

Eötvös Loránd Tudományegyetem
Informatikai Kar
Információs Rendszerek Tanszék

Pizzafutár webalkalmazás

Témavezető:

Kotroczó Roland
Doktorandusz

Szerző:

Mákos Dániel Nándor – E0DEFO
Programtervező informatikus BSc

Budapest, 2023

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:

Név: Mákos Dániel Nándor

Neptun kód: E0DEFO

Képzési adatok:

Szak: programtervező informatikus, lapképzés (BA/BSc/BProf)

Tagozat: Nappali

Belső témavezetővel rendelkezem

Témavezető neve: Kotroczó Roland

munkahelyének neve, tanszéke: ELTE IK, Információs rendszerek Tanszék

munkahelyének címe: 1117, Budapest, Pázmány Péter sétány 1/C.

beosztás és iskolai végzettsége: Doktorandusz, programtervező informatikus Msc

A szakdolgozat címe: Pizzafutár webalkalmazás

A szakdolgozat témája:

(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

Szakdolgozatom egy fiktív pizzéria futárszolgálati problémáit hivatott megoldani. A vállalkozás innovációja, hogy egy poszt-COVID időszak igényeit hivatott kielégíteni, így a kiszolgálás teljes mértékben korlátozódik a házhozszállításra. Ennek következménye (és lehetősége), hogy az ételt - annak frissességének megőrzése érdekében - a kiszállítást végző valamely járműnek a fedélzetén készítik az út folyamán.

A program szerveroldali megvalósítását Java Spring keretrendszerrel hajtom végre.

Legyen egy gráf tárolva, amely lefedi a cég kiszállítási területét. Például egy város stilizált útvonalhálózata, amit a szervert minden indulásakor a perzisztencia rétegből egy adatbázisból tölt be.

Egy útkereső algoritmus segítségével határozzuk meg, hogy a flotta elérhető járművei közül melyik végezze a rendelés teljesítését. A szimuláció során a rendelések folyamatosan érkeznek. Gyakoriságuk részben véletlenszerű, de igyekeznek visszaadni egy életszerű forgalmat. Pl.: csúcsforgalomra délben számíthatunk.

A szerveroldalon futó programnak gondoskodnia kell a járművek státuszának felügyeletéről. Egy járműhöz csak akkor köthető a rendelés teljesítése, ha az képes az üzemanyag- és alapanyagszintje szerint vállalni azt. Abban az esetben, ha nem, akkor a telephelyre kell visszaküldenie utánpótlásért.

A rendelést a kliens a menü olvasásával állíthatja össze, így azt is adatbázisban kell tárolni. A menü elkészítéséhez szükséges összetevőket is tudnia kell a szervernek, az előző bekezdésben említett járműkapacitás szempont miatt. Az így keletkező táblák, azok kapcsolatai MySQL adatbázisban kerülnek tárolásra.

A kliensek, amelyektől a rendelés és helyszínadat bekérésre kerül, szintén adatbázistáblából azonosítható be. A kliens szemszögéből a rendelés nyomonkövethetővé válik a rendelés leadása után, egy erre nyitott Websocketen keresztül.

A rendelések automatikusan naplózásra kerülnek, kényelmi funkció, hogy megfelelő jogosultsággal, kijelölt időintervallum szerinti, exportálhatóak Microsoft Excel táblába, a jobb vizualizálhatóság érdekében.

Budapest, 2023. 03. 09.

Absztrakt

A szakdolgozat egy fullstack webalkalmazás, amely *Java Spring* szerveroldali technológiát felhasználva *API* felületet (*Application Programming Interface-t*) biztosít a vele *HTTP* kapcsolatot (*Hypertext Transfer Protocol alapú*) létesítő kliensek számára. A kliens létrehozhat entitásokat a szerveren, amelyeket az *ORM* leképezés (*Object Relational Mapper*) segítségével tárol a háttérben futó adatbázisban.

A szerveren futó kód emellett betölt a memóriába egy súlyozott élű gráfot, amelyet további *szerver-kliens interakció* során olvas, útkeresés szempontjából. Az entitásokat különböző szempontok alapján a gráfon “mozgatja”, azaz futási időben, memóriában a létrehozott példányukhoz hozzárendel más-más gráfcsúcsot, mint tartózkodási helyszínt. Az entitás új helyszíne elérhető a korábbiából, azaz út köti össze veleⁱ. A két csúcs között felírható utak közül a legrövidebb meghatározása *Dijkstra* algoritmusánakⁱⁱ felhasználásával történik.

Ennek eredményeként a webalkalmazásom képes meghatározni, hogy a gráf több tárolt csúcsa közül, melyikből írható fel a legkisebb élköltségű út a gráf egy új pontjába. Ez példázza egy *dispatcher* szerepét, egy autóflotta menedzselésekor, ha azt kell meghatározni, hogy egy helyszínhez mely járműve ér oda leghamarabb.

Kulcsszavak: fullstack, webalkalmazás, gráf, legrövidebb út

Tartalom

1. Bevezetés.....	6
2. Felhasználói dokumentáció.....	8
2.1. Vásárló.....	8
2.1.1.Regisztráció	9
2.1.2.Rendelés.....	9
2.2. Séf.....	11
2.2.1.Szerkesztő táblák	11
2.2.2.Menü összetevőinek megadása	12
2.3. Sofőr	13
2.4. Admin	14
2.4.1.Fiók létrehozása szerepkörrel	15
2.4.2.Flotta menedzsment	15
2.4.3.Raktár- és rendeléselőzmények.....	15
2.4.4.Előzmény exportálás.....	16
2.5. Szimuláció	17
2.5.1.Alapanyag feltöltés	17
3. Fejlesztői dokumentáció	20
3.1. Szoftver futtatása	21
3.2. Adatbázis főbb táblái	21
3.2.1.User.....	22
3.2.2.Edge	23
3.2.3.Menu	24
3.2.4.Food Orders	24
3.2.5.Car.....	25
3.2.6.Inventory	25
3.3. ORM leképezés.....	25
3.3.1.Hibernate specifikus utasítások.....	26
3.3.2.Entitások kulcsai	27
3.3.3.Objektumok közötti reláció.....	27
3.3.4.Validáció	30
3.4. Autentikáció.....	31
3.5. Végpontok.....	32
3.5.1.Serializáció.....	33
3.5.2.Kérések	35

3.5.3.Websocket végpontja	38
3.5.4.Szimuláció	38
3.6. Végpontok tesztelése	41
3.7. Kliensoldali megoldások	43
3.7.1.Adatelérés	43
3.7.2.Adatkontextus	44
3.7.3.Manuális tesztelés	44
3.7.4.Bejelentkezés tesztése	44
3.7.5.Rendelés tesztése.....	45
3.7.6.Felugróablak tesztése	47
4. Összefoglalás	49
5. Hivatkozásjegyzék	50

Bevezetés

Szakedolgozatomban ötlete egy jelenkori probléma, vagy más szemszögből, üzleti lehetőség fennállása kapcsán született. A jelenleg működő, ételekre specializálódó futárszolgálatok sikere jól mutatja napjaink tendenciáját: az éttermek célközönsége a helyben fogyasztás helyett egyre többször választja a házhozszállítás alternatíváját. Egy elvárás viszont változatlan, mégpedig az, hogy a vendég mindig friss és étvágygerjesztő formájában szeretné a rendelését tányérján, vagy manapság inkább haddobozában látni.

Ebből ered a futárszolgálatok elsőszámú megoldandó problémája, az idővel való verseny. Aki ezt megnyeri, az a konkurenciával szemben domináns előnyre tesz szert. Erre a konklúzióra juthatott Julia Collins és Alex Garden is, a kaliforniai *Zume Pizza* startup vállalkozás két alapítója isⁱⁱⁱ. Ahhoz, hogy részesedéshez jussanak a nagyrészt *Dominos* és *Pizza Hut* által uralt piacon, robotizálni próbálták a pizzakészítés folyamatát. Mivel emberi felügyeletet így kevésbé igényelt, a pizza megsütését akár a házhozszállítás folyamán is el tudták végezni, a flotta járműveinek fedélzetén, átadás előtt pár perccel. A végeredmény a csökkentett várakozási idő és a friss pizza. A logisztikai problémák száma is csökkenhet, ha a jármű feltöltésekor előre megbecsülhetőek a rendelések és tárazható előre az azok elkészítéséhez szükséges alapanyagok, így nem kell a telephelyre annyiszor visszatérni.

A tématerületemnek választott webalkalmazás-fejlesztés is az utóbbi startup ötletéből nyert motivációt. Fiktív pizzériám telephelye Újbudán üzemel és innen látja el a teljes város, annak stilizált úthálózata mentén. Az oldal, felhasználói interakció után szimulálja, hogy milyen módon oldja meg egy rendelés teljesítését. Ehhez rendelkezésére áll egy több autóból álló flotta, amiket a központi telephelyről tölt fel alapanyagokkal. A városban cirkáló flotta azon tagjának adja ki a rendelés teljesítését, aki legközelebb áll a kiszállítási helyszínhez és rendelkezik az összetevőkkel, hogy az út folyamán, amíg célhoz ér, pizzát süssön belőlük. Gondoskodik arról is, hogy ha kifogy egy autó, azt a raktárhoz rendelje és utántöltse.

Dokumentációm tartalmaz mind az átlagfelhasználónak, mind a szoftverfejlesztő programozónak szánt fejezetet. A felhasználónak érdemes megértenie a webalkalmazás használata előtt, hogy milyen szerepkörök szerint strukturált a számára megjelenő tartalom. Megtudhatja, hogyan kapcsolhat szerepköröket fiókokhoz, majd a különböző fiókokkal való bejelentkezés után, mennyi funkcionalitást ér el az alkalmazás teljes

működéséből. Továbbá bepillantást nyerhet abba is, hogy milyen mellékhatások történnek a háttérben, bizonyos interakciók kapcsán.

A programozó bepillantást nyerhet abba, hogy milyen programozói eszközöket alkalmaztam az implementálás során, milyen architektúrát követ az alkalmazásom felépítése. Számára nem csak az a fontos, tudomást szerezzen arról, hogy a böngészőn keresztül milyen interakciók sorával éri el a funkciókat, de az is, hogy a szervert és a hozzá tartozó adatbázist képes legyen üzembe helyezni. A megvalósításnak megadom minden részletét ahhoz, hogy a jövőben képes lehessen más is arra hogy, új fejlesztésekkel, funkciók implementálásával növelje a program értékét.

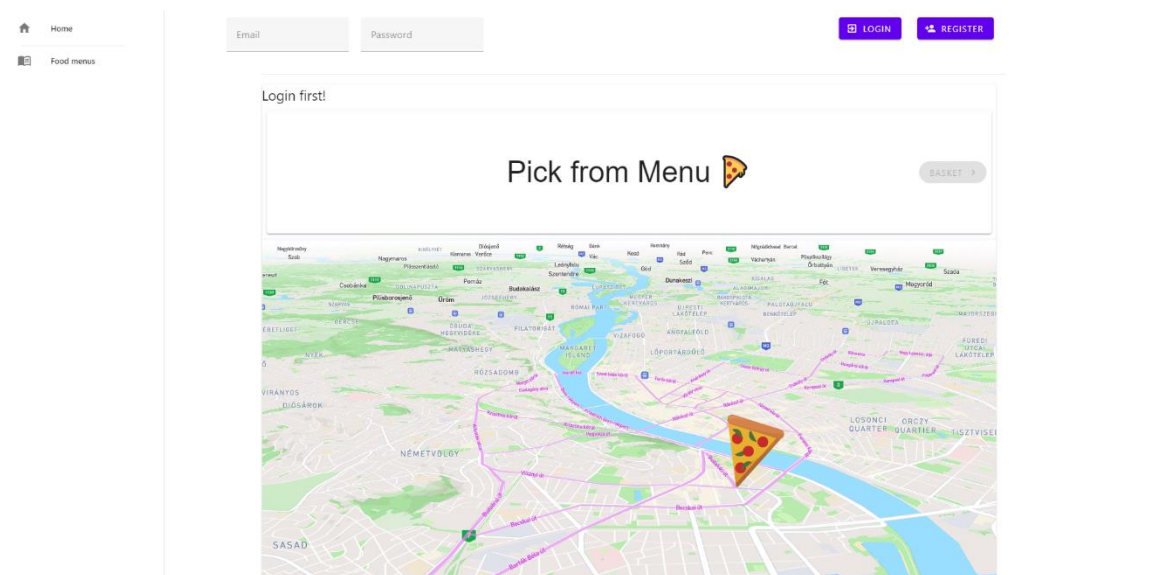


1. ábra - A Zume vállalkozás egy kamionja^{iv}

Felhasználói dokumentáció

Ahhoz, hogy a felhasználó számára leírt opciók elérhetőek legyenek, elengedhetetlen a 3.1. *Szoftver futtatása* c. fejezetben leírtak szerint eljárni, és *legalább* a *basic* adatcsomagot betölteni, amely biztosítja a john.smith@example.com admin-fiók meglétét.

Az útmutató az autentikációs szintek felől közelíti meg az applikáció használatát, hiszen oldal megjelenése nagyban függ attól, hogy mely szerepkör része a felhasználó. Ha az oldalra navigál a még be nem jelentkezett felhasználó, minimális funkcionalitását éri el az oldalnak: böngészheti a pizza menüt és megtekintheti az ételek allergénjeit, ha vannak (*Food menus* menüpont), emellett a főoldalon (*Home* menüpont) mozgathatja a térképet. További menüpontok megjelenítéséhez szükséges a bejelentkezés, erre a rendelés leadásához használatos lapozható felület is figyelmeztet a főoldalon.

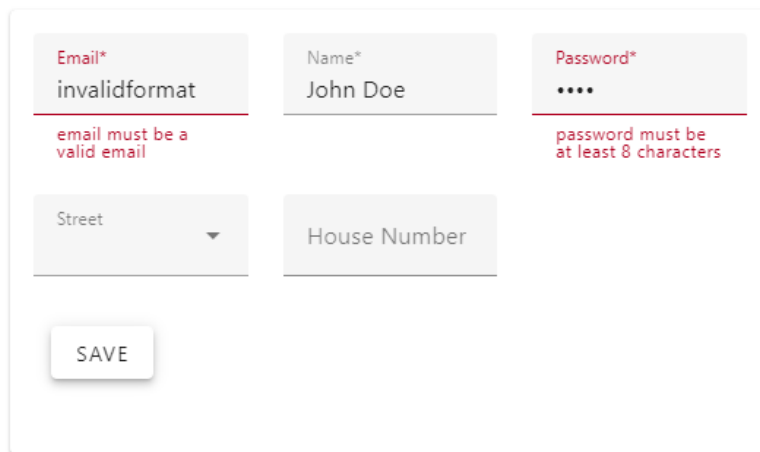


2. ábra - Főoldal

Vásárló

Az autentikációs szintek legalsó foka, amellyel minden regisztrált fiókot felruház a program, a vásárlói szerep (*Customer role*). Tehát bármely, az oldalra látogató friss felhasználónak vagy regisztrálnia kell a *Register* gomb segítségével, vagy bejelentkezhet a már meglévő fiókjába a regisztrációhoz felhasznált emailcím és hozzátartozó jelszó (*Password*) megadásával. Az oldal tetején lévő mezők hiányos kitöltése esetén a *Login* gomb nem enged tovább, rosszul megadott, nemlétező kombinációnál pedig hibaüzenetet kapunk, tehát az autentikáció sikertelen.

Regisztráció



The registration form contains the following fields and errors:

- Email***: invalidformat. Below the field, a red message states: "email must be a valid email".
- Name***: John Doe
- Password***: Below the field, a red message states: "password must be at least 8 characters".
- Street**: A dropdown menu.
- House Number**: A text input field.
- SAVE**: A button at the bottom left.

3. ábra - Regisztráció

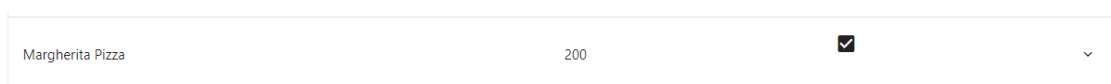
A regisztrációs folyamat elindításával egy űrlapot kap a felhasználó. Itt a kötelezően kitöltendő mezőket csillag jelzi. Emailnek csak megfelelő email formátum fogadható el, jelszónak minimum 8 karakter hosszúnak kell lennie.

Az utca (*Street*) kiválasztása egy legördülő panelen történik, a könnyebb használhatóság kedvéért ebben keresés is lehetséges. Ez, és a hozzátartozó házszám (*House number*) megadása opcionális, hiszen itt csak a kiszállítási területhez tartozó utca és az utca tárolt házszám-tartományán belüli szám választható, amelyet, ha nem tud még a vásárló, akkor rendeléskor is megadhatja. (Lásd következő oldal.) Amennyiben minden begépelt adat megfelelő, a *Save* gombra való kattintással befejezhető a folyamat, az adatok sikeres tárolásáról visszajelzést kapunk.

Rendelés

A bejelentkezett felhasználó munkamenetének (*browser session*) létrejöttét legegyszerűbben az oldal tetején, a *Login* és *Register* gombok helyett megjelenő *Logout*, azaz a kijelentkezés lehetőségét biztosító gomb megjelenése jelzi. Ekkor már elérhető a menüből a felhasználó (*User*) ikon alatt a saját adatok megtekintésére és szerkesztésére, cím hozzáadás. A fiók törlése a *Delete* gombbal kérhető. Ezen az oldalon, még a múltbéli rendeléseinket is kilistázza a rendszer.

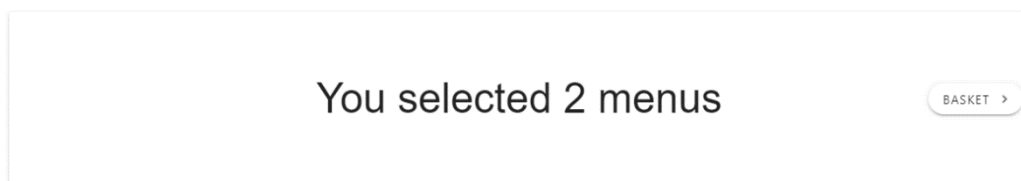
Az új rendelés létrehozásának első lépése, a pizza menük oldalán, a megrendelni kívánt tétel(ek) kiválasztása, vagyis a jelölő-négyzetének bepipálása. (egyelőre a tétel darabszámára való tekintet nélkül)



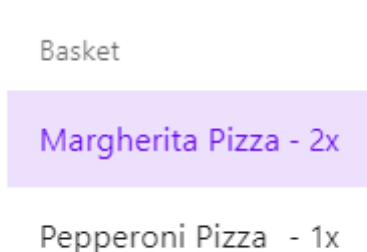
Margherita Pizza	200	<input checked="" type="checkbox"/>	▼
------------------	-----	-------------------------------------	---

4. ábra - Kiválasztott menü

Ha rendelhető tételek kerültek a kosarunkba, akkor egy kosár ikon jelenik meg az oldal tetején, a *Logout* gomb mellett, erre kattintva a rendszer visszavigyal a kezdőoldalra, ahol aktívvá vált a rendelésleadási folyamat lapozó-füle.



5. ábra - Rendelési folyamat



6. ábra - Kosár tartalma

A lapozófülre való kattintás után először egy listát kapunk a kosár teljes tartalmáról. Itt, a lista elemeikén szereplő ételekre is kattinthatunk, a továbblépés előtt. A pizzák klikkelése teszi lehetővé, hogy a rendelésbe az adott tétel többször is bekerüljön. Maximálisan ötig növelhető egy tétel előfordulása, ezután körbeér a számláló és visszaáll egyre.

Az oldal tovább lapozásával elérünk a kiszállítási cím megadásához, amelyet a rendszer automatikusan kitölt, ha létezik már a felhasználói fiókhoz rendelt cím. Amennyiben ezt most töltjük ki, használható segítségként a térkép, ahol jelölt a pizzeria helyszíne, és lila színnel a lehetséges kiszállítási útvonalak hálózata. Így könnyen meghatározhatjuk a számunkra legjobban kiszállítási pontot. A cím megadása után lépünk tovább a következő oldalra, ahol a végösszeget látjuk. A Felhasználói Feltételek elfogadásával (jelölőnégyzet pipálása, *Agree to Terms and Conditions* néven) leadhatóvá válik a rendelés. Lehetséges, hogy a megrendelés egyáltalán nem kiszállítható annak leadásának pillanatában, mivel nincsen a kiszállításhoz elérhető jármű. Ekkor nem jelenik meg a záróoldal, helyette hibaüzenet érkezik és a kosár tartalma is megmarad, hogy később újra próbálhassa a felhasználó. Ha volt futár, aki felvette a rendelést, konfetti animáció kíséretében eljutunk a záró oldalra, ahol jelen időben változik a futár helyzete, mígnem megérkezik a rendelés. A kosár kiürül.

Séf

A következő, oldal megjelenítése szempontjából megkülönböztetett szerepkör a *chef*. A nevéhez méltán az étel menük összeállításához kapcsolódó menüpontok jelennek meg pluszban: alapanyagok (*Food Ingredients*) és ételallergiák (*Ingredient Allergies*). A séf bejelentkezését követően az ételek menüpontból elérhető táblázat is kiegészül szerkeszthetővé.

Szerkesztő táblák

Ezen szerkesztő-felületű (*Create, Read, Update, Delete*) táblázat található mindhárom menüpontban, illetve még más szerepkörhöz kapcsolódó menüpontban is. A rekordokhoz kötődően esetleg további részletek is felmerülhetnek, de az alapvető elrendezése ugyanaz marad.

The screenshot shows a table titled 'Ingredient table' with columns: Name, Allergy, Price/Unit (Ft), and Actions. The table contains 11 rows of ingredients. Annotations include: 'Táblázat neve' pointing to the title; 'Keresés' with a search icon; 'Új rekord készítése' with a '+' icon; 'Oszlop szerinti rendezés' pointing to the Name column header; 'Oszlopnév' pointing to the Allergy column header; 'Sorban lévő rekord szerkesztése' pointing to edit and delete icons in the Actions column; 'Elérhető oldalak lapozása, vagy ugrás az elejére/végére' pointing to pagination controls; 'Megjeleníteni kívánt rekordok száma oldalanként' pointing to the 'Items per page' dropdown; and 'Megjelenített rekordok tartománya az összesből' pointing to the '1-10 of 11' range.

Name	Allergy	Price/Unit (Ft)	Actions
flour	gluten	250	[edit] [delete]
ham		500	[edit] [delete]
corn		10	[edit] [delete]
cheese	lactose	300	[edit] [delete]
chili		50	[edit] [delete]
mushroom		200	[edit] [delete]
salami		2000	[edit] [delete]
tuna		100	[edit] [delete]
asparagus		500	[edit] [delete]
artichoke		500	[edit] [delete]

7. ábra - Táblázat^{vi}

The dialog box titled 'Ingredient details' has three input fields: 'Name' with the value 'mozzarella', 'Price' with the value '100', and a 'Select' dropdown menu with 'gluten' selected. At the bottom are 'CANCEL' and 'SAVE' buttons.

Ingredient details

Name

mozzarella

Price

100

Select

gluten

CANCEL

SAVE

8. ábra - Dialógus ablak

Az új rekord készítése gomb minden táblázatnál az aktuális rekordtípusra szabott mezőkkel rendelkező, illetve a mezőket a rekordok természete szerint validáló

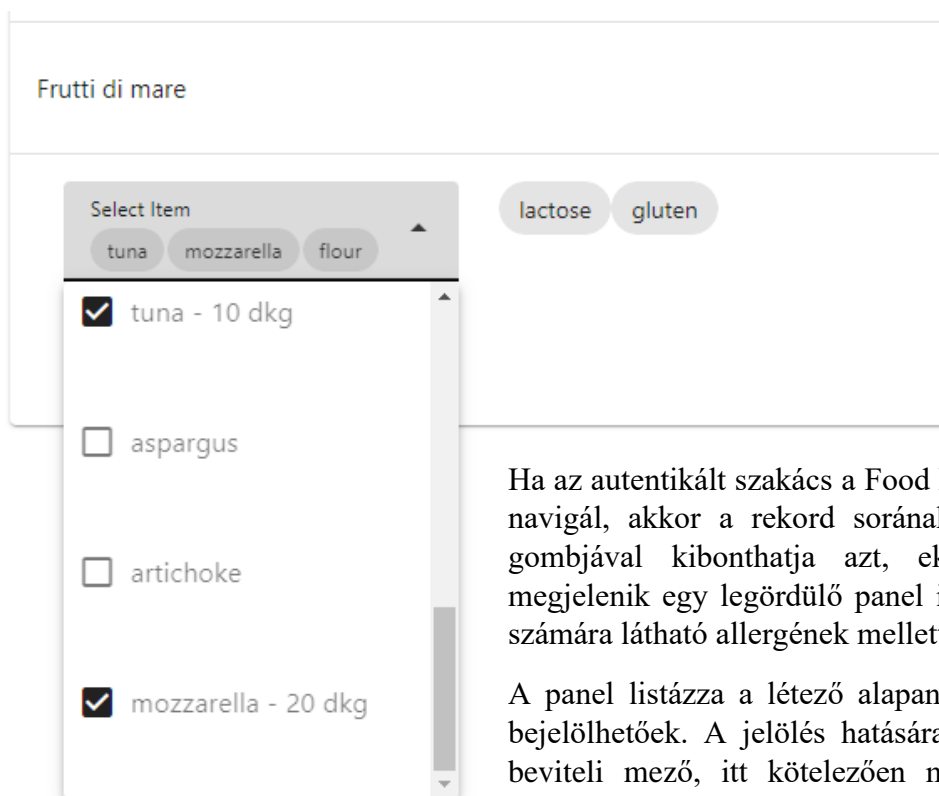
dialógus-ablakot jelenít meg. A példát képező táblázatban jól látszik, hogy egy alapanyaghoz például nem kötelező az allergének közül hozzárendelni, így menthető anélkül is, viszont név, és egységár mindenhol kell, hogy szerepeljen. A szerkesztése esetén is ugyanezt az ablakot kapjuk, de a mezőkben megjelennek a rekord aktuális értékei. A mentés (*Save* gomb), sikeres validáció esetén a dialógust bezárja, visszajelzés jelenik meg a mentés sikerességéről. Sikeres mentéshez fontos, hogy ha bármely, eddig említett három táblázathoz új rekordot adnánk, vagy meglévőnek nevét (*name*) szerkesztenénk, akkor sem szerepelhet ebben az oszlopban két azonos érték. Másképpen fogalmazva, a rekordoknak név szerint egyedülieknek kell lenniük.

Törlés nem hajtható végre minden esetben:

- Nem törölhető az az allergén, ami hozzá van rendelve bármely alapanyaghoz.
- Alapanyag sem törölhető, ha már szerepel bármilyen menüben, vagy futár/raktárkészlet-nyilvántartásban, autóban.
- Menü törléséhez biztosra kell mennünk, hogy nincs olyan korábbi rendelésfelvétel, ami tartalmazza.

Menü összetevőinek megadása

Ha vannak létrehozott alapanyagok, és a menük is elnevezésre kerültek, akkor a séfnek szükséges egymáshoz rendelni őket. Így a vásárlók megtekinthetik az esetlegesen szereplő allergéneket. A hozzárendeléskor megadandó a receptben szereplő összetevő mennyisége is, ez a menü megrendelésekor az alkalmas futár meghatározása miatt elengedhetetlen.



Ha az autentikált szakács a Food Menus oldalra navigál, akkor a rekord sorának jobb szélső gombjával kibonthatja azt, ekkor számára megjelenik egy legördülő panel is, a mindenki számára látható allergének mellett.

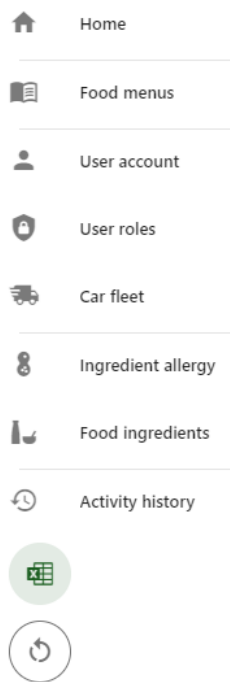
A panel listázza a létező alapanyagokat, ezek bejelölhetőek. A jelölés hatására felugrik egy beviteli mező, itt kötelezően meg kell adni számmal egy dkg mértékegységben értendő mennyiséget.

Amennyiben ezt a felhasználó megszakítja, a kiválasztása az alapanyagoknak semmissé lesz, tehát a hozzárendelés sem történik meg. Ha sikeres a hozzárendelés, arról visszajelzést ad a felület. Ez bármikor visszavonható a kiválasztás törlésével a listából. A lista bővítésével megjelennek az új allergének is.

Sofőr

Létező szerepkör még a *Driver*, aki a rendelés kiszállítására használt autók vezetéséért felelős. A számára pluszban megjelenő *Car fleet* menüpontban azt az autót látja kilistázva, amelyhez ő van csatolva, mint sofőr, vagy olyan autót, amihez nincsen még sofőr hozzárendelve. Ekkor számára a táblázat szerkesztési lehetősége elérhető, autóként megjeleníthető egy *Leave* és *Drive* gomb. Értелеmszerűen az az aktív, ami megváltoztathatja az állapotot. Ha a sofőr egy új autót szeretne vezetni, biztosra kell mennie, hogy más autóhoz nincsen hozzákapcsolva, vagy ha igen, akkor kijelentkezik abból először. Ellenkező esetben sikertelen mentésről kap visszajelzést.

Admin



A minden előző szerepkört magába foglaló az *admin* szerepköre. A képen látható az elérhető menüpontok teljes listája, amelyben a *chef* és *driver* szerepköréből ismert menükön kívül még elérhető egy szerepkörök és egy előzmények menüpont. Az admin-felhasználói nézetéhez kapcsolódik szintén két gomb, melyekből az első az előzmények exportjához köthető, másik pedig a szimuláció befolyásolásához. (Lásd 2.5. *Szimuláció* alfejezete.) A *User roles* listázza nekünk az összes eddig említett szerepkör tárolt példányát. Amennyiben nincsen hozzájuk kapcsolt felhasználói fiók, ezek törölhetőek, viszont amennyiben újonnan szükség lenne rájuk, ügyeljünk rá, hogy egyezzen a pótoltt példány neve, a régijével. (Az admin fiók nincs feltüntetve, hiszen annak törlése az előbbi kijelentések alapján sosem lehetséges.) Ezeken kívül is létrehozhatóak szerepköri példányok, persze más névvel, mint bármely előző. Ők, a nevükből eredően, már nem fognak különböző menüpontokhoz hozzáférést biztosítani a felhasználói fiókjaiknak, viszont fiók-kategorizálás szempontból hasznosak lehetnek.

10. ábra – Teljes oldalmenü

Fiók létrehozása szerepkörrel




Mielőtt még a többi új funkciót tárgyalná a felhasználói dokumentáció, pár meglévőhöz kapcsolódó új funkcióra tér ki. A szakács menüpontjai mind megjelenésükben és funkcionalitásukban is megegyeznek a két különböző szerepkörben. Vessünk azonban egy pillantást a *User account* pontra. Itt az admin a saját fiókján kívül egy táblázatban találja az összes másik fiókot, amelyeket a korábban megismert módon szerkeszthet. Törlésük sikertelen, amennyiben szerepelnek autósóforként. Továbbá létrehozhat fiókot, ekkor bármilyen létező jogosultsággal, a regisztrációs módszerrel ellentétben. (Admin fiók nem távolítható el és új sem hozható létre.) Egy emailcímmre nem hozható létre több fiók természetesen.

Flotta menedzsment

Az admin nagyobb hatáskörrel rendelkezik a rendeléseket kiszállító flotta irányításában is, lát minden autót. Hozzáadhat újakat, egyedi rendszám táblával, akár egyből beültethet bármilyen *driver* szerepkörű személyt, aki másik autóhoz nincs hozzárendelve. Meglévő, vezetett autóból pedig kimotozhatja annak sofőrjét. Autó törlése akkor lehetséges, ha nem része futár/raktárkészlet-nyilvántartásnak és a szimulációban sem szerepel. Az utóbbi akkor vehető biztosra, ha a szimulációt. (Lásd 2.5. *Szimuláció* alfejezete.)

Raktár- és rendeléselőzmények

Az említett új oldalak közül a *History page* egyedül az admin számára elérhető. Itt három különböző fül alatt tekintheti meg a rendeléseket kiszállító rendszer működésének mellékhatásaként létrejövő rekordokat, gyakorlatilag *log* bejegyzéseket. Az *Inventory* fül alatt találjuk, az autók alapanyaggal való ellátásáért felelős raktár készleteinek módosulását leíró bejegyzéseket. A sorok értelmezésekor tartsuk fejben, hogy egy bizonyos alapanyaghoz kötődő mennyiség mindig a jelenleg elérhető teljes összegzett mennyiséget jelöli. Tehát az alábbi képen, ahol a rekordok idő szerint csökkenő sorrendben vannak, azt láthatjuk a *Current quantity* oszlop alapján, hogy az *asd-123* rendszámú autó 1988 egységnyi lisztmennyiségről feltölti a raktárt 2000 egységre, a különbözetet pótolni pedig 3000 egységnyi pénzbe került, az *Expense* oszlop alapján.

Modified at: ↓	Ingredient	Car	Current quantity	Expense	Actions
2023-05-17T17:42:16.000+00:00	flour	asd-123	2000	3000	
2023-05-17T17:30:55.000+00:00	ham	asd-123	1980	0	
2023-05-17T17:30:55.000+00:00	flour	asd-123	1988	0	

11. ábra - Raktárelőzmények

Az *expense*, azaz kiadás 0 értéket kap, ha úgy keletkezett a sorban látható mennyiség, hogy az autó feltankolt, tehát csökkentette, vagy munkája végén deponálta a nála maradt, fel nem használt mennyiséget. Hogy a kettő közül melyik esetről volt szó, azt úgy tudjuk megállapítani, hogy megvizsgáljuk, hogy az élelmiszer mennyisége csökkent-e, vagy nőtt, a legutóbbi róla szóló bejegyzéshez képest. A raktárkészlet a program futása során sokszor automatikusan módosul, viszont a felhasználó is hozzáadhat hiányzó alapanyagból manuálisan. (Bővebben a 2.5. *Szimuláció c.* alfejezetben.) Ekkor azt a mennyiséget kell megadnunk az adott alapanyaghoz, amennyivel növelni akarjuk a mennyiségét. A kiadás értéke ezután ki lesz kalkulálva és megjelenik a sorban.

Mivel a rekordok mindig a keletkezett mennyiséget jelölik, így a törlés során figyelembe kell vennünk azt a korlátozást, hogy alapanyagonként mindig csak az időben legutolsó módosítást fogja törölni a rendszer. Ezáltal visszapörgethetjük az állapotát a raktárnak.

Az *Orders* fül alatt megtekinthetjük a felvett rendeléseket, és sorszámukat vizsgálva megtaláljuk a *Deliveries* fül alatt, hogy mikor kerültek kiszállításra. Műveletként törölni lehet, viszont mivel egy rendelés felvétele és kiszállítása szoros relációban áll egymással, így, ha az *Orders* egyik rekordja megszűnik, a hozzátartozó kiszállítás is törlésre kerül.

Előzmény exportálás

A fülek alatt található táblák

a hozzájuk tartozó, szerveren tárolt adathalmaz utolsó 100 rekordját jelenítik meg. Hogy a teljes táblákat lementhessük, lehetőségünk van exportálni. Ez a korábban említett gombok közül az *Excel* ikonnal történik. Ha rákattintunk, először felugrik egy dialógus ablak,

amelyben megadhatjuk a kívánt dátumot, ami előtti bejegyzéseket szeretnénk menteni. Ezután a webalkalmazás készít számunkra egy *xlsx* kiterjesztésű fájl, amely azt a dátumot kapja névként, amely alatt a legkésőbbi rekord található a korábban megadott

Print records before:

<

May 2023

>

Mon	Tue	Wed	Thu	Fri	Sat	Sun
01	02	03	04	05	06	07
08	09	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	01	02	03	04

DOWNLOAD

12. ábra - Exportálás dialógusa

tartományon belül. A letöltött fájl tartalmazza a három táblának megfelelő három munkalapot.

Szimuláció

Ahhoz, hogy megfelelően üzemeltessük az oldalt és a lehető legtöbb rendelés kiszállításra kerüljön, fontos átlátni a szerveren működő háttérfolyamatokat. A szimulációra legegyszerűbb úgy tekinthetünk, mint egy munkanapra. Reggeli tevékenység a raktár kapacitásáig való feltöltése, innentől kezdve a sofőrök hozzárendelhetőek a parkolóban várakozó pizzakészítő- és kiszállító járművekhez, hogy forgalomba helyezzék őket. A munkanap feltehetőleg véget is fog érni, ekkor a rendszer feladata, hogy visszaállítsa állapotát egy új munkanap megkezdésére alkalmas helyzetbe, tehát minden autót visszarendeljen a parkolóba, azok sofőrjeit kiléptesse, bennük tárolt maradék, el nem adott alapanyagokat a raktárba deponálja. Az utóbbi folyamat megkezdése az admin által, parancs formájában bármikor kiadható, a menü oszlopában megjelenő alsó gombbal. Ekkor a nap elején említett feladat, a raktár hiányzó mennyiségeinek pótlása is újra megtörténik.

Mit érdemes figyelembe venni üzemeltetőként? Amennyiben nincs autó, amelyhez sofőr van hozzárendelve, semmilyen rendelés kiszállítása sem lehetséges. Amennyiben van, a rendszer minden rendelés beérkezte után priorizálja a vezetett autókat, a kiszállítási címhez való közelségük szerint, hogy kinek kellene teljesíteni azt. Ha a sofőr aktuálisan ült az autójába, és mivel a helyszín a raktáré, megtörténik az autó annak kapacitásáig való feltöltése, a raktárban tárolt alapanyagokból. Így, ha az ő helyzete a legközelebbi, akkor biztosan elkészíti a menüket, és kiszállítja. Később is visszaküldhet autót a rendszer a raktárhoz, alacsony vagy nem elégséges összetevőmennyiség esetén.

Alapanyag feltöltés

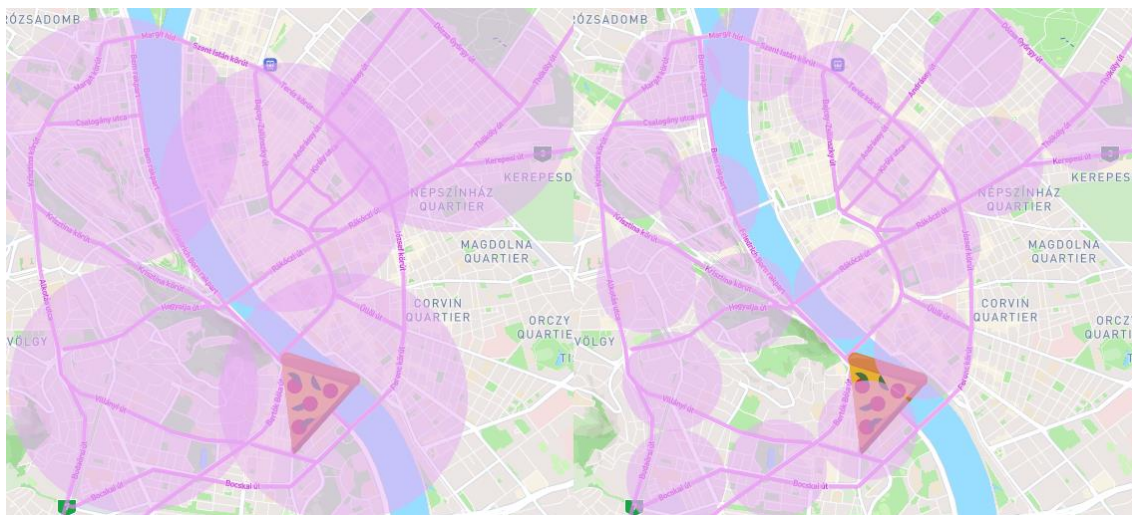
Amiért fontos érteni az autók fel- és újra töltését is, mert annak automatikus elvégzése - továbbá a rendelések folyamatos készítése és teljesítése - csak akkor garantált, ha van megfelelő mennyiségű tartalék a raktárban. Ha esetleg a szakács napközben állít össze egy menüt egy újonnan rögzített alapanyagból, amelyre egyből befut egy rendelés, akkor előfordulhat, hogy egyáltalán nem is fellelhető a raktárban, ami a sütéshez szükséges. Csak abból kerül tárolásra, ami már a szimuláció kezdetekor létezett már. Ha a rendelés csak az új menü miatt nem lenne teljesíthető, a szoftver nem keresi tovább az elkészítésre alkalmas autókat, hiszen semelyik sem rendelkezik még az összetevővel, emellett az újratöltés sem megoldás. Az új menü kihagyása a rendelésből, ami megoldja a

patthelyzetet. Viszont korántsem ilyen egyszerű a megoldás, ha például a, minden pizzához elengedhetetlen liszt nincs elegendő egy autóban sem, és már a raktár sem tud további utánpótlással szolgálni.

A lehetséges megoldások az admin részéről:

- a) Kiiktatni egy autó sofőrjét, ekkor az üres autó úgy lesz lekezelve, hogy alapanyagai deponálásra kerülnek, és ez a plusz behozatal elég lehet más autó újra töltésére.
 - Előnye, hogy lehet kiadás nélkül oldja meg a helyzetet.
 - Hátránya, hogy nemtudható mennyi autót elegendő ehhez kivonni a forgalomból, az új alapanyag problémáját semmiképpen sem oldja meg.
- b) A raktár kezelőfelületéről pótolni a hiányzó alapanyagmennyiséget.
 - Célzottan megoldja bármelyik problémát.
 - Hátránya, hogy sok rendelés teljesítése után valószínű, hogy nemsoká újra jelentkezik a probléma csak más alapanyaggal.
- c) A szimuláció újraindítása a korábban említett gombbal.
 - A raktár minden alapanyagának (immáron az újaknak is) kapacitásig való feltöltése hosszútávon fedezi a flotta szükségleteit.
 - Hátránya a teljes flottára gyakorolt hatása: minden autót visszarendel, kiürít, és szétkapcsol a sofőrjétől.

Az üzemeltetés során megfontolandó, hogy mennyi autót indítunk egy nap folyamán. Minél több autóval rendelkezhet a rendszer, azokat a térkép minél több pontján helyezheti el, annál nagyobb a garancia a rövidebb utakra. Ellenkező esetben, például egytagú flottával, ha az utolsó rendelése miatt Pestre megy az autó, nem nagyobb annak a valószínűsége, hogy maradhat ezen a városrészén, mint annak, hogy vissza kell térnie Budára a legközelebbi rendelés alkalmával. A kérdés jól szemléltethető az alábbi ábrával. Fedjünk le egy térképrészletet először nagyobb, 250px széles, majd kisebb, 100px-es körökkel. Egy kör az egy autó kiszállítási területét fogja szimbolizálni. Míg a baloldali térképrészlet lefedése 5 darab körrel történt, ahol egy kör sugara kb. 1,2 kilométer - ezt a távot minden autó potenciálisan teszi meg, - addig jobb oldalon ez a távolság redukálódott 500 méterre, viszont 18 darab kör árán. (A magyarázat erejéig tekintsünk el a körök metszete alá eső területektől és attól a tényről, hogy az autó nem helikopter módjára, légvonalban közlekedik.)



13. ábra - Térképrészlet^{vii}, $1px \sim 10.09$ méter

Több autó előnye egyértelmű, viszont felső határ kiszabása is érdemes a flotta méretére, mégpedig azért, hogy elkerüljük a raktártartalékok elaprózását. *(Előző bekezdésben tárgyalt probléma.)*

Fejlesztői dokumentáció

Első lépésként vegyük sorra, hogy a program futtatásához milyen hardverkövetelményeknek kell, hogy teljesüljenek.

- Intel(R) Core(TM) i3-6100 CPU, vagy annál újabb
- Minimum 8 Gb RAM
- Programfüggőségek telepítésre kevesebb, mint 500 Mb tárhely
- Adatbázis fájlok számára min. 50 Mb

Mivel a szoftver még fejlesztési stádiumában jár, így futása *local* (helyi) környezetben történik. Mit értünk ez alatt? Semmilyen *integration pipeline* (automatizált szoftverfolyamat a programunk más környezetbe való integrálására^{viii}) nem biztosítja az integrációt/deployment-et távoli szerverre. Egyedül a fejlesztés során használt IDE (*Integrated Development Environment*) alkalmazás tesztesei futtathatók manuálisan. Tehát mind a server- és adatbáziskommunikáció, mind a front-end megjelenítésére generált kliensoldali *JavaScript* kód lekérése, egy-egy helyi, erre a célra nyitott hálózati *port* használatán keresztül történik. A futtatáshoz bizonyosodjunk meg róla, hogy szabadok az alább említett portok.

Helyi környezeten futó/futtatandó szoftverek a fejlesztés pillanatában:

- Windows 10 - 21H2 verziószám
- XAMPP 3.3
 - MySQL - localhost:3306 cím
- Java JDK 17
- Maven 3.9
 - Spring Boot 3.0 - <http://localhost:8081> cím
 - MariaDB Java Client 3.1
- NPM 9.6.4
 - Vite 4.2 - <http://127.0.0.1:5173> cím
 - Vue 3.2
 - Vuetify 3.1

Szoftver futtatása

A *start_server.ps1* nevű file futtatásának hatására a következő folyamatok fognak elindulni, szekvenciális sorrendben. Ez fontos a szerver bekonfigurált adatbázisán való függősége miatt, illetve annak pár táblája miatt, ahonnan a szerver már az induláskor töltene be adatokat.

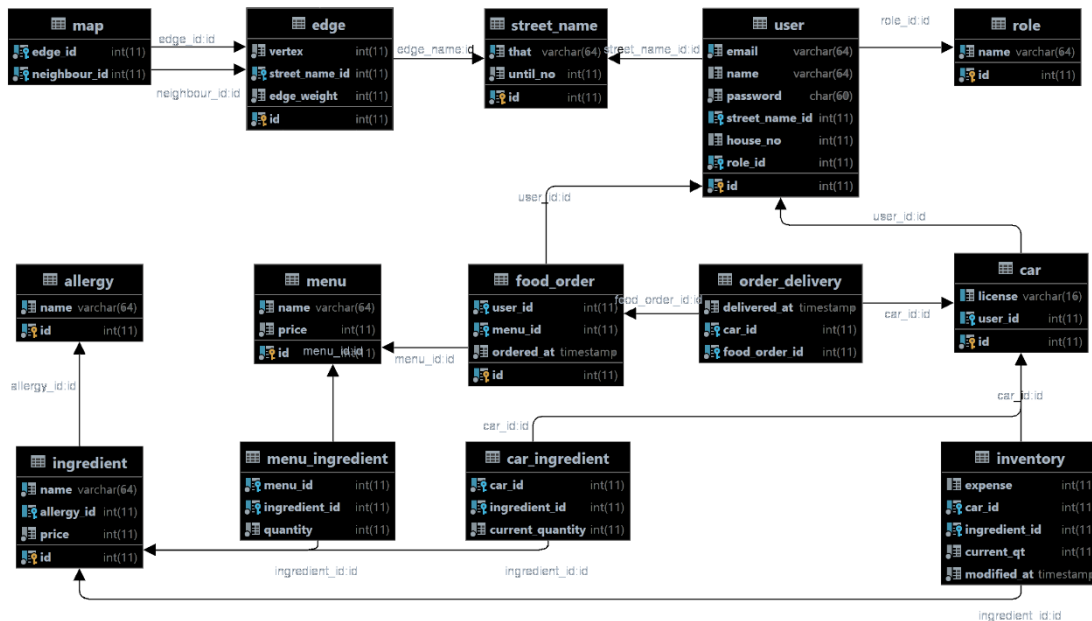
1. XAMPP program indítása, amely futtatja az adatbáziskezelő motort, az arra megadott konfiguráció szerint.
2. Adatbázis létrehozása
 - 2.1. Táblák definiálása
 - 2.2. Bizonyos táblák feltöltése a *FetchMapData.js* visszatérési adataival
3. Szerver, telepítése, indítása.

Ha a szerver a kérések fogadására kész, két féle adatcsomag betöltése indítható: a *script/db_fill_basic.ps1* fájl által kiadott kérés csak az *admin* fiókot hozza létre, amíg ugyanitt a *db_fill_test* már a tesztelésre alkalmas mennyiségű adatot szűr be. A két csomag között való váltás előtt futtassuk a *db_initialize* fájlt.

A kliens bármikor indítható a *start_client.ps1 fájl* használatával.

Adatbázis főbb táblái

A szakdolgozatom megvalósítására létrehozott szerver, az absztraktban említett módon, adatait egy adatbázisban tárolja. Ezek hozzáférését MVC (**M**odel **V**iew **C**ontroller) architektúrával biztosítja a klienseinek, az autentikáció függvényében. Fejlesztői dokumentációm a modell létrehozás felől kezdi a főbb algoritmusok magyarázatát, sorra veszi az adatbázisban létrehozott táblák ORM leképezését.



14. ábra - Adatbázis táblák diagramja

User

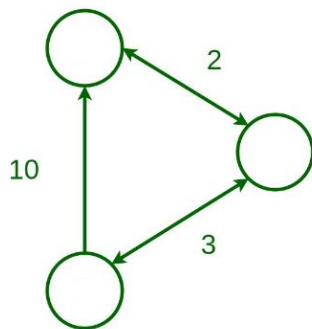
A *User* tábla felelős a felhasználó regisztrációját követően létrejött adatok tárolásáért. Az *email* mezője egyedi, tehát egyértelműen azonosítja a sort a táblájában. Típusa *varchar*, ami hasonló a karakterek sorozatát tároló *string* típushoz, azonban itt létrehozáskor megadható a maximum hosszúsága, ez jellemzően 64 karakterhosszúság lesz az adatbázisban. Az említett típushoz tartozik még a *name* mező is. A *password* mező *char* típusú, fix 60 karakter hosszúsággal. Ebben a *BCrypt* algoritmus által, a megadott jelszóból generált 448 bites *hash* kódokat fogjuk tárolni^{ix}.

A következő *role_id* mezőben egy másik tábla, név szerint a *Role*, egy rekordjának *PK* kulcsát (**Primary Key**) tároljuk, ami egyediként azonosítja annak egy rekordját. (megemlítendő, hogy a *Role* tábla *name* mezője is egyedi, viszont, ahhoz, hogy hivatkozzunk rá, a rekord *id*-jét használjuk). A *role_id* mező csak olyan *int* típusú szám lehet, ami *id* néven nyilvántartott kulcsként a *Role* táblában létezik. Ezt a megkötés a *User* tábla szempontjából egy *FK constraint (Foreign Key)*, azaz idegen kulcsra adott korlátozás. A jelenlegi *User* táblának is van ilyen *PK* kulcsa, így más tábla pl. *Car*, hivatkozhat rá egy saját rekordjának sorából.

Minden eddig említett mező a *not null*, azaz nem nulla kategóriába tartozik: vagy a bevitt adatból kell, hogy kötelezően létrejőjenek, vagy a PK esetében annak *auto_increment* típusa miatt, mindig az előző sor mezőjéhez képest eggyel nagyobb számmal hozza létre a rendszer. A maradék *street_name_id* és *house_no* *int* típusú mező, az előzőkkel ellentétben, később is kitölthető, ha a rekordot frissítjük. Emlékezzünk vissza a Felhasználói dokumentáció 2.1.1 *Regisztráció* c. fejezetére, ahol az említett űrlap utca és házszám mezőjének kitöltése csak opcionális. Emellett a *street_name_id* mező is egy FK megkötéssel rendelkezik, emiatt szuffixuma nevének az “id”. A többi tábla mezőinek elnevezésekor is a következő konvenció szerint jártam el, tehát ha a mezőnév “id” végződésű, akkor a mezőnév elejével megegyező táblának *id* PK mezője által azonosított sorára hivatkozik. A *Street_name* tábla *that* mezője a *Role* tábla *name* mezőjéhez hasonlóan egyedi.

Edge

Az *Edge* tábla használata a program során csak az algoritmusok által történik, a felhasználó nem szerkesztheti a tartalmát. A tábla adatai a program futtatása elején, script kód segítségével kerülnek feltöltésre egy külső API használatával. (Lásd 3.1 *Szoftver futtatása* c. fejezet). Mezői a *vertex*, ami tárolja az él legmagasabb számú házszámát, *street_name_id*, amely a *Street_name* tábla egyik rekordjára hivatkozik, az *edge_weight* pedig a hosszát hivatott tárolni. A hossz méterben értendő. Az API adta megkötés, amelyen később a helyszínonosítás alapszik, hogy az *id* szám növekedésével ellentétben csökken a legnagyobb házszám, amennyiben több *Edge* rekord is ugyanazt az utcanevet hivatkozza. Hétköznapiabb megfogalmazásban: több él is ugyanazon utca része.



15. ábra - Példa gráf

Egy élnek a kapcsolatait a *Map* tábla tárolja. Amennyiben egy irányított gráf éléből elérhető annak a gráfnak egy másik éle, akkor található a *Map* táblában olyan rekord, ahol relációban szerepel a két él. Példának a balra látható gráf^x kapcsolatait az alábbi módon kerülnének tárolásra: {(10, 2), (2, 10), (2, 3), (3, 2), (3, 10)}, amely halmaz rendezett kettesei közül az első az *edge_id*, második a *neighbour_id* mezőnek felelnek meg, és mindkettőjüket FK megszorítás köti az *Edge id* mezőjéhez.

Fogalmazhatunk úgy is, hogy a *Map* az az *Edge* önmagával vett kapcsolótáblája. Minden él kapcsolódhat több élhez és ugyanaz az él elérhető lehet több élből is. (Kapcsolótáblákról bővebben a 3.2.2 *Menu és Inventory* c. alfejezetben.)

Menu

A *Menu* tábla központi szerepet tölt be az adatbázisban. Egyedi *name* mezője mellett rendelkezik PK azonosítóval is, mivel más táblából (*Food_Orders*) mutatnak soraira hivatkozások. Egy menü árát a *price* mező tárolja, ennek pozitív számnak kell lennie, amit a *model* réteg fog validálni. Egy menühöz tartozik több alapanyag, amelyek tárolását a *Menu_Ingredient* nevű *join table* (kapcsolótábla) végzi el.

A kapcsolótábla feltöltéséhez szükség van *Ingredient* táblabeli rekordokra is, amelyek létrehozása egy egyedi név (*name field*) kötelező megadásával történik. Emellett egy *Ingredient* hivatkozhat egy egyedi nevű *Allergy*-re is, de ez opcionális. (*Allergy* létrehozása szintén az előbb említett módon történik). Visszatérve a kapcsolótáblához, annak szerkezete a következő: egy rekord hivatkozik egy *Ingredient*-re és egy *Menu*-re, emellett a kapcsolatuk mellé tárol egy nemnulla mennyiséget is (*quantity*). A kapcsolótábla használatát megint csak az alapanyag és menü közötti reláció természete indokolja, mivel egy menü több alapanyagból is állhat, és egy alapanyag több menü elkészítéséhez is használatos lehet. Hasonló elvet követ a *Car_Ingredient* tábla is.

Food Orders

Ez a tábla a leadott rendeléseket tárolja. Értelmszerűen hivatkozik egy *User* és egy *Menu* rekordra, ez is egyfajta kapcsolótábla az előbbi két tábla között. Ami pedig a reláció mellé tárolt információ, az annak keletkezésének idejét rögzítő időbélyeg (*ordered_at timestamp*). Amiért e tábla rendelkezik kulccsal, annak indoka, hogy minden rekordhoz tartozik legfeljebb egy rekord az *Order_Delivery* táblából, azaz annak rekordjai a *Food_Orders* táblát hivatkozzák. Viszont a program futása során nem történhet meg, hogy két különböző kiszállítás ugyanarra az egy rendelésre hivatkozzon.

Egy *Car* rekordot is hivatkozik a kiszállítás, a hivatkozott autóra pedig tranzitív módon tovább vihető az állítás: nem létezhet, hogy egy rendeléshez hozzacsatolható legyen több autó is. Az *Order_Delivery* a *Food_Orders* táblához hasonlóan tartalmaz egy *timestamp* típusú mezőt. A sorok esetleges összecsatolása után, a rendelések *id* mezője alapján, két bélyeg együttes olvasása ad számukra igazi értelmet: különbségük megadja, hogy mennyi időt vett igénybe a kiszállítás.

Car

Az előző alfejezetben már említésre kerülő *Car* tábla, a szimuláció során használt autókat (lásd 3.5.4 Szimuláció c. alfejezet) tárolja. *Name* mező helyett itt a *license* tárol az autóhoz egy nemüres, egyedi rendszámot. *Id* PK mezőjének hivatkozásai közül említésre került már a *Car_Ingredients* kapcsolótábla-beli előfordulása, az előző bekezdésben pedig szó volt a *Order_Delivery* táblában betöltött szerepéről. Emellett még találunk rá hivatkozást az *Inventory* táblában, amely felfogható egy második számú *Car_Ingredients* táblának, erről bővebben írok a következő bekezdésben. Találunk még egy *User* tábla referenciát, amely lehet *null*, azaz üres értékű hiszen nem ül mindig sofőr egy autóban, nincsen véglegesen hozzárendelve.

Inventory

Utoljára tárgyalt tábla az *Inventory* nevezetű. Amint arról már szó esett, funkcionalitásának szoros kapcsolata van a *Car_Ingriedient* táblával. Szintén autó és élelmiszer kerül hivatkozásra egy sorában, időbélyeggel, mennyiséggel és a hozzátartozó kiadást tároló mezővel ellátva. Ugyan nem kimondottan kapcsolótábla, de nem tartozik hozzá PK mező. Sorai az autó, alapanyag, időpont rendezett hármasával azonosíthatóak egyértelműen. Egyik mező sem lehet üres. A kiadás lehet 0, amennyiben a bejegyzett élelmiszermennyiség nem vásárlásból származik, hanem a *Car_Ingriedient* táblával folytatott tranzakcióból. (További, a tábla sorai közötti összefüggésekről, a *Felhasználói útmutató*, 2.4.3. *Raktár- és rendelésselőzmények* c. fejezetében.) Tehát, a program működéséből eredően leszűrhető még az a következtetés is, miszerint, ha a *Car_Ingriedient* táblában változás történik, akkor abban a rekordban szereplő, autó, alapanyag kettessel kell, hogy létrejöjjön az *Inventory* táblában is egy bejegyzés, 0 kiadással.

ORM leképezés

Az ORM szoftver amiatt lett kitalálva, hogy az adatbázis rétegében tárolt adatok táblái megfeleltethetők legyenek az objektum-orientált programozásból (*object oriented programming*^{vi}) ismert osztályoknak, ezzel entitásokat képezve. Tehát így, hogy egy tábla rekordjaira úgy tekinthetünk, mint objektumokra, az ORM hidat teremt *persistence* réteg (adatok tárolása valamilyen formában) és az architektúránkban lévő *model* réteg között.

Hibernate specifikus utasítások

Az entitásokhoz tartozó *service* funkcionális osztályokba olyan függvényeket kerülnek, amelyek az adatbázissal kommunikálnak, új rekordok entitásokon keresztül történő létrehozása, lekérése, frissítése, és törlése végett. Ahelyett viszont, hogy ezekben a függvényekben általunk írt natív SQL kód szerepelne, a *Spring* webszerver ORM feladatait végző *Hibernate*, olyan interfészt biztosít minden entitáshoz, amelyek használatával nekünk nem kell közvetlenül az adatbázissal kommunikálni^{xii}. A *CrudRepository* biztosít minden fontosabb függvényt, egy *T* típusú, sorait *ID* típus alapján azonosítható entitás táblájának hozzáférésére. (*Create*, *Read*, *Update*, *Delete* feladatkörbe tartozó parancsok interfésze).

- `<S extends T> S save(S entity);`
- `Optional<T> findById(ID primaryKey);`
- `Iterable<T> findAll();`
- `Long count();`
- `void delete(T entity);`
- `boolean existsById(ID primaryKey);`

A függvénynevek árulkodóak azok funkcióit illetően. A visszatérési értékek kapcsán a *findById(...)* nevű függvény szorul magyarázatra esetleg. Mivel egy kulcshoz nem feltétlen tartozik rekord a táblából, megeshet, hogy a lekérdezés nem ad vissza rekordot, amiből entitás objektuma létrehozható, így az *null* értékű marad. Az *Optional* osztály az ezzel kapcsolatos problémákat segít áthidalni, például *orElse(...)* metódusával megadhatjuk, hogy milyen *T* típusú értékkel térjen vissza, ha a függvény *null*-t adna.

Komplexebb *query*-k (adatbázis lekérdezések) esetében is segít az ORM. Egy adott entitáshoz tartalmazott függvényei a következő módon épülnek fel: **alany** (pl.: `find`, `exists`, `delete`) + **állítmány** (Az állítmány akár egy további almondat is lehet, amely alanyai az entitás táblájának mezőiből származtatott *property*-k, *query* specifikus kulcsszavakkal összefűzve) példa:

```
List<User> findByNameAndRoleByRoleId(String name, Role role);
```

Entitások kulcsai

Amint láthattuk az adatbázis-manipulációs függvényeket biztosító interfész létrehozásánál nem csak az entitás típusa generikus paraméter, de a kulcsé is. Ebből két dolog szűrhető le. Először, hogy kulcs *property*-vel minden entitás rendelkezik, másrészt létezik, amikor a kulcs nem integer. Tehát valami módon mindenképpen kell, hogy egyértelműen azonosítsuk egy tábla sorait.

Amikor egy táblát létrehozunk az adatbázisban, nincs kikényszerítve a kulcs jelölése. Azaz nem kell tudnia a táblának, hogy egy rekordja milyen mezőtől, vagy mezőkombinációtól válik egyedivé (lehet, hogy nem is az). Viszont a *Hibernate* elvárja ezt. Amely tábla rendelkezik PK mezővel, ott egyértelmű az azonosítás, a kulcs természeténél fogva. Viszont egy kapcsolótábla rekordjai, mint a *Car_Ingredient*, nem feltétlen vannak ellátva kulccsal, így nekünk kell definiálni a mezőkombinációt, kompozit kulcs^{xiii} formájában. Ez nem jelent mást, mint egy osztály formájában egységbe zárni a kulcsként felfogható mezőkombinációt, az utóbbi tábla esetében a *carByCarId* és az *ingredientByIngredientId* mezőt. Ez alapján fogja keresni az entitásnak megfelelő rekordot az ORM. Az első találattal tér vissza, vagy mentés esetén azt írja felül. A program működése szempontjából kényelmes, hogy a *save(...)* metódus a *Car_Ingredient* táblában egyszerre beszúr, vagy ha már létezik rekord a *Car* és *Ingredient* alapján, akkor frissít, pontosabban frissíti a kompozit kulcshoz tartozó mennyiséget.

Objektumok közötti reláció

Az ORM szintén nagy segítséget nyújt a táblák közötti relációk definiálásában. Ha a *query* függvény definiálásának példájára visszatekintünk, megfigyelhetjük, ahhoz, hogy név és szerep alapján lekérdezett *User*-ek listáját úgy kapjuk meg, hogy a függvény *role* argumentumának egy *objektumot* adunk. Tehát azt, hogy amilyen szerepű felhasználót kell keresnie, nem egy *int* típusú kulcsból, hanem egy konkrét *Role* példányból határozza meg. Miért lehetséges ez? Az ORM egy rekord példányosítása során elvégzi el azt a lépést, hogy az esetleges FK kapcsolatok alapján hozzárendel egyet-egyet azokból a példányokból, amikkel kapcsolatban áll az objektum. (Az *object*-ek típusaként szolgáló *class*-ok adatbázis séma alapján történő automatikus generáltatásakor, a relációs példányt tároló *property*-t, *roleByRoleId* néven hozza létre az IDE.)

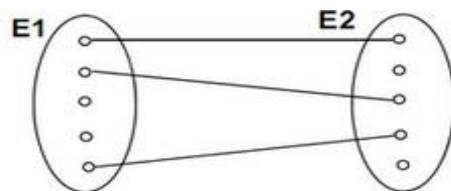
Az adatbázis-beli táblák kapcsolatait az ORM három különböző típusba sorolja^{xiv}:

- *One-to-One*: egy rekord kulcsára legfeljebb egy másik táblának egyedüli rekordjából történik hivatkozás. Például a *Car* egy példányának egy *User* példányra történő hivatkozása. Vagy az említett *Food_Order* tábla bármely rekordjára is csak egy *Delivery* rekord hivatkozhat.
- *One-To-Many*: egy kulcsra több helyen is történik hivatkozás, gondoljunk például a *User* rendelésre: több rekord is hivatkozhatja őt a *Food_Order* táblából. Ekkor egy *User* példányban egy *Collection* adatszerkezetben érhetjük el a vele relációban álló rendelések példányait.
- A *Collection*-ben tárolt példányok szemszögéből *Many-To-One* a kapcsolatuk típusa a *User*-rel, innen is találunk rá visszamutató referenciát. Mivel ez az előző kapcsolat megfordítása, így nincs külön számontartva.
- *Many-To-Many*: Két tábla a rekordjai többszörösen hivatkozzák egymás. A *Menu* és *Ingredient*, vagy *Car* és *Ingredient* tábla között fennálló kapcsolat típusa is ilyen. A kapcsolótábla rekordjainak explicit módú, entitásnak való megfeleltetése kerülhető el ezzel. A jelen szoftver implementálása során nem használt ilyen annotáció mert a kapcsolótáblák rekordjai is példányosításra kerülnek, ezekkel pedig *One-To-Many* kapcsolatban állnak az említett táblákhoz tartozó példányok. Az entításokban így, a relációban álló sorok példányain kívül, más mezők is szerepelhetnek, mint például a mennyiség.

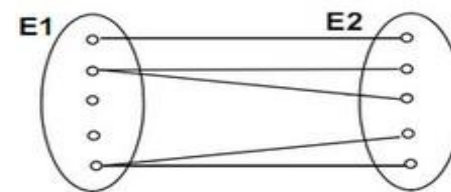
Az entítások *singleton* példányainak egymás közötti referálása önmagában végtelen rekurzióhoz vezetne a példányok szerializációja során, erről a jelenségről és megoldásáról bővebben a 3.5.1 *Szerializáció* című alfejezetben írok.

A típusokkal megegyező annotációk használatával jelölhetőek a *Hibernate* számára, hogy miként próbálja létrehozni a példányokat. Fontos érteni, hogy az adatbázis nem ad garanciát, hogy valóban teljesülnek a *Hibernate* magasabb szintű korlátozásai, vagy arra, hogy natív módon létrehozott adatok az adatbázisban ne rontsák el azt. Így az sem biztos, hogy a szerver entitás példányosítására tett próbálkozása ne eredményezzen hibát. Az alábbi ábra halmazok formájában demonstrálja, hogy mikor teljesül egy megkötés az adatbázisban lévő relációkra. (E1 és E2 halmazok adatbázisbeli megfelelője bármely két tábla, amely rekordjai között kapcsolat áll.)

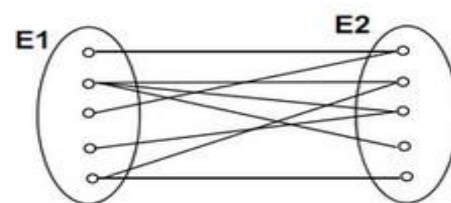
**One to one
1:1**



**One to many
1:M**



**Many to many
M:N**



16. ábra - Kapcsolat típusok^{xv}

Validáció

Az ORM magasabb szintű megkötései természetesen arra is tökéletes, hogy beszűrődjön adatot validáljunk. A relációs mezők mellett natív típusokra is léteznek annotációi a *Spring Validation* könyvtárnak, melyek segítségével megszabható a *string* vagy *integer* típusú mező nagysága^{xvi}. Ennek használatára találunk példát a programban, bármelyik mennyiség mező nemnulla, pozitív léte kapcsán, vagy az *Ingredient price* mezője korlátozásánál.

Mezőket jelölhetünk *null* és *nonnull* annotációval. A *null* programban való használatára a deszerializációval létrehozandó entitások esetében kerül sor, amikor nem az adatbázis egy rekordja alapján példányosítunk. Ennek megfelelően a validáció kérhető a perzisztálástól különböző pillanatban is. A *NonValidatedOnPersistTime* validátor csoport ebből a célból lett létrehozva. A csoport szóhasználat, itt arra utal, hogy különböző mezőkre írt validátorokat különböző csoportok tagjaként tartja számon a *Spring*. Amennyiben nem adunk meg explicit csoportot, a validátor a perzisztálás pillanatában lép működésbe. Az új csoport létrehozásával, és a validátor annak tagjaként való megjelölésével, ez elkerülhető. Így a *null* tartalom csak az objektum deszerializációból való létrehozásnál lesz kikényszerítve. Ennek jelentőségéről bővebben a 3.5 *Végpontok c.* című fejezetben. Jelen fejezet zárásaként, egy *package* diagramban foglalom össze a fenti szempontok alapján, az ORM leképezést megvalósítótására létrehozott entitások osztályait. Ahol szükség volt kompozit kulcs használatára, ott táblanév és “PK” szuffix a névhasználat.



17. ábra - Package diagram^{xvii}

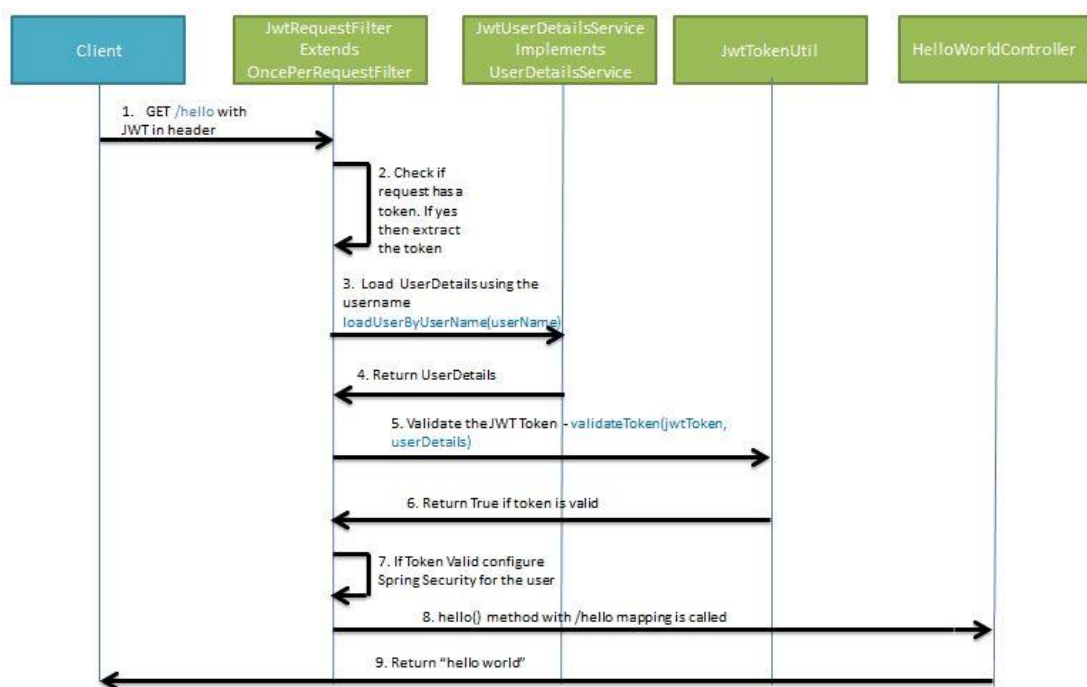
Autentikáció

Az autentikáció már a Felhasználói dokumentáció során többször említésre került. Amíg azonban a korábbi fejezetben csak a különböző szerepkörök által elért adatforrások voltak a középpontban, addig ebben a fejezetben bemutatásra kerül az is, hogy a szerver hogyan azonosít egy felhasználót, illetve hogyan ad számára időkeretet, amíg újabb autentikáció nem lesz szükséges.

A témakör szempontjából fontos definiálni, hogy milyen szerverről beszélünk. Programom megvalósításához a *stateless*^{xviii} szerver (állapotot nem tároló) használata mellett döntöttem, mivel nem szükséges a szerveren tárolni semmilyen felhasználói munkamenet specifikus adatot. Egy *stateful* szerver esetével ellentétben így nem kell törődni a *Spring* működését szolgáló *bean*-ek^{xix} életciklusának kezelésével a *user session* szerint. Ahelyett a felhasználó minden kérés során csatol egy személyazonosságát bizonyító token-t. A token szimpla dekódolásával lesz a felhasználó azonosítva, így a kérések kiszolgálása is gyorsabb lesz, jobban skálázható nagyobb felhasználó számra.

Hogyan történik egy token generálása^{xx}? Amikor a felhasználó egy token kiadását kéri, vagy hétköznapi megfogalmazásban bejelentkezni próbál, felhasználónév és jelszó párosát küldi egy *POST request*-ben a bárki számára elérhető *"/authenticate"* végpontra. A programnak meg kell vizsgálnia, hogy a bejelentkezési kísérlet sikeres-e. Átadja az adatokat a *BasicConfiguration* osztályban felkonfigurált *authenticationManager* egy példányának, ami lekéri az adatbázisból a regisztrált felhasználó adatait. (Amennyiben nemlétező email címet kapott, a bejelentkezési kísérlet értelemszerűen sikertelen.) Összehasonításra kerül a kérésben szereplő jelszó *hashed* változata, és az adatbázis lekérdezéséből kapott jelszó, amennyiben egyezik a kettő, a felhasználó sikeresen autentikálta magát. A *JwtTokenUtil* osztály segítségével kiállít a program a felhasználó nevére egy *JWT (Json Web Token)* token-t, amely a létrehozása időpontjától számított 5 óráig lesz érvényes. A *GET* kérés válaszüzenete ez a token lesz, amit innentől a kliens böngészőjének a felelőssége tárolni, hogy azt minden jövőbeli kéréshez csatolhassa. (Erről bővebben a 3.7 *Kliensoldali megoldások c.* fejezetben.)

A következő fontos lépés a token validálása minden további, védett végpont eléréséhez. Ebben A *JwtRequestFilter* osztály játszik fontos szerepet. Minden beérkező kérés további feldolgozása előtt az osztály *doFilterInternal(...)* metódusa fog lefutni, amely *valid* token esetében hozzáadja az abból kiolvasható felhasználót a *Spring Security* kontextusához, így az eredeti kéréssel elérni kívánt védett *endpoint* megállapíthatja, hogy a felhasználónak van-e jogosultsága adatot lekérni tőle. (Amennyiben a token *invalid* vagy *expired*, azaz lejárt státuszú, a nem kerül felhasználó beállításra a kontextusba és minden védett végponttól el lesz utasítva, 401 – *UNAUTHORIZED* státusszal.) Íme egy példa egy fiktív “/hello” *endpoint*-ra érkező kérés tokenének validálására.



18. ábra - Token validálás^{xxi}

Végpontok

Mivel a kliens végpontjain keresztül éri el a szervert, ezért kiemelt fontosságú, hogy listába szedjük őket. A jogosult kéréseket a szerver valamelyik *Controller* osztály kezeli, annak egy metódusa a kérésre a *View* rétegben tartozó adattal tér vissza válaszul. Ez a fajta szerver-kliens kommunikáció megfelel a *decoupling*^{xxii} elvének, azaz anélkül tudnak sikeresen együttműködni, hogy bármilyen szinten függésben lennének egymás megvalósításától. A kliens csak *API*-ként tekint a szerverre, annyiban függ tőle, hogy egy bizonyos formáját kell várnia a *View* réteg reprezentálásának, ami az alkalmazásom esetében a *JSON (JavaScript Object Notation)* adatszerkezet lesz.

A végpontok egyedi módon jelölnek adatforrásokat, alágaikkal azonosító szerint mutatnak egy-egy objektumra. Az objektummal történő esetleges változtatás formáját a kérés típusa jelöli. Az alkalmazásban is használt típusok az alábbiak, amelyek különböző válasszal reagálnak:

- **GET:** azonosító alapján történő adat lekérése a szervertől
 - 200 - *OK* státuszkód alatt választ küld a szerver, benne a kért adattal
 - 404 - *NOT FOUND* státuszkódú alatt érkezik válasz, amennyiben a kért adat nem létezik
- **POST:** adatküldés szervernek, annak eltárolása céljából
 - 201 - *CREATED* státuszkóddal a tárolt adat, mint válasz
 - 400 – *BAD REQUEST* státuszkóddal küld választ, ha a tárolni kívánt adat valamiért nem menthető, például már korábban regisztrált emailcím alatt akar új felhasználó fiókot létrehozni
- **PUT:** szintén adatot küldünk a kéréssel, azonban egy azonosítóval jelölt végpontra, így az azonosítóhoz tartozó rekord az adatbázisban felülírásra kerül az új adatokkal
 - 201-es kóddal a frissített adat
 - 400-as kód, ha nem hajtható végre a kért változtatás, mert megszorításba ütközik
 - 404-es kód, ha az azonosítóval jelölt adat nem létezik
- **DELETE:** az azonosítóval jelölt végpontra beérkező kérés hatására a szerver törli az azonosítóhoz tartozó adatot
 - 200-as válasz, ha a törlés megtörtént
 - 404-es kód, ha az azonosítóval jelölt adat nem létezik

Szerializáció

Ahogy az már korábban említésre került, a kliens JSON formátumú válaszokat vár/kap. Ezek az entitások szerializálásának végeredményeképp jönnek létre. A program során használt *Jackson Databind* könyvtár segítségével, a *Controller* osztályok metódusainak visszatérési értékét automatikusan szerializálja a program, mielőtt HTTP üzenetbe csomagolja azt. Pontosabban egy objektumnak azokat az adattagjait éri el, amelyekhez tartozik *getter* és nem blokkjuk egyéb megkötéssel (ellenkező, deszerializációs irányból *setter*).

Miért szükséges néhány esetben blokkolni az elérést? A korábban említett *Hibernate* által elvégzett, kapcsolatban lévő entitások automatikus összekapcsolása végtelen rekurzióhoz, a futás során *stack overflow*-hoz vezetne. Így fontos tudatni a *Jackson* algoritmusával, hogy melyek az aktuálisan sorosított objektumára visszamutató referenciák. Erre vegyük példának a *Menu* és *Ingredient* entitás esetét. Egy *Menu* példányt szeretnénk *JSON* formátumúvá alakítani és szükséges, hogy tudjuk annak összetevőit, hiszen a szakács szerkesztheti őket, a vásárló pedig megtekinti a rajtuk keresztül felmerülő allergéneket. Ekkor a végtelen rekurzió elkerüléséhez jelezzük a *JsonManagedReference*^{xxiii} és *JsonBackReference* annotációk közös használatával, hogy a *menuIngredientsById* adattag elemeinek bejárásakor az elért objektumok *menuByMenuId* adattagján keresztül visszajutna az eredeti *Menu* példányba. A szerializáció kiinduló iránya szerint először érintett problémás *property*-re használjuk a “kezelt referencia” jelölést és az általa elért objektumban pedig a “visszamutató referencia” annotációt. Az irány megkülönböztetése amiatt is hasznos, hogy az *Ingredient* irányából induló esetleges sorosítás esetében blokkolja a folyamat további rekurzióját, ha *BackReference* annotációval találkozunk. (Persze emiatt nem lenne alkalmazható ez a módszer, ha valamiért le akarná kérdezni a kliens, hogy egy bizonyos összetevő mely menükben szerepel.) További esetekben, ha a lekérés során nincs szükség az adott *property* által hordozott információra, használhatjuk önmagában a *JsonIgnore*^{xxiv} annotációt és nem ütközünk problémába. A programban az egyszerűsége miatt sok helyen használtam.

A szerializáció ellentéte a már említett deszerializáció, amely használatára akkor kerül sor, amikor a *JSON* formátumú *object*-et akarjuk entitássá konvertálni. Tehát egy beérkező kérés feldolgozása során nem teljesen az adatbázisból nyerjük az objektumot, amellyel végezzük a műveletet, hanem részben a kérés során kapott adatból konvertáljuk át azt. A létrejövő entitás ekkor is validálásra kerül, ahogy az szintén említve volt az ORM funkcióit tárgyaló 3.3.4 *Validáció c. alfejezetben*. *Invalid* adat fogadását követően az aktuális *Controller* automatikusan megszakítja a kérés kiszolgálását, 400-as státuszkóddal válaszol a kérésre, még mielőtt bármilyen *service* metódusát hívná.

Kérések

Az alábbi diagrammon sorra veszem, hogy milyen végpont milyen kérést vár, milyen autentikációs szint megléte mellett. (A faszerkezet törzséből indul a főútvonal, majd annak alsóbb szintjein, a levelek felé haladva a főútvonal lehetséges folytatásai vannak felsorolva.)



* az *owner* nem egy konkrét létező szerepkör, hanem azt a megszorítást jelöli, hogy az *admin*-on kívül csak az objektum létrehozója férhet hozzá. (Ez *id property* alapján kerül ellenőrzésre.)

*** a *roleByRoleId* adattag csak az *admin* jogosultsággal rendelkező kérések során adható meg. Magyarázata a 2.1.1 *Regisztráció c.* fejezetben.

*** a *userByUserId* adattag itt kötelező jelleggel kell, hogy *streetByStreetId* és *houseNo property*-ket tartalmazzon. Erről a 2.1.2 *Rendelés c.* fejezetben leírt folyamat gondoskodik.

**** az eredetitől különböző *license* tartalommal küldött *PUT* kérés küldése csak az *admin* számára engedélyezett

Ahol paraméter van megadva a *GET* kérés mellé, ott az csak szűrési feltételként használt, tehát anélkül is értelmes választ ad a szerver. Ezen kívül a *body* adattagjai közül a zárójelben lévők megadása is opcionális, mivel a kérésből szerializált entitások mögött rejlő mezők értéke is lehet *null*. Minden *object* típusú *body* adattag küldésénél a szerver annak csak azonosító mezőjét tartalmazó példányát várja, a csillagozással jelölt *user* kivételével. Az *object*nek megfelelő, teljesértékű entitás példányosítását a szerver végzi, ezzel elkerülve, hogy a szükségtelenül sok klientsől nyert adat inkonzisztenciát okozhasson. Hasonló megfontolásból, a nem felsorolt adattagok küldése tilos. Tehát a szerver arra is validál, hogy a kérésből deszerializált entitások néhány adattagja *null* maradjon. Amennyiben nem így lenne, észrevétlen hibákat ejthetnénk néhány kéréssel. Pl.: a 7-es menühöz való alapanyag-kötés céljából *PUT* kérést küldünk a *"/menu/7/ingredient"* végpontra. Tegyük fel, hogy teljes szerializált formáját küldjük a *MenuIngredient* entitásnak, tehát redundánsan a menüt jelölő *Menu* adattagot is. Ha a kérés törzsében a menü a 7-es azonosítótól különbözik, akkor végeredményként az eredeti menünek a végpontján keresztül meg tudtuk változtatni egy attól különböző menü összetevőlistáját.

Ugyan annotációval az már jelölve nincsen, de a *POST* kéréseknél az azonosító küldését is szűri a *Service* algoritmusai, megakadályozva ezzel azt, hogy mentés néven felülírás történjék egy másik, már létező rekord mezőinél. (Ennek eshetősége már említve volt a 3.3.1. *Hibernate specifikus utasítások c.* fejezetben.)

Websocket végpontja

A szerver egy *websocket* végponttal is rendelkezik, amelyre a kliens a működése során akkor csatlakozik, amikor rendelést ad le és miután azt elküldte, folyamatos visszajelzést vár a futár helyzetéről. Az utóbbi elvárás miatt esett a választás az ilyen típusú *socket* alapú kommunikációra, mert ez lehetőséget ad egy csatorna folyamatos nyitvatartására, amelyen a keresztül a szerver rendszertelen időközönként tud üzenetet küldeni. Ha a kliensnek kellene a szervert valamilyen *polling rate* (faggatás) szerint kérésekkel terhelni, a rendelés helyzetét illetően, akkor az vagy túl ritka lenne ahhoz, hogy pontos visszajelzést kapjon a kiszállításról, vagy gyakori, de feleslegesen terhelő a szerver számára. Ehelyett lehetővé tesszük, hogy mindkét fél képes legyen bármikor üzenetet küldeni a másiknak, addig amíg valamelyik nem kezdeményezi a csatorna bezárását^{xxvi}. A kliens tehát egy `"/delivery"` nevű védetlen végpontjára kapcsolódik a szervernek, amittől kezdve a szervertől kisebb időközönként egy *heartbeat* nevezetű üzenetet kap, amire ő nyugtázza, ha megkapja, így nyitva tartva a kommunikációs folyosót, amin keresztül mindkét fél biztosítja a másikat, hogy jelen van. Emellett minden kliens feliratkozik egy saját chatszobába a szerverrel, azzal az előfeltételezéssel, hogy a szerver is ide fog üzenetet küldeni nekik. Ez valójában egy köztes, *mediátor* szerver lesz. Az, hogy egyediként legyenek megkülönböztetve az útvonalak, a megvalósítás során a felhasználó azonosítójának segítségével lett megoldva, tehát kliensenként a szerver a `"/delivery/{id}"` szobán keresztül fog kommunikálni, ahol az *id* az a *user*, akit éppen kiszolgál.

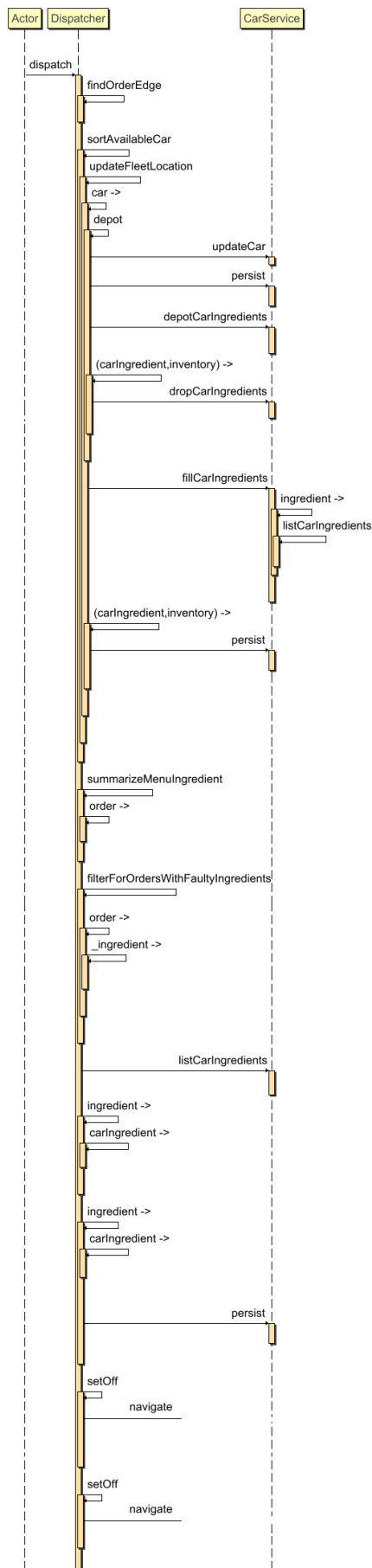
Szimuláció

A szimuláció számára egy végpont van létrehozva, az `"/inventory/reload"` út alatt. Erre küldhető kérés, ami hatására végrehajtódik lényegében a szimuláció osztály rekonstruálása. Azonban nem ekkor történik a szimuláció konstruktorának első futása, és nem ekkor végzi az első műveletet az *Inventory* táblával.

A szimulációs folyamatokért felelős *Dispatcher* osztály ugyanúgy *Service*-ként van megjelölve, mint bármelyik, entitáshoz kötődő, *service* szolgáltatásokat nyújtó osztály. Ennek gyakorlati lényege, hogy a *dispatcher* életciklusáért ugyanaz a *Application Context* nevezetű *IoC* (*Inversion of Control*) konténer felel, mint ami elvégzi bármilyen más *Autowired*-vel megjelölt *field* komponensének injektálását az aktuális osztályba. Minden komponenst ott példányosít először a *singleton* mintának megfelelően, ahol először hívásra kerül. A szimuláció a program futása során egész végig be van töltve a

memóriába, mivel annak példánya, a *Main class* szerepét betöltő *ServerApplication* osztályába, kontextus konténeréből injektált (*autowired*).

Az osztály konstruktorának lefuttatásával feltöltjük az *Inventory* táblát a készlet hiányait pótló mennyiségekkel, az aktuálisan elérhető alapanyagokból. Emellett hívhatóvá válik a *dispatch(...)* nevű függvény, a kiszállítási folyamatok elindítására a beérkező rendelés kapcsán. Az utóbbi módszernek egy folyamatábrán vannak szemléltetve a részletei (a folyamatábra^{xxvii} erősen egyszerűsített és csak a *CarService* osztályig elérő hívásokat mutatja):



20. ábra - Dispatcher folyamatábrája

A *findOrderEdge* lépés beazonosítja, hogy a rendelés házszáma melyik, az utcanévhez tartozó élen van.

A kiszállítást végző autók memóriában való tárolása, vagy ahol megszűnik a sofőr, ott ürítése, a folyamat elején történik - *sortAvailableCars*. Erre az érv, hogy ilyenkor szükséges *update*-elni a rendszer számára, hogy milyen flotta lesz elérhető a kiszállításhoz, mielőtt még távolság szerint rendezzük. Az *updateFleetLocation* az újonnan elérhetővé vált autó készletének táblájába ekkor szűrja be a rekordokat a feltöltésről - *fillCarIngredients*, és a memóriában pedig eltárol hozzá egy helyzetet, ami a pizzéria-töltőpont lesz ideiglenesen. Ha üríteni kell az autót - *depot*, a fent említett okokból a memóriából, a készlet táblájának rekordjait törli a *depotCarIngredients*. (A *Dispatcher* törlésekor egy *PreDestroy* metódus is elvégzi az utóbbit, és a sofőröket is kiiktatja.) A *summarizeMenuIngredient* egy listába szedi a teljes mennyiséget azokból az összetevőkből, ami szükséges, hogy leszállítható legyen a rendelés.

A *filterForOrdersWithFaultyIngredient* azért lényeges, hogy ne juthasson holtpontra a szimuláció az 2.5.1. *Alapanyag feltöltés c.* fejezetben tárgyalt probléma miatt. Végül bejárja az algoritmus a már rendezett, elérhető autók listáját és kiválasztja az első olyan autót, ami megfelel az összesített mennyiségekben megszabott alapanyag kritériumnak. Az első ilyen autót a *setOff* függvény egy *task*-nak adja át, amelyben elnavigálja a vásárlóhoz, ahol

kézbésít az autó, helyzetét frissítve a memóriában. Amíg folyik a navigáció, addig a helyzet nem frissül folyamatosan, helyette *null* lesz. Minden más autót olyan *task*-nak ad át, amely a telephelyre navigál és újratölt. A *task*-ok végrehajtását konkurens módon, új szálon hajtja végre egy *executorService*, így a *dispatcher* már azelőtt újra futhat, hogy teljesülne a legutóbbi kiszállítás. Itt újra említeném még az aktuálisan úton lévő autók helyzetét. Az, hogy *null* értékű lehet ez, azért is fontos, mert egy *lock* módjára jelzi a következőként futtatandó *sortAvailableCars* algoritmusának, hogy ez az autó ne legyen még újra elérhető ezen a szálon.

Végpontok tesztelése

Az alábbi táblázatban sorra szedem az API végpontjaira írt teszteseteket. Ezek működéséhez elengedhetetlen a *test_data* adatcsomag betöltése. Első teszteset az autentikációról szól. Ezt követően annak folyamata nem lesz külön részleteként említve, a kéréshez csatolt *{szerepkör}* - token megelégszik, egy, abba a szerepkörbe tartozó felhasználó korábbi sikeres autentikációját. A színkódolás a kérések típusának jelölésére megegyezik a típusok ismertetése során használt verzióval. A kéréseknél előfordulhat, hogy a törzs *object* típusú mezőinél az *id*-n kívül más is kitöltött, a közérthetőbb üzenet miatt.

Végpont	Üzenet	Elvárt
/authenticate	"email": "admin@domain.com", "password": "secret"	Token válasz
/authenticate	"email": "admin@domain.com", "password": "invalid"	401
/authenticate	"email": "invalid@domain.com", "password": "password"	401
/role/add	"name": "role"	401
/role/add	Admin-token, "name": ""	400
/role/add	Admin-token, "name": "customer"	201
/role/add	Admin-token, "name": "customer"	400
/role/add	Admin-token, "name": "driver"	201
/role/add	Admin-token, "name": "chef"	201
/role/99	Admin-token, "name": "cook"	404
/role/4	Admin-token, "name": "admin"	400
/role/4	Admin-token, "name": "cook"	201
/role/99	Admin-token	404
/role/4	Admin-token	200
/user/register	"email": "customer@domain.com", "password": "secret", "name": "John Doe", "roleByRoleId": {"id": 2, "name": "customer"}	201

/user/register	"email": "jane.doe@domain.com", "password": "secret", "name": "Jane Doe", "roleByRoleId": {"id": 2}, "streetByStreetId": {"id": 3}, "houseNo": 37	201
/user/register	"email": "customer@domain.com", "password": "secret", "name": "Duplicate Doe", "roleByRoleId": {"id": 2}	400
/user/register	"email": "invalid.email", "password": "secret", "name": "John Doe", "roleByRoleId": {"id": 2}	400
/user/register	"email": "name@domain.com", "password": "secret", "name": "John Doe"	400
/user/register	Admin-token, "email": "driver@domain.com", "password": "secret", "name": "Driver Doe", "roleByRoleId": {"id": 99, "name": "invalid"}	400
/user/register	Admin-token, "email": "driver@domain.com", "password": "secret", "name": "Driver Doe", "roleByRoleId": {"id": 3, "name": "driver"}	201
/user/1	Customer-token, "email": "jane.doe@domain.com", "password": "secret", "name": "Mrs Jane Doe", "roleByRoleId": {"id": 2}	401
/user/3	Customer-token, "email": "jane.doe@domain.com", "password": "secret", "name": "Mrs Jane Doe", "roleByRoleId": {"id": 2}	201
/user/3	Customer-token, "email": "customer@domain.com", "password": "secret", "name": "Mrs Jane Doe", "roleByRoleId": {"id": 2}	400
/user/3	Customer-token, "email": "jane.doe@domain.com", "password": "secret", "name": "Mrs Jane Doe", "roleByRoleId": {"id": 1, "name": "admin"}	400
/user/1	Customer-token	401
/user/3	Customer-token	200
/car/add	Admin-token, "license": "asd-123"	201
/car/1	Driver-token, "license": "asd-123", "userByUserId": {"id": 5}	201
/menu/add	Admin-token, "name": "pizza", "price": 100	201
/menu/1 /ingredient	Admin-token, "ingredientByIngredientId": { "id": 99, "name": "invalid"}, "quantity": 10	404
/menu/1 /ingredient	Admin-token, "ingredientByIngredientId": { "id": 1, "name": "flour"}, "quantity": 10	201

Kliensoldali megoldások

Alkalmazásomnak kliens részét a Vue^{xxviii} nevezetű JavaScript keretrendszer használatával készítettem el. Progresszív webalkalmazás lévén egy mobilapplikáció kényelmét biztosítja. Az állapotok átmenete között minimális a várakozási idő, mivel az oldal sosem egészében, hanem csak komponensenként frissít, általában animáció kíséretében. Az oldalváltásokhoz, felugróablakokhoz az erőforrások betöltése a háttérben, aszinkron módon történik. Ezzel jelentősen javul a *user-experience* (felhasználói élmény). Egyetlen kompromisszuma, hogy az első betöltés hosszabb, mivel ilyenkor sok, az oldal működtetéséhez szükséges scriptkód kerül lekérésre.

Adatelérés

Az oldalhasználat közben az adatok lekérése a szervertől az *Axios* könyvtár segítségével történik. A *PizzaServerAPI* fájlban van a szervernek címe eltárolva és az alapján példányosítva kapcsolat. Minden, az osztályból exportált függvény ezen a kapcsolaton keresztül éri el a szerver egy-egy végpontját, és küldi el rá az argumentumként átadott törzset.

Szintén itt kell gondoskodni róla, hogy a bejelentkezésnél tárolt token-t a kérések *header* részéhez csatoljuk. A token tárolására a böngésző *cookie* tárhelyét választottam. Emellett azok az érvek szóltak, hogy az oldalhoz tartozó tárterület *javascript* kódból könnyen kezelhető és oldalfrissítéstől függetlenül őrzi tartalmát. Egy *cookie* automatikus ürítésére csak annak lejáratakor kerül sor, ha van beállítva lejárat idő. Mivel JWT token tárolásáról van szó, ami kiállítását követően érvényességét veszíti egyszer, így kényelmes, hogy a tárolására is megadható egy időmaximum. Természetesen a *user* bármikor manuálisan ürítheti a tárhelyet a kijelentkezés parancsával. A tárhelyen tárolva van még egy *userId* *cookie*, amit felhasználunk a „/user” végpontok hívására, és egy *role*, aminek tartalmához a megjelenítés van kötve.

A *PizzaServerAPI* importálható interfészének függvényei mind aszinkron típusúak. Tehát, amelyik komponensben éppen hívjuk valamelyiket, ott nem lesz blokkoló a hatása és folyhat más komponensek kirajzolása.

Adatkontextus

Néhány adat közösen használt több komponens által is, pl. a szerepek listázva vannak a táblázatukban, viszont, ha az *admin* új felhasználót hoz létre, ő is választ a szerepek közül, hogy melyet rendelje a fiókhoz. A kosár tartalma a Menü komponense által bővített, a rendelés leadásánál pedig olvasott. Hogy mindenki ugyanazt érhesse el, az alsóbb szinteken lévő komponenseket mindet magába foglalja egy *AppContext*^{xxix} nevű komponens, ami feladataként elvégzi a tárolást, és tárolóinak elérését biztosítja, illetve azok módosításához függvényeket (*provider*). Az alsóbb szintek (*consumer*-ek), ezeket injektálják, így, ha az egyik komponensből változtatunk az egyik adaton, az, bármely másik komponensnek, ahová injektálva van, újra rajzolását okozza.

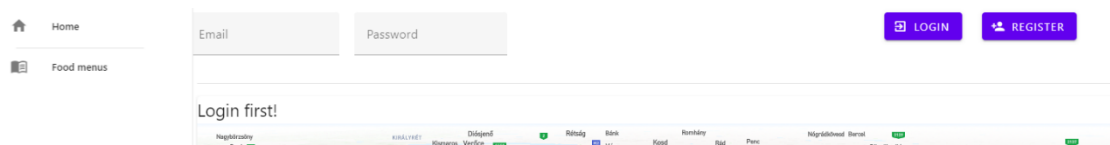
Manuális tesztelés

A kliens működésének tesztelését manuális módszerekkel végeztem. A tesztadatok az egyik *Controller* osztályban, név szerint *FillController* fájlban, a kódban vannak tárolva. Ezeket az adatokat a *ChatGPT* mesterséges intelligencia modell generálta. A kód akkor fog futni, ha kérés érkezik a “/fill-ingredient/...” végpontokra. Természetesen ezek védett végpontok és az *admin*-nak elérhetőek egyedül, viszont arra a szerepre sincs létrehozva még fiók, így az alábbi *work-around* megoldást alkalmaztam: az autentikáció filterének kiváltható egy mellékága, amely automatikusan *admin*-ként autentikálja a kérést, ha az *Authorization header*-ben elhelyezzük a *root* kulcsszót. A *fill-server* parancsfájl az utóbbi megoldást alkalmazva küld kérést, előre definiált sorrendben, a végpontokra.

Bejelentkezés tesztelte

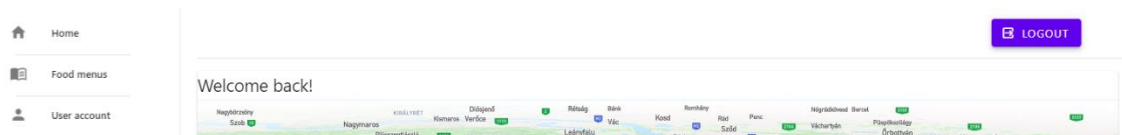
Bejelentkezési utáni *role cookie* sikeres tárolását ellenőrzi az első tesztet. A menü komponens pontjainak automatikus bővülését várjuk, miután bárki bejelentkezik. Ez azért történhet meg, mert a *role* minden komponens számára a *provider*-ből van injektálva, a bejelentkezés a *provider*-ben állítja be a *cookie*-ból nyert adatot, amiről minden alkomponens értesítve lesz.

1. Állapot - bejelentkezés előtti:



21. ábra - Első állapot

2. Állapot - sikeresen bejelentkezve:



22. ábra - Második állapot

Az oldal újratöltése nem változtatja meg az állapotot. A kijelentkezés gomb viszont azonnal visszarajzolja az 1. állapotot. Az újratöltéssel az 1. állapot akkor érhető el, ha a bejelentkezés óta eltelt 5 óra és ürül a lejárt *cookie* a tárhelyből.

Alternatív állapot a másodikra, bizonyítja, hogy tényleg fiók alapon történik a bejelentkeztetés. Érvénytelen adatok megadásánál sikertelen a bejelentkezés, amiről szintén kapunk értesítést:

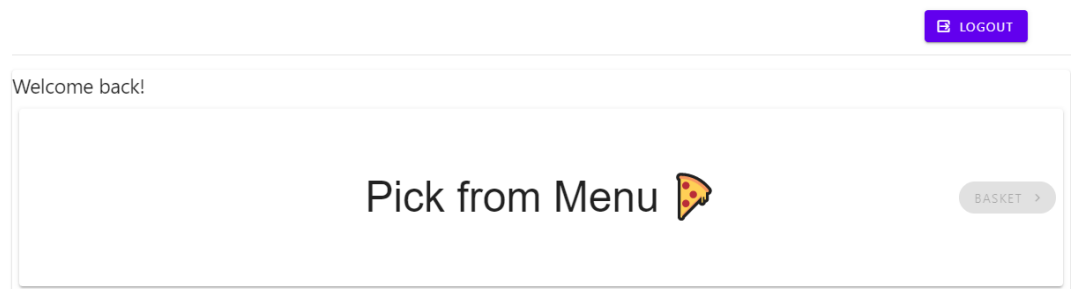


23. ábra - 3. állapot

Rendelés tesztése

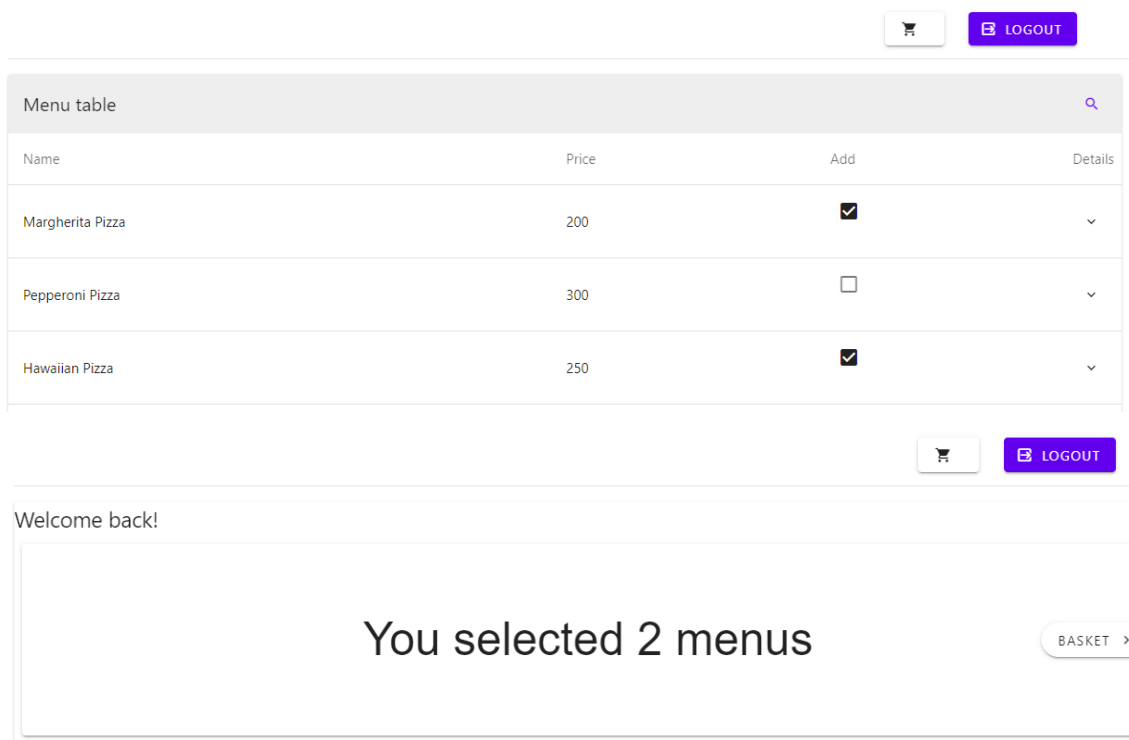
A rendeléskor a *provider* kosarának állapotát módosítja a menü táblából történő ételválasztás. A kosár viszont üresként van még inicializálva az oldal betöltése után.

1. Bejelentkezett vásárló üres kosára utáni állapot:



24. ábra - Első állapot

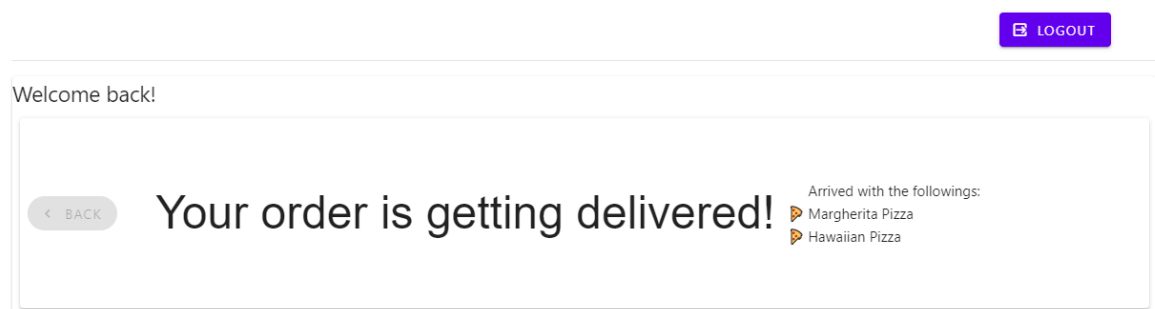
2. A menü néhány eleme kiválasztásra került:



25. ábra - Második állapot

Látható, hogy a felső sor kosár gombjának kirajzolása nem pusztán a menü táblázatának az állapotához kötött, hiszen a főoldalon is jelen marad, miután visszairányított minket oda. A kosárba helyezett menüket a főoldal lapozható komponense is látja, aktívvá vált annak lapozófüle.

3. A rendelés sikeres leadása utáni állapot:





26. ábra - Harmadik állapot

Látható, hogy a kosár automatikus ürítésére sor kerül, miután leadtuk a rendelést, hiszen a méretéhez kötött gomb állapota is megváltozott: újra el lett rejtve.

Felugróablak tesztése

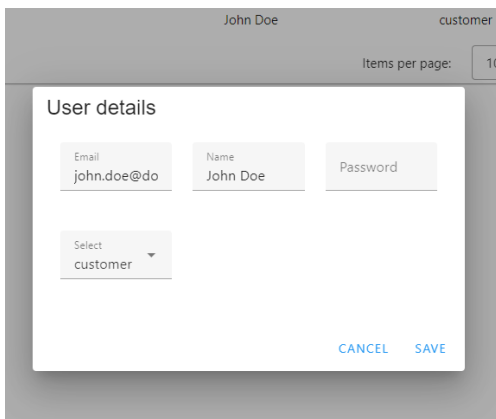
A tesztet alanya a *User* táblában, meglévő felhasználó szerkesztésére használt felugróablak. Tesztelésével megvizsgáljuk a háttérben történő folyamatokat.

1. Alapállapot, a folyamat kezdeményezése előtt:

User table			
Email	Name	Role	Actions
john.doe@domain.com	John Doe	customer	 
Items per page: 10 11-11 of 11 < > >			

27. ábra - Első állapot

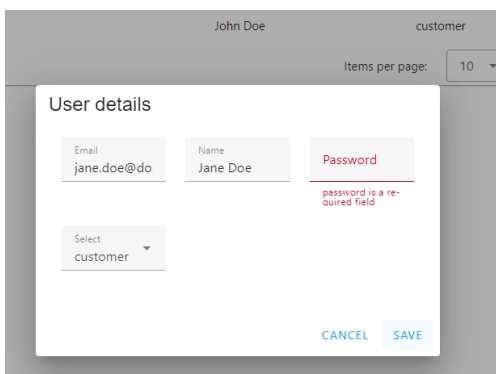
2. A folyamat elindítása a szerkeszteni kívánt rekord ceruza ikonjával:



28. ábra - Második állapot

Látható, hogy a megnyíló felugróablak komponense a legfelsőként kerül kirajzolásra, mezői pedig kitöltésére kerültek a szerkesztett sor azonosítója által jelölt adatokkal. Nem közvetlen szerkesztjük az adatokat, hanem azok egy *deep copy* (rekurzívan, minden objektum értékeinek másolata) változatát.



3. Mentési kísérlet, hibásan módosított mezőkkel



29. ábra- Harmadik állapot

A mentés üres jelszómező állapottal lett meghívva, emiatt nem zárult be a felugróablak és a háttérben is látszik, hogy nem módosult a rekord a többi *valid* mező értékére, tehát nem változott annak a komponensnek az állapota.

4. Sikeres mentési kísérlet

User table			
Email	Name	Role	Actions
jane.doe@domain.com	Jane Doe	customer	 
Items per page: 10 11-11 of 11 < > >			

Updated

30. ábra - Negyedik állapot

A felugróablak bezárult, mivel sikeresen futott le a validáció. Ez szintén igaz a szerveroldalon is, tehát nem egy, már létező, email azonosítóra próbáltuk módosítani az előző fiókét. Ennek a pozitív szerverválasznak hatására került megváltoztatásra a tábla komponens állapota és lett kirajzolva a pozitív visszajelzés a *user* felé.

Összefoglalás

Szakedolgozatom dokumentációja végig vezet egy full-stack webalkalmazás megvalósításán. A futárszolgáltatásként működő pizzéria ötlete még a bevezetésben került felvetésre, annak néhány problémáját, mint például a flotta menedzselésével, autók utántöltését, már ez a fejezet is említi.

Ezt követően a felhasználói dokumentációmban végig vezetem az olvasót, a problémák megoldásaira általam kitalált módszereken. A front-end abban a hitben készült hogy letisztult és felhasználóbarát módot biztosít az interakcióra a háttérben megbúvó adathalmazsal, és felhasználói könnyen eligazodnak használatában. A dokumentáció első része ezt segíti.

A második rész egy mély rálátást nyújt a programot működtető szoftveres megoldásokra. Mivel már ismertette a dokumentáció az elvárt végeredményeket, sok esetben visszautal a felhasználói részre a fejlesztői rész, az implementáció miértjeinek magyarázatához. A bemutatott architektúra abszolút back-endjétől, az adatbázistól, eljutunk a front-end kigenerálásáig. Ezt az utat áthidaló szoftveres megoldások közül igyekeztem azokat elmagyarázni legrészletesebben, amelyeket számomra is legnagyobb kihívás volt megérteni.

Végeredményképpen az alkalmazás külső inputokból nyert és generált adatokat képes adatbázisban konzisztensen tárolni, képes onnan lekért adatokkal komplex algoritmusok szerint műveleteket végezni, majd azok eredményeit a felhasználó számára megjeleníteni.

Hivatkozásjegyzék

-
- ⁱ [https://en.wikipedia.org/wiki/Path_\(graph_theory\)](https://en.wikipedia.org/wiki/Path_(graph_theory))
- ⁱⁱ Élsúlyozott gráfok és algoritmusai 3.1
<http://aszt.inf.elte.hu/~asvanyi/ad/ad2jegyzet/ad2jegyzetGrafok2.pdf>
- ⁱⁱⁱ <https://www.businessinsider.com/zume-pizza-robot-expansion-2017-6>
- ^{iv} <https://i.insider.com/59dd58a56d80ad1d008b59bf>
- ^v https://www.freepik.com/free-vector/pizza_2900463.htm a térképen marker, a böngészőben favicon
- ^{vi} <https://vuetifyjs.com/en/components/data-tables/basics/>
- ^{vii} <https://www.mapbox.com/mapbox-studio>
- ^{viii} <https://agiluu.hu/continuous-delivery-1/>
- ^{ix} <https://stackoverflow.com/questions/5881169/what-column-type-length-should-i-use-for-storing-a-bcrypt-hashed-password-in-a-d>
- ^x <https://www.geeksforgeeks.org/why-prim-and-kruskals-mst-algorithm-fails-for-directed-graph/>
- ^{xi} <https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP>
- ^{xii} <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>
- ^{xiii} <https://jpa-buddy.com/blog/the-ultimate-guide-on-composite-ids-in-jpa-entities/>
- ^{xiv} <https://datacadamia.com/data/type/relation/modeling/relationship>
- ^{xv} <https://datacadamia.com/data/type/relation/modeling/relationship>
- ^{xvi} <https://reflectoring.io/bean-validation-with-spring-boot/>
- ^{xvii} <https://plugins.jetbrains.com/plugin/7324-code-iris>
- ^{xviii} <https://www.geeksforgeeks.org/what-is-a-stateless-server/>
- ^{xix} <https://www.baeldung.com/spring-bean>
- ^{xx} <https://www.javainuse.com/spring/boot-jwt>
- ^{xxi} <https://www.javainuse.com/spring/boot-jwt>
- ^{xxii} <https://www.ibm.com/topics/rest-apis> második pont
- ^{xxiii} https://www.tutorialspoint.com/jackson_annotations/jackson_annotations_jsonmanagedreference.htm
- ^{xxiv} https://www.tutorialspoint.com/jackson_annotations/jackson_annotations_jsonignore.htm
- ^{xxv} <https://jsoncrack.com/>
- ^{xxvi} <https://medium.com/swlh/websockets-with-spring-part-3-stomp-over-websocket-3dab4a21f397>
- ^{xxvii} <https://plugins.jetbrains.com/plugin/8286-sequence-diagram>
- ^{xxviii} <https://vuejs.org/guide/introduction.html>
- ^{xxix} <https://vuejs.org/guide/components/provide-inject.html>