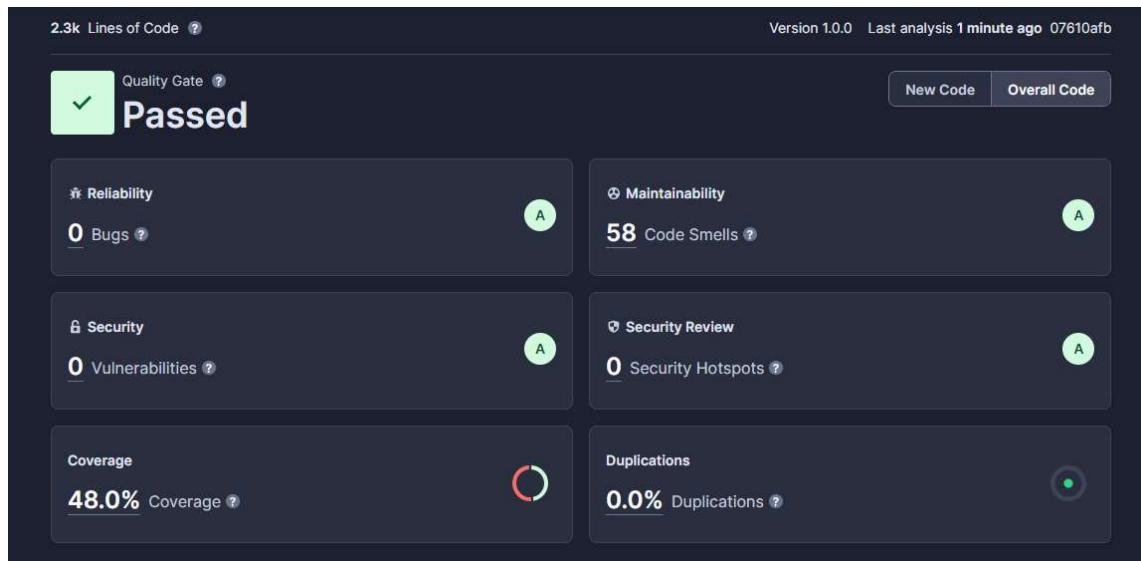
7.2 Sonarqube

Célomnak tűztem ki, hogy minden egyes git feltöltés után visszajelzést kapjak, mint fejletsző a kódom állapotáról. Erre egy statikus analízis eszközt a sonarqube-ot választottam megoldásul. Mivel az alkalmazásomat githubon verziókezelem és a github rendelkezik sonarqube integrációval, ezért elég volt mindössze egy sonarcloud profilt létrehoznom és egy `yml` fájlt definiálni a `.github/workflows` mappán belül. A `yml` fájlban definiálom, hogy a main ágra történő felültés esetén fusson le a sonarqube és ehhez a sonar számára létrehozott github titkokat (`secret`) használja.

28. ábra Githubon található sonarqube futások

Githubon belül az actions fül alatt láthatóak a futások és eredményeik. Mivel létrehoztam egy másik workflowt is csak az alkalmazás építésének (build) tesztelésére, ezért mindegyik külön lefut minden egyes main változtatáskor.

A sonarqube rendelkezik egy hozzá tartozó webes felülettel is, ahol különböző statisztikák érhetőek el az alkalmazás futtatásairól és megadhatóak különböző feltételek, amelyek alapján a sonarqube eldönti hogy az éppen feltöltött verzió elfogadható-e.



29. ábra Projekthez tartozó sonarqube analízis eredménye

Az képen látható kimeneten látható, hogy a Unit tesztek a kód 48%-át fedik le, amely a *service* osztályokat tartalmazza, ahogyan említettem az a kód üzleti logikájának tesztelését fedi le.

Véleményem szerint a sonarqube egy tökéletes alkalmazás a program fejlesztése közben történő tesztelésre, ráadásul így a tesztek fejlesztőkörnyezetben történő futtatása nélkül is elérhető azok eredménye.

8 Projekt összefoglalása

Az alkalmazás megvalósítása során sikeresen megvalósítottam feladat kiírásban meghatározott követelményeket és sikeresen lefedtem a szükséges funkcionálitásokat is.

A szakdolgozat elkészítése során rengeteget tanultam a különböző technológiákról és az azokat körülvevő konvenciókról, legjobb szokásokról. Az Angular, mint technológia a projekt kezdete előtt mindössze egy fogalomként volt tudatomban, viszont rengeteget tanultam a technológiáról a fejlesztés során és véleményem szerint sikerült az alapokat elsajátítanom. Megismertem többek között a modulokat és a komponensek felépítését, szerepét. Backend oldalon a Spring technológiát már a projekt kezdete előtt is ismertem, habár ezen a téren is sikerült jelentős ismeretekre szert tennem, erre példa az OpenApi integráció és a Spring security megismerése. Megtanultam a Spring technológiát alkalmazva emailt küldeni és a Spring applikációk dokkerizálásában is sikerült tapasztalatot szerezni.

A Stripe, mint harmadik fél integrálása is nagyon izgalmas feladatnak bizonyult és meglepően egyszerűen tudtam a fizetést integrálni az alkalmazásomba mindezt úgy, hogy minden össze az API kulcs cseréje szükséges a tényleges fizetéshez. A másik oldalon viszont a fájl feltöltéshez használt mikro szolgáltatás is sok tanulságot hordozott magában főleg frontend oldalon a byte halma megjelenítése volt kihívás.

8.1 Továbbfejlesztési lehetőségek

A projekt egy ételrendelő alkalmazás alapvető funkciót látja el, ezért természetesen rengeteg fejlesztési lehetőséget, potenciált tartalmaz az alkalmazás. Első sorban, mint a modernebb alkalmazásokban megvalósítható lenne a bejelentkezés integrálása valamilyen más platformmal, mint például a Facebook vagy Google fiók. Emellett létrehozhatóak akár kategóriák az ételekhez, akciózás lehetősége vagy akár lehetőség az esetleges allergiák definiálására a fiókban, amely szűrné a felkínált ételeket.

Megvalósítható lenne akár google maps integráció is, amely segítségével jelezni lehetne a vásárló számára, hogy honnan érkezik az étel.

Természetesen a felhasználói felület is fejleszthető, hogy még rögzítményesebb legyen és még modernebb felhasználói érzést keltsen.

Egy másik fejlesztési lehetőség lenne mondjuk grafana hozzáadása a projekthez, amely segítségével a hívott végpontokat lehetne ellenőrizni és visszajelzést kapna az alkalmazás karbantartója a kérésekéről és státuszaikról.

Végül a dockerizálás egy kezdő lépés lehet az alkalmazás felhőbe migrálásához, amely véleményem szerint az alkalmazás végcélja is lehetne.

9 Hivatkozások

- [1] „Az űsember élete és haladása.” 26. október 2023. [Online]. Available: https://www.geocities.ws/741/Tolnai_vilagtortenelem/01.04.html.
- [2] „A vendéglátás története.” 26. október 2023. [Online]. Available: https://hu.wikipedia.org/wiki/A_vend%C3%A9gl%C3%A1t%C3%A1s_t%C3%BCnete.
- [3] „Az éhes vásárlóval nem jó ujjat húzni – a NetPincér-sztori.” 26. október 2023. [Online]. Available: https://hvg.hu/kkv/20170404_netpincer_cegportre_ekereskedelem_szorad_gabor.
- [4] „Étel- és élelmiszer házhozszállítás új szintre emelve: ezért lett a NetPincérből foodpanda.” 27. október 2023. [Online]. Available: <https://www.foodora.hu/contents/netpincer>.
- [5] „Foodora weboldal által felhasznált technológiák.” 27. október 2023. [Online]. Available: <https://builtwith.com/?https%3a%2f%2fwww.foodora.hu%2f>.
- [6] „Wolt – a Finnországból indult népszerű ételrendelő szolgáltatás május 28-tól Budapesten is elérhető.” 27. október 2023. [Online]. Available: <https://insiderblog.hu/blogzine/2018/05/29/wolt-a-finnorszagbol-indult-nepszeru-etelrendelo-szolgaltatas-majus-28-tol-budapesten-is-elerheto/>.
- [7] „Introduction to Spring Framework.” 29. október 2023. [Online]. Available: <https://docs.Spring.io/Spring-framework/docs/3.0.0.M4/reference/html/ch01s02.html>.
- [8] „Spring IoC, Spring Bean Example Tutorial.” 29. október 2023. [Online]. Available: <https://www.digitalocean.com/community/tutorials/Spring-ioc-bean-example-tutorial>.
- [9] „Quick Guide to Spring Bean Scopes.” 29. október 2023. [Online]. Available: <https://www.baeldung.com/Spring-bean-scopes>.
- [10] „Introduction to Angular concepts.” 29. október 2023. [Online]. Available: <https://Angular.io/guide/architecture>.

- [11] „Introduction to modules,” 29 október 2023. [Online]. Available: <https://Angular.io/guide/architecture-modules>.
- [12] „Ordertracker on github,” [Online]. Available: https://github.com/rajmunddongo/order-tracker/tree/documentation_fixes. [Hozzáférés dátuma: 7 december 2023].
- [13] „Docker (software),” 29 október 2023. [Online]. Available: [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software)).
- [14] „TailwindCSSComponents,” 1 december 2023. [Online]. Available: <https://tailwindcomponents.com/>.
- [15] „RFC 7591,” [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc7591>. [Hozzáférés dátuma: 5 december 2023].
- [16] B. Ali, „Spring boot 3 jwt security example repository on github,” [Online]. Available: <https://github.com/ali-bouali/Spring-boot-3-jwt-security/blob/main/src/main/java/com/alibou/security/config/JwtService.java>. [Hozzáférés dátuma: 3 december 2023].

10 Ábrajegyzék

1. ábra <i>Netpincér eredeti oldala</i>	10
2. ábra Foodora mai (2023.11.07) oldala	11
3. ábra Wolt mai (2023.10.27) oldala.....	12
4. ábra A Spring modulok csoportosítva [7]	14
5. ábra Angular modulok architektúrája [11]	19
6. ábra Kereskedőhöz tartozó használati esetek.....	22
7. ábra Vásárlóhoz tartozó felhasználói használati esetek.....	23
8. ábra Vásárlóhoz tartozó profil kezelésére alkalmas használati esetek	23
9. ábra Kereskedő oldalának nézete mockup	29
10. ábra Főoldal képernyőkép	32
11. ábra Vásárló regisztráció képernyőkép	33
12. ábra Kereskedő regisztráció képernyőkép.....	34
13. ábra Bejelentkezés képernyőkép	34
14. ábra Kereskedő oldaláról képernyőkép.....	35
15. ábra Rendelés összegzése képernyőkép	36
16. ábra Fizetés integráció képernyőkép	37
17. ábra Rendelés állapotának követése képernyőkép	38
18. ábra Termék hozzáadása a kereskedőhöz képernyőkép	39
19. ábra Rendelések állapotának menedzselése képernyőkép	39
20. ábra Rendelések státuszának módosítása képernyőkép	40
21. ábra Elfelejtett jelszó felhasználói nézet képernyőkép	41
22. ábra Új jelszó megadása képernyőkép	42
23. ábra Admin felület képernyőkép	42
24. ábra Termék mentése a vásárló kosarába.....	50
25. ábra Képek lekérdezése a frontendről	27
26. ábra Githubon található sonarqube futások	62
27. ábra Projekthez tartozó sonarqube analízis eredménye.....	62

