Miskolci Egyetem

Gépészmérnöki és Informatikai Kar

Általános Informatikai Intézeti Tanszék



**Ételajánló webes alkalmazás**

**Szakdolgozat**

**Készítette:**

**Név:** Szilágyi Gábor

**Neptunkód:** S9NJK6

**Szak:** Mérnök Informatikus BSc

Korszerű web technológiák szakirány

**1. Bevezetés**

A mindennapjaink egyik fontos része az evés és az ezt megelőző döntéshozatal. A mai választási lehetőségekben, egyre több emberrel esik meg, hogy nem tudja eldönteni mit egyen. Ezen a helyzeten az sem segít, hogy az embereknek egyre kevesebb idejük van arra, hogy saját maguknak főzzenek.

Szakdolgozatom célja olyan webalkalmazás elkészítése, ami ezt a mindennapi döntéshozatalt és a velejáró extra munkát könnyíti meg. Az alkalmazás a felhasználó által megadott város és konyha alapján étel és étteremajánlást biztosít kérdések segítségével. A felhasználó ugyanakkor segíthet az opciók bővítésében azzal, hogy ezeket elküldi az erre megadott felületen az adminisztráció felé. Az adminisztrációnak ugyanakkor biztosított egy olyan felület ahol kezelheti az adatbázis adatait.

A szoftver megvalósítására a *Spring* és az *Angular* keretrendszereket választottam. A Spring keretrendszer *Java* programozási nyelvű alapokra épül, az *Angular* pedig a *JavaScript* nyelvre. Ezen keretrendszerek a szoftver fejlesztését nagyban megkönnyítik a különböző, már előre implementált szolgáltatásaikkal. Választásomat nagyban befolyásolta az a tény, hogy ezek a keretrendszerek, technológiák az iparban is keresettek. Adatbázisnak a *MongoDB*-t választottam, mivel az alkalmazáshoz nincs szükség egy relációs adatbázisra és a funkciók implementálását nagyban megsegíti egy már létező és jól személyre szabható keretrendszer használata.

Az alkalmazás használatához a felhasználóknak és az adminisztrátoroknak nem szükséges komolyabb informatikai tudással rendelkezniük. A felhasználóknak nem szükséges sok inputot megadniuk, mert a választási lehetőségek előre adott igen/nem válaszok, valamint a települések és konyhák neveinek listája is adott. Az adminisztrációs felület annyiban bonyolultabb, hogy az adminisztrátorok az adatbázist kezelik, ezért az inputok nagy része szöveges. A szöveges formátumok miatt lehetnek olyan kérések, amelyek hibás formátumúak.

**2. Felhasznált eszközök, technológiák**

**2.1. Felhasznált eszközök**

**2.1.1. *InteliJ Idea***[1]

Az *InteliJ* *Idea* a *JetBrains* által fejlesztett *Java* alapú integrált fejlesztői környezet. Arra fejlesztették ki, hogy nagyban megkönnyítse a fejlesztők feledatát, akik *Javában*, *Kotlinban, Groovyban* és más *JVM* alapú nyelvben fejlesztenek. Nagyon népszerű ez a fejlesztői környezet, hiszen sok segítséget nyújt a programozónak és a számtalan könnyen elérhető biztonságos plugginjaival nagy testreszabhatóságot biztosít.

A választásom azért erre a programra esett, mivel az iparban is nagyon sok helyen használják, könnyű magának a programnak a használata és én magam is már évek óta használom, így jobban ismerem, mint a többi versenytársát.

**2.1.2. *WebStorm***[1]

A *WebStorm* szintén egy *JetBrains* által fejlesztett IDE. A *WebStorm* *JavaScript* és a hozzá kapcsolódó nyelvekkel való programozás megkönnyítésére lett kifejlesztve. Hasonló az *InteliJ*-hez mind a felhasználó felületében és kezelhetőségében valamint abban, hogy ugyan úgy biztosítja számunkra a nagy testreszabhatóságot.

Azért esett erre a programra a választásom, mert nagyban hasonlít a már említett *InteliJ*-hez és szintén már évek óta használom ezzel megkönnyítve a programom fejlesztését.

**2.2. Felhasznált technológiák**

**2.2.1. *MongoDB***[2]

A *MongoDB* egy nyílt forráskódú *NoSQL* adatbázis kezelő rendszer, ami egy jó alternatíva a relációs adatbázisokra. Tipikusan nagymennyiségű adatok tárolására és ezek kezelésére szokták használni és dokumentum orientált adattárolásra és kezelésre is képes. Szervezetek még többek közt ad-hoc lekérdezésekre, terhelésmegosztásra és szerver oldali JavaScript futtatásokra is szokták használni.

*NoSQL* mivoltából a *MongoDB* nem táblázatszerű architektúrát használ, mint egy relációs adatbázis, hanem gyűjteményekből és dokumentumokból architektúrát. A dokumentumok kulcs-érték párokat tartalmaznak, míg a gyűjtemények (*SQL* tábláknak felelnek meg) dokumentumok kollekcióit.

A dokumentumok hasonlítanak a *JSON*-hoz de a *MongoDB* ennek egy variációját használja, ami a *BinaryJSON* (*BSON*). Ennek az előnye az, hogy sokkal több féle adattípust támogat ezzel megkönnyítve a fejlesztők munkáját. A tárolt adatok lehetnek akár másik dokumentumok, tömbök, de akár ezeknek a kombinációja is a felhasználói kézikönyv szerint.

Tehát a *MongoDb* összegezve:

Miért használják?

* Nagymennyiségű strukturált és strukturáltalan adat tárolása függőleges és vízszintes skálázhatósággal. Keresés mezővel, indexel és kifejezéses lekérdezésekkel.
* Integrált adatok alkalmazásokhoz, beleértve hibrid és többfelhős alkalmazásokat.
* Az adatbázis lehetővé teszi a beágyazott struktúrákat.
* Több szerveren lehet futtatni egyszerre.

Előnyei:

* Séma nélküli, ami lehetővé teszi akármilyen típusú adat tárolását lehetővé téve ezzel az egyszerű skálázhatóságot és nagyobb flexibilitást relációs adatbázisokhoz képest.
* Dokumentum orientált, ami egyszerűbbé teszi a más nyelvekben objektumokká való átfordítást és csökkenti az adatbázis join-ok használatát ezzel csökkentve a költségeket.
* Egyszerű skálázhatóság támogatva az adatbázis szilánkosítását és az adatok zónákba való csoportosítását kulcsokkal.
* Harmadik féltől származó tároló motorok támogatása.
* *DBMS*-be közvetlen beépített aggregációs lehetőség.

Hátrányai:

* Az automatikus feladatátvételi stratégiájának köszönhetően, ha a mester node nem működik, akkor egy másik node átveszi a feladatát ezzel folytonosságot biztosítva, de ez a folyamat akár egy percet is eltarthat.
* Egy mester node használata miatt az írási mennyiség is limitálva van ezzel potenciális torlódást okozva.
* Nem biztosít, teljes referenciális integritást mivel nem használ idegen kulcsokat.
* Felhasználó hitelesítés nincs alapértelmezetten engedélyezve ezzel rést hagyva potenciális hackereknek bár hozzá lett adva, hogy egy konfigurálatlan adatbázis letilt minden hálózati kapcsolatot.

Továbbá a *MongoDB* sok programozási nyelvet is támogat, többek között a *C, C++, C#, Java, Ruby* és a *Swift*.

**2.2.2. *Spring Boot***[3]

A *Spring Boot* egy *Java* alapú nyílt forráskódú keretrendszer micro szervizek készítésére. Fejlesztését Pivotal *Team* végzi, és arra használják, hogy gyorsan gyártásra kész spring alkalmazásokat hozzanak létre.

A *Spring Boot* egy jó platformot biztosít Java fejlesztők részére, mivel mint már fentebb említve gyorsan önálló gyártási minőségű *Spring* alkalmazást lehet készíteni, amit csak futtatni szükséges. Mindez lehetséges, azért mert elég csak egy minimális konfiguráció kezdésként így nincs szükség elkészíteni egy teljes Spring konfigurációt.

A következő előnyöket biztosít a fejlesztők részére:

* Könnyen érthető és fejleszthető *Spring* alkalmazások.
* Termelékenység növelése.
* Fejlesztési idő csökkentése.

Céljai a *Spring Boot*-nak:

* Bonyolult *XML* beállítások elkerülése.
* Egyszerűbb gyártás kész alkalmazás fejlesztése.
* Alkalmazás önálló futtatása és fejlesztési idő csökkentése.
* Egyszerűbb kezdés biztosítása az alkalmazással.

További funkciók:

* Flexibilis *Java Bean, XML* és adatbázis tranzakció beállítások biztosítása.
* Erős kötegelt feldolgozás és *REST* végpontok kezelése.
* Minden automatikusan van konfigurálva.
* Annotáció alapú alkalmazás biztosítása.
* Egyszerű függőség kezelés.
* Tartalmaz *Beágyazott Servlet Konténert*

A Spring Boot automatikusan beállítja az alkalmazásodat a hozzáadott függőségek alapján az **@EnableAutoConfiguration** annotáció használatával. Ugyanakkor egy adatbázist hoz létre a memóriában, ha nincs kapcsolat beállítva az adatbázisomhoz, mindeközben automatikusan megkeresi és beállítja a projektben lévő összes komponenst.

Tehát a *Spring Boot* egy egyszerűen és gyorsan használható Java keretrendszer, ami nagyban megkönnyíti a programozók életét a széleskörű és mély lefedettségével.

**2.2.3. *Angular***[4]

Az *Angular* egy ingyenes nyílt forráskódú *TypeScript* alapú webalkalmazás keretrendszer, amit a Google *Angular Csapata* vezet, magánszemélyek és más vállalatokkal együtt. Az *Angular* egy *Egy Oldal Aplikáció* Keretrendszer (*SPA*) amit gyors webalkalmazások készítésére használnak. Az *SPA* koncepciója az, hogy csak egyszer kelljen, az oldalt betölteni utána csak adatokat kelljen kérnünk a szervertől, ami gyorsabbá teszi a webalkalmazásunkat.

A keretrendszert *HTML* és *TypeScript* segítségével tudjuk használni és különböző funkciókat megvalósítani. Maga a rendszer is *TypeScript*-ben íródott, ami implementál különböző funkcionalitásokat, amit mi is importálhatunk a saját alkalmazásunkba.

**2.2.4. *TypeScript***[5]

A *TypeScript* egy magas szintű programozási nyelv, amit a *Microsoft* fejleszt és tart karban. A programozási nyelv a *JavaScript* egy alfaja mivel hozzáad egy opcionális szigorú típusitást. Arra lett tervezve, hogy nagyméretű alkalmazások fejlesztésére használják és mivel a *JavaScript*-nek csak egy alfaja ezért át lehet fordítani JavaScript kóddá és ugyan ezt meg lehet csinálni fordítva is.

**2.2.4. *Bootstrap***[6]

A *Bootstrap* egy ingyenes nyílt forráskódú *CSS* keretrendszer, amit a reszponzív front-end webfejlesztésre fejlesztettek ki. *HTML*, *CSS* és *JavaScript* könyvtárakat tartalmaz, amik célja az informatív weboldalak készítésének a megkönnyítése.

Az elsődleges célja a *Bootstrap*-nek az hogy egy már előre definiált kinézeteket tudjunk adni a weboldalunkon megtalálható elemeknek és ezek a kinézetek reagálni tudjanak a megjelenítő eszközök tulajdonságaira, főként a méretre. Fő célja ezzel az, hogy minden megjelenítő eszközön ugyan úgy nézzen ki a weboldalunk és ennek a megvalósítását könnyebbé tegye számunkra. Ugyanakkor lehetővé teszi számunkra, hogy a már definiált elemeket szeméjre szabjuk a saját szükségleteinkre.

*JavaScript* komponenseket is tartalmaz, amikhez nincs szükség külső könyvtárakra. Ezek főként a dinamikus, interakcióba léphető elemeknél lehet megtalálni, mint például egy legördülő mező vagy folyamatjelző sáv. Néhány komponense a keretrendszernek ugyanekkor ki bővítheti a már létező interfész elemet, mint például a bemeneti mezők automatikus kiegészítése, de a legfőbb feladata és komponensei a weboldal elrendezéséért felelősek.

**3. Feladat leírása**

**3.1. Téma**

Mint már azt az előbbiekben is említettem már az alkalmazásom fő témája az, hogy megkönnyítsem az emberek döntéshozatalát abban, hogy egy-egy étkezés során mit egyenek és azt a megadott városukban hol kapható.

Ezt a témát főként azért választottam mivel jómagam is, mint sokan mások, gyakran szenvedek attól, hogy nem tudom, mit szeretnék pontosan enni csak azt, hogy milyen ízvilágú ételt kívánok abban a pillanatban. Ugyanakkor gyakran azt se tudom, hogy az adott városban ahol éppen tartózkodom, milyen éttermek vannak, és azok mit szolgálnak fel. Ezért döntöttem úgy hogy megpróbálok egy olyan alkalmazást készíteni, ami valamilyen formában a fenti problémáimat meg tudja oldani.

**3.2. Hasonló alkalmazások**

Bár már léteznek hasonló alkalmazások, a legtöbbjük inkább fókuszál az éttermekre és az ételrendelésre mintsem, arra, hogy segítsen az ételválasztásban. Amelyik alkalmazások pedig segítentek ételt választani azok csak egy, a felhasználó által megadott listából választanak ki egyet, ami nem oldja meg azt a problémát, ha nem tudjuk mit együnk, vagy esetleg valami új ételt is megkóstolnánk, amiről nem tudunk.

Jelenleg (dolgozat írásának idején) legjobb tudomásom szerint nem is létezik olyan alkalmazás, ami kifejezetten ételt ajánl fel a felhasználónak, hanem éttermeket sorol fel a megadott szűrési feltételekre. Hasonló ételrendelő alkalmazások, mint például a *Wolt* és a *Foodpanda* csak az éttermekre szűrnek és ezek a szűrési feltételek is eléggé tágak, ezzel megnehezítve a dolgunkat. Ugyanakkor ezek az alkalmazások néha eléggé pontatlanul adják meg, néha teljesen elhagyják azokat, hogy egy-egy étterem milyen témájú és mit szolgál fel, ami megnehezíti és pontatlanná teszi az éttermek szűrését.

Ezért az én alkalmazásom az előbbiekben említett hibákon szeretne javítani és a ki nem használt lehetőségeket kihasználni, ezzel egy a piacon lévő lyukat betömni. Ugyanakkor az alkalmazásom nem veszi el a már piacon lévők létjogosultságát, hanem azokat kiegészíti.

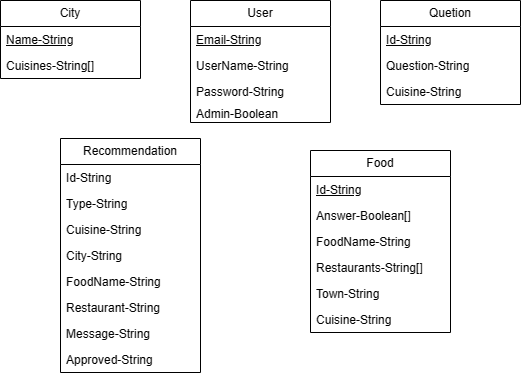
**4. Fejlesztői leírás**

**4.1. Backend**

**4.1.1. Adatbázis**

Az adatbázis tervezésénél a fő szempontom az volt, hogy a lehető legkevesebb táblából álljon, miközben ezek a táblák felépítése a lehető legátláthatóbb és legtöbb adatot tárolják, amiket együtt kezelne a rendszer. Ez a kritériumok miatt és az alkalmazás mivoltából adódóan sikerült egy olyan adatbázist létrehoznom, amiben nincsenek relációk, bár van olyan helyzet az alkalmazás futása során, amikor egy-egy adat függhet a másiktól. Ezek viszont annyira ritkák és specifikusak, hogy egyszerűbb ezeket eset specifikusan a probléma felmerülésénél megoldani, mint az egész adatbázist e-köré tervezni és fölöslegesen bonyolítani ezzel azt.

Ezért a már korábban említett *MongoDB* szolgáltatásait vettem igénybe az adatbázisomhoz, hiszen ez a rendszer biztosít egy ingyenes és könnyen használható *NoSQL* adatbázist, ami az én céljaimnak tökéletesen megfelel. Ugyanakkor támogat több külső keretrendszert is, amik nagyban megkönnyítik az adatbázissal való kommunikációt a fejlesztő részére.



*1. ábra – Adatbázis táblák és mezőik*

Az 1. ábrán látható az adatbázisban megtalálható összes tábla, azok mezői és típusai. Több táblában is látszik, hogy tartalmaznak tömböket és egyes táblákban van olyan mező, amiknek van relációja az adatbázis ennek ellenére sem relációs hanem *NoSQL*. Amennyiven az adatbázis relációs adatbázis lenne, akkor a sok tömb miatt az egyes táblákban sok kapcsolótábla jönne létre, ami nagyban lassította volna az adatbázis működését abból kifolyólag, hogy egy-egy konyhának akár több száz különböző étele is lehet és több városban is lehetnek jelen éttermek, amik ezeket felszolgálják, elkerülhetetlen a nagymennyiségű adat.

Ezért is döntöttem úgy, mivel már az előbbiekben is említettem, hogy ebben a fajta adatbázisban, a nagymennyiségű adatban a keresés gyorsabb és megéri ezt használni és kis kellemetlenségek, amit okozhat ebből kifolyólag.

**4.1.2. Repository**

Az adatbázisomhoz való könnyű hozzáférésem és kezelésem érdekében a repository szinten a *Spring* által szolgáltatott függőséget használtam. Ennek köszönhetően nem volt szükséges leprogramoznom külön minden lekérdezést és műveletet, amit az adatbázisomtól kértem volna, hiszen a függőségnek köszönhetően az alap CRUD műveletek rögtön használhatóak. Továbbá a bonyolultabb műveletek elvégzését is nagyban megkönnyíti, mivel azokat se kell nekem megvalósítanom csak az interfacemben a metódus névnek egy *SQL* parancshoz hasonló nevet kell adni és a megfelelő annotáció használatával megadni az esetleges változók helyét és nevét. Példa erre a következő kódrészlet, ami az ajánlott ételt kérdezi le:

@Query("{'answer': ?0,'town': ?1,'cuisine': ?2}")

Optional<FoodEntity> findFoodEntityByAnswerAndTownAndCuisine(boolean[] answer, String town, String cuisine);

Ahhoz hogy ezeket a funkciókat tudjuk, a megfelelő kiterjesztést kell használnunk, amit a függőségünk biztosít. Erre a kiterjesztésre egy példa az előző kód interfaceja: public interface FoodRepository extends MongoRepository<FoodEntity, String>. Itt látható, hogy nekünk nincs szükségünk csak egy osztály megadására, ami úgy néz ki, mint az adatbázis táblánk és a használt Id típusának a megadására.

Ezen a repository szinten az alkalmazásomban nem található más csak a már előbb (1. ábrán) látható táblák mezőinek Entitásokká (osztályokká) való átfordítása és az egyes táblázatokhoz tartozó műveleteknek az interfacejai az előbb már említett és bemutatott módon. Egy Entitás kódrészlete:

@Document("Foods")

public class FoodEntity {

@Id

private String id;

private boolean[] answer;

private String foodName;

private String[] restaurants;

private String town;

private String cuisine;

Ez a fajta működés nagyban hasonlít más *JDBC*-s támogatásokhoz, hiszen a működésüknek az alapja ugyan azon alapszik az *Inversion of Control (IoC) –on.* Továbbá ezek a függőségek lehetővé teszik az egyszerű kapcsolat létrehozást az adatbázissal hisz csak a program application.*properties* fájljába kell megadnunk a szükséges adatokat és minden mást magától megold.

**4.1.3. Service**

Ebben a rétegben van implementálva a szerver logikája, bár ebben az esetben ezek inkább a szerverben felmerülő kivételek kezelését jelenti. A réteg béli osztályok és interfacek téma beli felbontása hasonló a repository rétegéhez, bár itt megjelenik a login service is, ami a felhasználók beléptetéséért és regisztrálásáért felel. Erre azért volt szükség mivel így egy kicsit jobban átlátható a felsőbb réteg számára és később majd a kliens számára ez a szolgáltatás teljesen elkülönül a többi felhasználókhoz köthető szolgáltatásoktól.

Mivel egyelőre a szerver szolgáltatásai megoldhatók voltak egy-egy adatbázis lekérdezéssel, így a service réteg kivételeket kezel, amik esetlegesen előfordulhatnak és a felhasználónak segítségére lehet. Ezek saját kivételek az egyszerűbb olvashatóság érdekében, amik az alábbiak: EntityAlreadyExistsException, NoSuchEntityException, NoSuchFoodException.

A kivételek nevei le is írják azt, hogy mikor is dobódnak és mivel ezek csak a program futása alatt derülnek ki, ezért ezek az osztályok a RuntimeException osztályból származnak. Maguk a kivétel osztályok csak annyiban különböznek a többi már létezőtől, hogy megváltoztatják a dobott üzenetet egy a probléma specifikusra, amire itt éppen szükség volt.

Továbbá a bejövő és kimenő adatok megfelelő formátummá való konvertálás is itt történik az egyes rétegek számára. Ilyenek például a lekérdező metódusok, ahol az alsóbb rétegtől kapott adatokat át kell alakítanunk a felsőbb réteg számára hasznos és értelmezhető adatokká. Erre, konkrét példa az alábbi kód részlet:

public Iterable<CityDto> allCities() {  
 List<CityDto> output = new ArrayList<>();  
 for (CityEntity cityEntity : cityRepository.findAll()  
 ) {  
 output.add(new CityDto(cityEntity));  
 }  
 return output;  
}.

Ez azért szükséges, mivel létezik egy olyan elv, ami szerint csak a közvetlen alattunk lévő rétegeket és a saját rétegünket láthatjuk, továbbá nem biztos, hogy minden adatot, amit kapunk fel is használunk vagy továbbadunk, erre egy példa a **Repository** részénél leírt entitás és az alábbi hozzá kapcsolható Dto összehasonlítása:

public class FoodDto {  
 private boolean[] answer;  
 private String foodName;  
 private String[] restaurants;.

Ez a konvertálás ugyan-úgy előjön, mikor adatot szeretnénk felvinni vagy azt módosítani. Itt is a már előbb említettek miatt szükséges csak itt, nem azért mert nincs szüksége bizonyos adatokra, hanem azért mert esetlegesen több adatra van szükség az alacsonyabb réteg használatához, mint amit esetlegesen egy adattaggal be tudnánk kérni. Erre egy példa a már említett ételeket kezelő résznél van ahol az entity-nek szüksége van más adatokra is, amit külön kell kérnünk a metódusnál mivel ezeket az adatokat a Dto nem kezeli, konkrét kód példaként:

public FoodDto createFood(FoodDto foodDto, String town, String cuisine) throws EntityAlreadyExistsException {  
 Optional<FoodEntity> searched = foodRepository.findFoodEntityByFoodNameAndTownAndCuisine(foodDto.getFoodName(), town, cuisine);  
 if (searched.isEmpty()) {  
  
 FoodEntity output = foodDto.toEntity();  
 output.setCuisine(cuisine);  
 output.setTown(town);  
 return new FoodDto(foodRepository.save(output));  
 }  
 throw new EntityAlreadyExistsException(foodDto.getFoodName());  
}.

Természetesen ezek a plusz, az azonosítón kívüli, adattagok a lekérdező metódusoknál is megtalálhatók, hisz léteznek feltételes lekérdezések, amikhez szükségünk lehet adatokra, amit a Dto nem tartalmaz. Erre egy példa az a metódus, amivel törlünk egy ételt: public void deleteFood(FoodDto foodDto, String town, String cuisine). Erre itt azért van szükségünk mivel több azonos étel is létezhet különböző városokban vagy esetlegesen konyhákban és pontosan tudnunk kell, melyiket kell törölni.

**4.1.3. Controller**

A Controller rétegben találhatók meg a szerver által biztosított szolgáltatásoknak a végpontjai, amiket a kliensünk használhat, továbbá itt még megtalálhatóak, azok az adatmodellek is, amikkel a szerver a külvilággal való kommunikációja során használ.

Ezeket a végpontokat az előzőkhez hasonlóan csoportosítottam annyi különbséggel, hogy nagyobb szerepet kapott az, hogy az adott végpontot adminisztrátor fogja e használni vagy csak egy egyszerű felhasználó. Ezért fordul elő, hogy néhány szolgáltatás több helyen is előfordul.

Az itt található controller osztályok nagyban függnek a megfelelő annotáció használatától, hiszen ezek segítségével tudja eldönteni a szerverünk, hogy az éppen bejövő http kérésekre melyik végpontot hívja meg és milyen formátumban válaszoljon. Ennek a folyamatnak az egyik lényeges része a megfelelő url-hez tartozó végpont megtalálása ahol fontos, hogy a két url pontosan ugyan az legyen. Ezt meg lehet adni akár osztály szinten is, mint például a @RequestMapping("/admin")-nal de közvetlenül a végpontoknál is meg lehet adni például a @GetMapping("/foods/{town}/{cuisine}")-nal. Ennek akkor van haszna, hogyha a controlleren belüli végpontok url címének az eleje ugyan az és csak az utolsó felében vannak eltérések például az elnevezésekben (többesszám jelölése) vagy az url-en keresztül kért változókban (száma, típusuk, nevük).

A másik fontos eleme a folyamatban a http kérés típusának a pontos megadása. Ez azért szükséges, mert ezzel jelezzük a rendszer felé milyen típusú a kérésünk és mivel maguknak a kéréseknek és azok válaszainak a felépítése, adattagjai sem teljesen ugyan olyanok. Erre egy példa a get és a post közötti különbség, míg a get kéréseknél általában nincs body rész megadva és a választ a böngészőben el lehet, menteni addig a post kérésnél szinte mindig van body rész megadva és az erre kapott választ nem lehet menteni a böngészőben. Ezekből a különbségekből kifolyólag lehetséges, hogy ugyan azzal az elérési úttal és változókkal rendelkezzen két különböző típusú http kérés ugyan azon az egy szerver felé. Erre egy példa a következő két végpont:

@DeleteMapping("/food/{town}/{cuisine}")

@PutMapping("/food/{town}/{cuisine}")

Az itt megtalálható Dto adatmodell osztályok tartalma az előző service réteg osztályaitól annyiban tér el, hogy az itt megtalálható adattagoknak megfelelő validációs kritériumokat kell adnunk annak érdekében, hogy minden szükséges adat, amire szükségünk van, megfelelően érkezzen be hozzánk és az ezekből fakadó hibákat már az elején el tudjuk folytani. Természetes ez csak akkor működhet, ha a megfelelő helyen a végpont metódus adattagjainál a megfelelő annotációt használjuk, ennek a programnak az esetében a javax @Valid annotációját használtam. A Dto-ban lévő kritériumokra példakód:

@NotEmpty  
@Email  
private String email;  
@NotEmpty  
private String userName;  
@NotEmpty  
private String password;  
private boolean admin;

Továbbá itt található még az alkalmazás kivételeit kezelő osztály. Ennek az osztálynak az a feladata, hogy amikor a program futási ideje alatt valamilyen kivétel keletkezik, azt feldolgozza és a keletkezett hibaüzenetet a kliens felé elküldi, amit majd később ott dolgoz fel.

Ennek a kivételkezelésnek több fajtája is létezik. Az egyik az, amikor magában a controllerben valósítjuk meg egy külön metódus és az *@ExceptionHandler* segítségével, hogy az adott controller hogyan reagáljon az adott kivételek dobásakor. Ennek a megoldásnak a legnagyobb hátránya az, hogy ezt a folyamatot minden controller osztályban el kell végeznünk, ha nem tudjuk valamiért megoldani azt, hogy minden controller ugyanazon egy alap osztályból származzon ahol ezt a kivételkezelést már implementáltuk.

Másik egyszerű megoldás az, amikor közvetlen a kivétel osztályt annotáljuk a *@ResponseStatus* –al. Ennek a megoldásnak a két nagy hátulütője az, hogy nem tudjuk, használ csak az egyénileg létrehozott kivételekkel és az, hogy ez a megoldás nem teszi lehetővé, hogy body-t állítsunk be a kliensnek küldött válaszba ezzel nagyban limitálva az átadható információ mennyiségét a felmerülő hibával kapcsolatban.

Az általam használt megoldás nagyban megoldást nyújt az előbbiekben bemutatott kivételkezelésekre. Ezzel a megoldással nincsen szükségünk csak egyszer egyetlen egy osztályban implementálnunk azt, hogy milyen formában szeretnénk, és milyen információkat akarunk átadni a kliens részére, azt az alábbi kódrészletben lehet látni, hogy milyen annotációra és osztály származtatására van szükségünk ahhoz, hogy mindezt megfelelően tudja kezelni a szerver:

@ControllerAdvice  
public class RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler.

Ezen kívül nincsen más teendőnk csak a már elsőként bemutatott módszer alapján kezelő metódusokat implementálni a különböző kivételekhez. Erre egy konkrét példa az alábbi kódrészlet:

@ExceptionHandler(value = NoSuchEntityException.class)  
protected ResponseEntity<Object> handleNoSuchElement(NoSuchEntityException noSuchEntityException, WebRequest webRequest) {  
 return handleExceptionInternal(noSuchEntityException, noSuchEntityException.getMessage(), new HttpHeaders(), HttpStatus.*NOT\_FOUND*, webRequest);}.

Az előbb bemutatott kivételkezelés lényegében lehetővé teszi a kivételek globálisan történő kezelését ezzel elkerülve a fölösleges kódismétlést és az ebből fakadó problémákat továbbá lehetővé teszi a relatív nagymértékű testreszabhatóságot.

**4.2. Frontend**

**4.2.1. Modell**

Ahhoz, hogy egyszerűen és hatékonyan tudjon kommunikálni a frontend kliensünk a backend szerverrel, elsősorban szükségünk van adatmodellek definiálására. Ezeknek az adatmodelleknek a feladata az, hogy biztosítsa számunkra azt, hogy megfelelő és könnyen kezelhető, egységes formátumban tudjuk elküldeni a szervernek a megfelelő adatokat és a kapott válaszokat, az eredményeket könnyen fel tudjuk dolgozni a kliens oldalon az előbb már említettek miatt.

A modelleknek a definiálása általában egy egyszerű interface-el történik, ami nem tartalmaz mást csak a szükséges adattagokat, bár lehetséges az is hogy a modellt osztályként definiáljuk, ami tényleges használatkor nem sokban különbözik, de viszont ilyenkor lehetőségünk van más metódusokat is implementálnunk a modell osztályon belül. Az előbb említett osztállyal való megoldást nem gyakran szokták használni, mivel ami plusz funkciókat kínál azt a service osztályokban szokták megvalósítani a kód jobb átláthatósága miatt. Mivel nekünk csak egy adatstruktúrára van szükségünk és a kivételek nagy részét többletkód és probléma nélkül meg lehet, oldani ezért jobban megéri interfacet használni az esetek nagy részében.

export interface UserModel {

email: string;

userName: string;

password: string;

admin: boolean

}

A fentebb látható kódrészlet egy példa az előbbiekben említett modellek definiálására. Jól látható hogy nem tartalmaz mást csak az adattagoknak a nevét, típusát és természetesen magát a nevét az interfacének amivel el tudjuk érni. Fontos megjegyezni, hogy az adattagoknak úgy adjuk meg a nevüket és típusukat, hogy azt a szerverrel való kommunikáció alatt a két oldalon lévő adatokat átfordító algoritmusok megfelelő és felhasználható formátumokba tudják azokat átalakítani.

**4.2.2. Service**[7]

Ezeknek a moduloknak a feladata az, hogy elkülönítse a komponensektől az összetartozó és újra hasznosítható adatokat, eljárásokat. Főként olyan metódusokat tartalmaznak, amik nem álnak közvetlen kapcsolatban a komponensek megjelenítésért felelős részeivel, erre a fő példák a különböző *http* kérések a szerver felé és a különböző felhasználó inputok ellenőrzése.

Bár nincs kényszerítve a programozó ezeknek a használatára, de erősen ajánlott, mivel így egyszerűen elkerülhető a kódismétlés és az ebből fakadó problémák, nehézségek, továbbá nagyban elősegíti az alkalmazás modularitását.

A programom esetében a már előzőekben bemutatott módokhoz hasonlóan modellenként és témákként különítettem el a service komponenseimet. Ez azt jelenti, hogy minden service, amit az adminisztrátori rész használ, modellekként külön komponensekbe vannak csoportosítva a felhasználóhoz köthetök kivételével mivel innen csak kevés olyan metódus van, amit kizárólag az adminisztráció használ fölöslegessé téve azoknak a külön csoportosítását. Továbbá van egy szervice, amiben olyan metódusok, találhatók, amiket az átlag felhasználóknak szánt megjelenítő komponensek használnak ezért itt több különböző modellt felhasználó eljárás is megtalálható.

Ezeknek a service komponenseknek a felépítése ugyan az, mint egy osztályé, amit fel lehet használni függőségként. Megtalálható az osztály, adattagok és konstruktor deklarálása, mint például az alábbi kódrészlet:

export class AdminCityService {

adminUrl: string;

constructor(private http: HttpClient) {

this.adminUrl = 'http://localhost:8080/admin' }

Továbbá még különböző bonyolultságú metódusok is megtalálhatóak, amik a komponens fő funkcióit teszik ki. A következő kódrészletben látható két bonyolultságú *http* get lekérdezés, természetesen ezeknél jóval eltérő és bonyolultabb eljárásokat is lehet a service osztályokban implementálni:

async getQuestions(cuisine: string) {

return lastValueFrom(this.http.get<string[]>(this.url + 'customer-questions/' + cuisine));

}

async getResult(answers: boolean[], town: string, cuisine: string) {

let httpParams = new HttpParams();

answers.forEach(answer => {

httpParams = httpParams.append('answers', answer);

})

let options = {params: httpParams};

return lastValueFrom(this.http.get<FoodModel>(this.url + 'result/' + town + '/' + cuisine, options));

}

**4.2.3. Login, regisztrálás**

A mikor megnyitjuk a weboldalt akkor először a belépésre szolgáló oldallal találkozunk, ahol megadhatjuk a regisztrált e-mail címünket és jelszavunkat amikkel bejelentkezve átirányít a jogosultságunknak megfelelő oldalra. Ugyanakkor egy gomb is megtalálható mindig ezen az oldalon, ami átirányít a regisztrációs oldalra ahol az adatok megfelelő kitöltése után, mint belépésnél, megjelenik a regisztrációt véglegesítő gomb, ami megnyomása után a belépő oldalra irányítva be is léphetünk.

Természetesen minden beviteli mezőnek az értékei ellenőrizve vannak, hogy megfelelő formátumúak legyenek és ténylegesen ki is legyenek töltve. Ez azt jelenti, hogy az e-mail címet a szabványnak megfelelő formában kell megadni és mivel jelenleg nincs külön megszabva jelszónak tartalmi követelés így csak minden másik mezőt nem szabad üresen hagyni. A továbbiakban is minden hasonló beviteli mező ugyan ilyen formában van ellenőrizve a kliens oldalon, a szerver oldali azonosításról pedig már az előzőekben a **4.1.3. Controller** nevű fejezetben volt szó.

A háttérben pedig ténylegesen a belépésnél egy egyszerű felhasználói adatoknak a lekérdezése történik, amiket helyileg tárolok az egyszerűség miatt, a regisztrálásnál pedig a megadott felhasználói adatok, egy alapértelmezetten hamis admin adattal kiegészítve, felvisszük az adatbázisunkba.

**4.2.4. Átlag felhasználói komponensek**

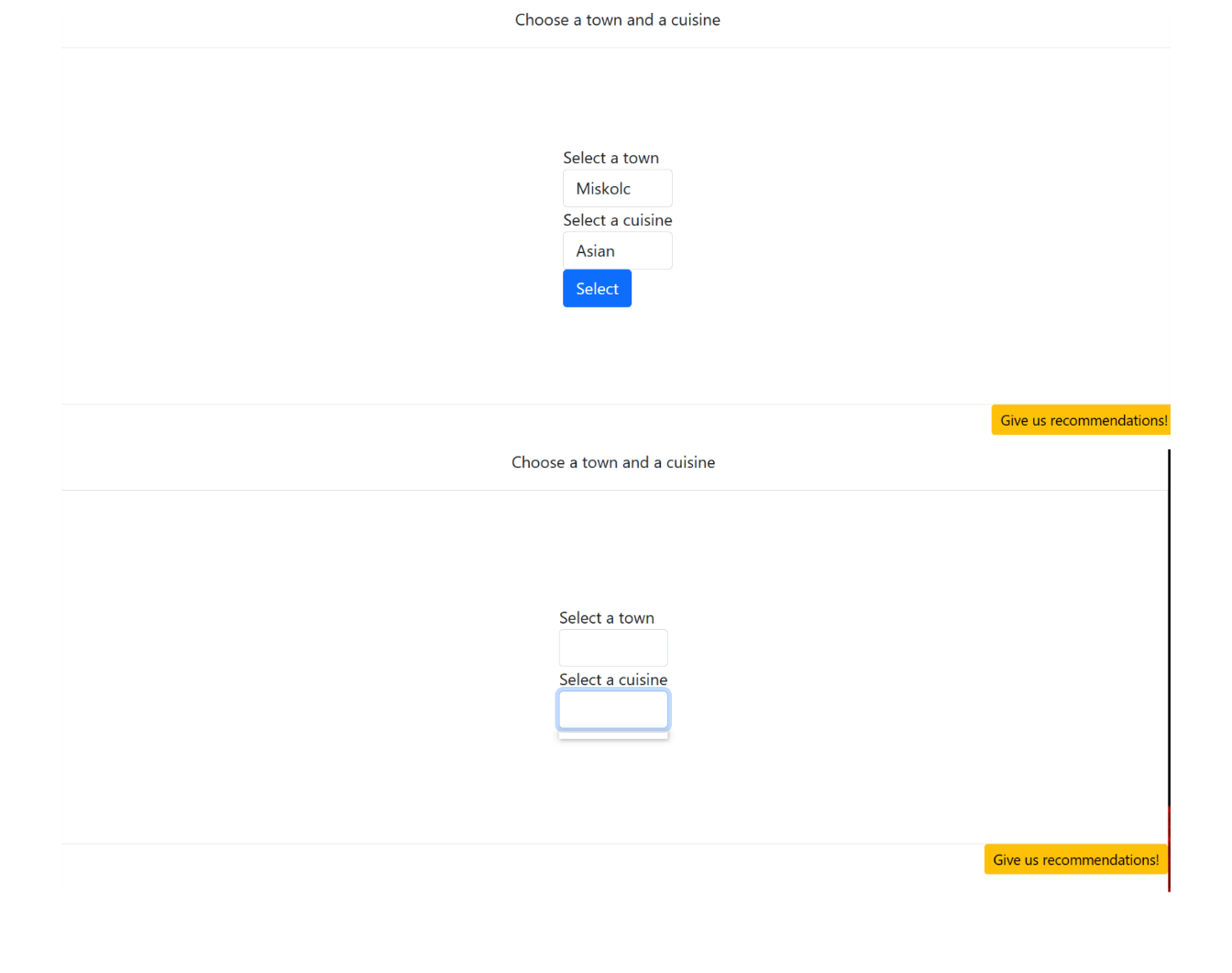
**4.2.4.1. Kezdő oldal**

Ezen az oldalon elsősorban két legördülő ablak található meg ahol a megadott lehetőségek közül kell választanunk a számunkra megfelelőt, és amikor ez megtörtént megjelenik a tovább lepéshez szükséges gomb. A későbbiekben minden hasonló véglegesítő, továbbléptető gomb csak minden szükséges adat megfelelő megadása után jelenik meg.

Mikor választani szeretnénk a legördülő ablakoknál, akkor észre vehetjük, hogy ha nincs, város kiválasztva akkor nem tudunk konyhát se választani. Ez azért van, mivel nem minden városban vannak egyforma ételeket felszolgáló éttermek, ezért egy változást figyelő metódus segítségével csak a városban megtalálható konyhák kerülnek kilistázásra opciónak.

Még ezen az oldalon jobbra található meg az a gomb, ami továbbvisz minket arra az oldalra, amin ajánlásokat tudunk küldeni az adminisztráció felé.

A következő ábrán (2. ábra) lehet látni legördülő ablakokat és az említett működésüket továbbá az ajánlat küldéséhez szükséges oldalhoz vezető gomb.



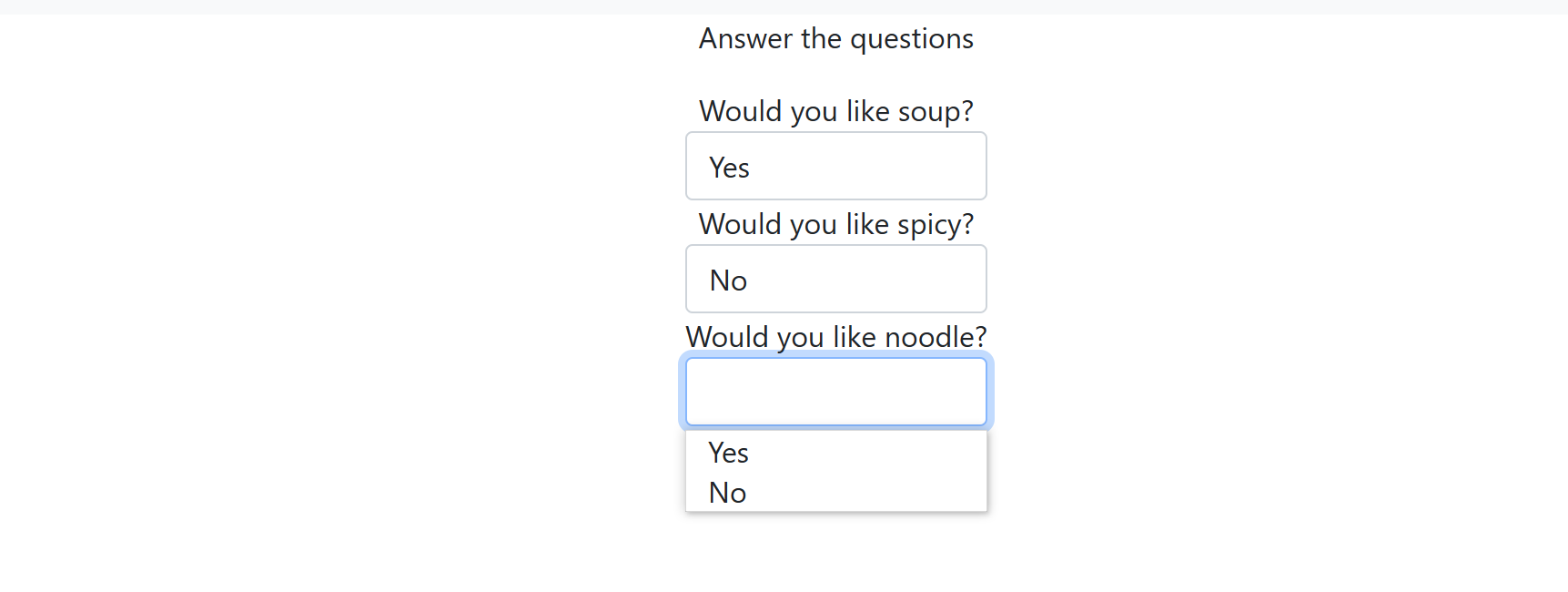
*2. ábra – Legördülő ablakok sajátossága*

**4.2.4.2. Kérdőív**

Miután kiválasztottuk a számunkra megfelelő opciókat azután egy kérdőívvel találjuk magunkat szemben. Ez a kérdőív csak konyhánként különbözik, tehát ha más városnál ugyan-azt a konyhát választjuk, akkor is ugyan azt a kérdőívet kell majd kitöltenünk.

Ez a kérdőív, mint a következő ábrán (3. ábra) láthatjuk kérdések és egy hozzá tartozó igen/nem válaszokat tartalmazó legördülő ablakok sorozatát tartalmazza. Minden kérdésre kötelező válaszolni, hiszen így tudunk pontosabban választ adni arra, hogy éppen milyen étel érhető el a városban, amire illik a leírásunk.

A *form* tényleges működését tekintve elég egyszerű. A kérdések kiíratása *label* ként történik meg egymás után sorban, ahogyan azt a lekérdezés által visszakapjuk és ezekhez a *label*-ekhez tartozik egy-egy legördülő ablak a megadott választási lehetőségekkel. Ezeket a válaszokat egy *form array*-ben tárolja és dolgozza fel a komponens. Ez a *form array* annyi válasz (*answer*) tagot tartalmaz amennyi kérdés tartozik az éppen lekérdezett konyhához. Mivel fontos, hogy minden adat megfelelő sorrendben legyen ezért az adatok kezelésénél nagy szerepe van a tömböknél a megfelelő indexek használatának.

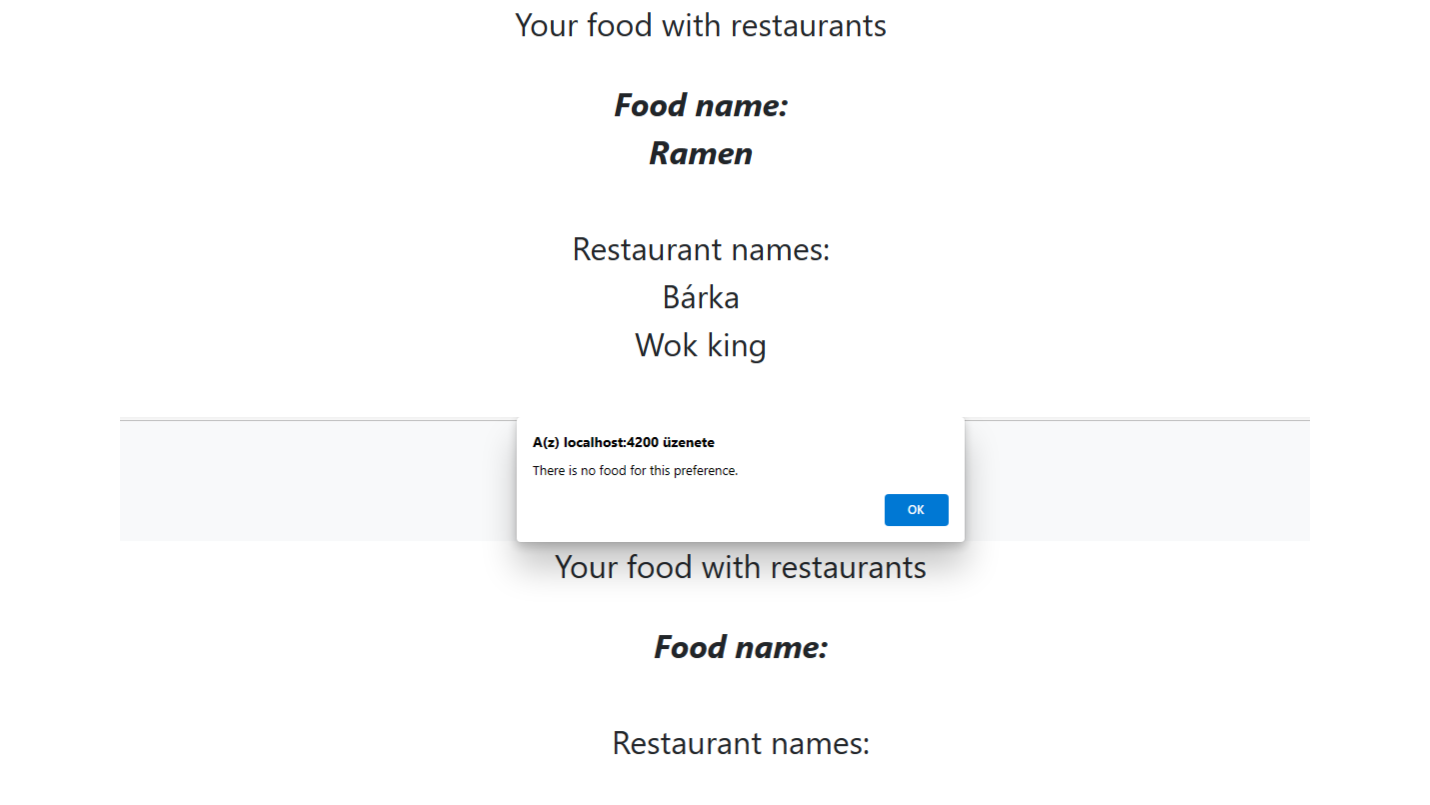


*3. ábra – Kérdőív felépítése, működése*

**4.2.4.3. Étel ajánlása**

A kérdőív kitöltése és elküldése után az eddig megadott adatok alapján kapunk eredményül egy ételt és az éttermek nevét ahol azt lehet valamilyen formában kapni. Abban az esetben, ha nincs egyezés a megadott adatokra, akkor egy felugró ablakban kapunk egy hibaüzenetet és egy az étel adatait üresen hagyott oldalt kapunk.

Ennek az oldalnak az előbb említett állapotait a következő ábrán (4. ábra) lehet látni.



*4. ábra – Étel eredmény oldal státuszai*

**4.2.4.4. Javaslat adminisztrációnak**

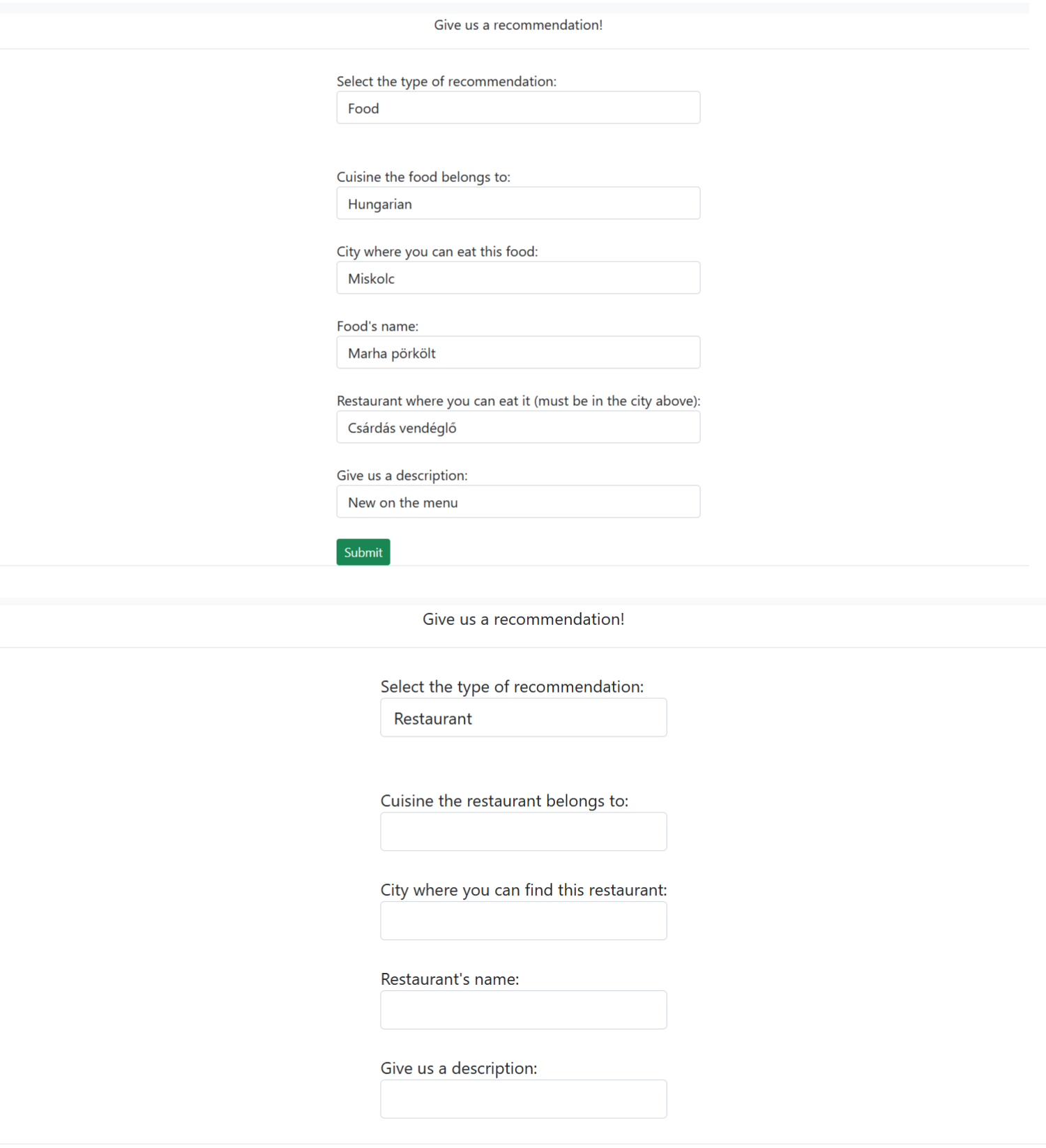
Az ezen az oldalon található űrlap kitöltésével tudunk különböző javaslatokat küldeni az adminisztráció felé. Az űrlap úgy működik, hogy először egy típust kell kiválasztanunk amilyen témában szeretnénk küldeni javaslatot, majd a témának megfelelő további opciók jelennek meg, amiket kitöltve tudjuk elküldeni a már teljes űrlapot.

A téma kiválasztásán kívül a felhasználó szabad kezet kap abban, hogy mit szeretne a különböző mezőkbe írni, mivel így lehetséges, hogy egy felhasználó egymás után több javaslatot is tegyen, még esetlegesen nem létező opciókat használva, például egy újonnan nyílt étterem teljes menüje vagy egy új város és abban megtalálható éttermek.

Ennek a megoldásnak viszont nagy hátránya a felhasználói hiba, mivel egységes modellt használ a komponens a különböző űrlapok elküldésére, így több helyen kell elvégezni a kliens oldalon az inputok ellenőrzését. Továbbá nem minden opció van jelen mindenhol ezért bizonyos elemeket ismételni kell a kiválasztott típusok függvényében, amit a különböző típusokra feltételkezeléssel lehetett megoldani. Ez a megoldás a komponens *html* oldalán sok kódismétléssel és egyéni input ellenőrzéssel jár, ami nem előnyös, mivel nehezebben olvashatóbbá teszi a kódot, de a különböző *label*-ek az inputoknál változnak mivel típusonként megfelelő kontextusba kell őket helyezni, ezzel szükségessé téve ezt a megoldást.

A javaslat típusainak a definiálása a kliens oldalon találhatóak, mivel a szerver nem szolgáltat mást csak a beküldött javaslatok kezelését, nem szabályozva annak tényleges tartalmát csak típusát, ami minden adatelemnél ugyan-az a szöveg típusú. Ez lehetővé teszi, hogy a felhasználó nagyobb mértékben tudja kifejezni magát az űrlap kitöltésénél és egyszerűbbé teszi, ha később másfajta javaslati típusokat szeretnénk bevezetni a meglévő modellel, mivel így csak a kliens oldali komponensnél kell ezt implementálni és nem kell a szerver oldalon semmit se változtatnunk. Az itt használt modellnek van egy jóváhagyva adattagja, ami megkönnyíti az adminisztrátorok nyomon követési lehetőségét, ami hamis értéket vesz fel minden esetben, amikor egy felhasználó beküldi az űrlapját.

A következő oldalon lévő ábrán (5. ábra) lehet, megfigyelni az előzőekben bemutatott űrlapot két példán keresztül ahol lehet látni a kontextusok közötti különbséget, és hogy milyen formában lehet kitölteni egy ételről szóló ajánlatot.



*5. ábra – Felhasználó általi javaslat űrlapjaira példa*

**4.2.5. Adminisztrátori felhasználói komponensek**

**4.2.5.1. Kezdőlap és navigációs sáv**

Ami után egy adminisztrációs joggal rendelkező felhasználói fiókkal bejelentkezünk akkor egy a főbb adatkezelési oldalakra vezető gombokat tartalmazó kezdőlapra leszünk irányítva. Amikor adminisztrátorként vagyunk bejelentkezve akkor mindig látható a weboldal felső sorában egy navigációs sor (6. ábra), ami lehetővé teszi a felhasználó által a gyors és egyszerű oldalváltást a különböző adatkezelési oldalak között.

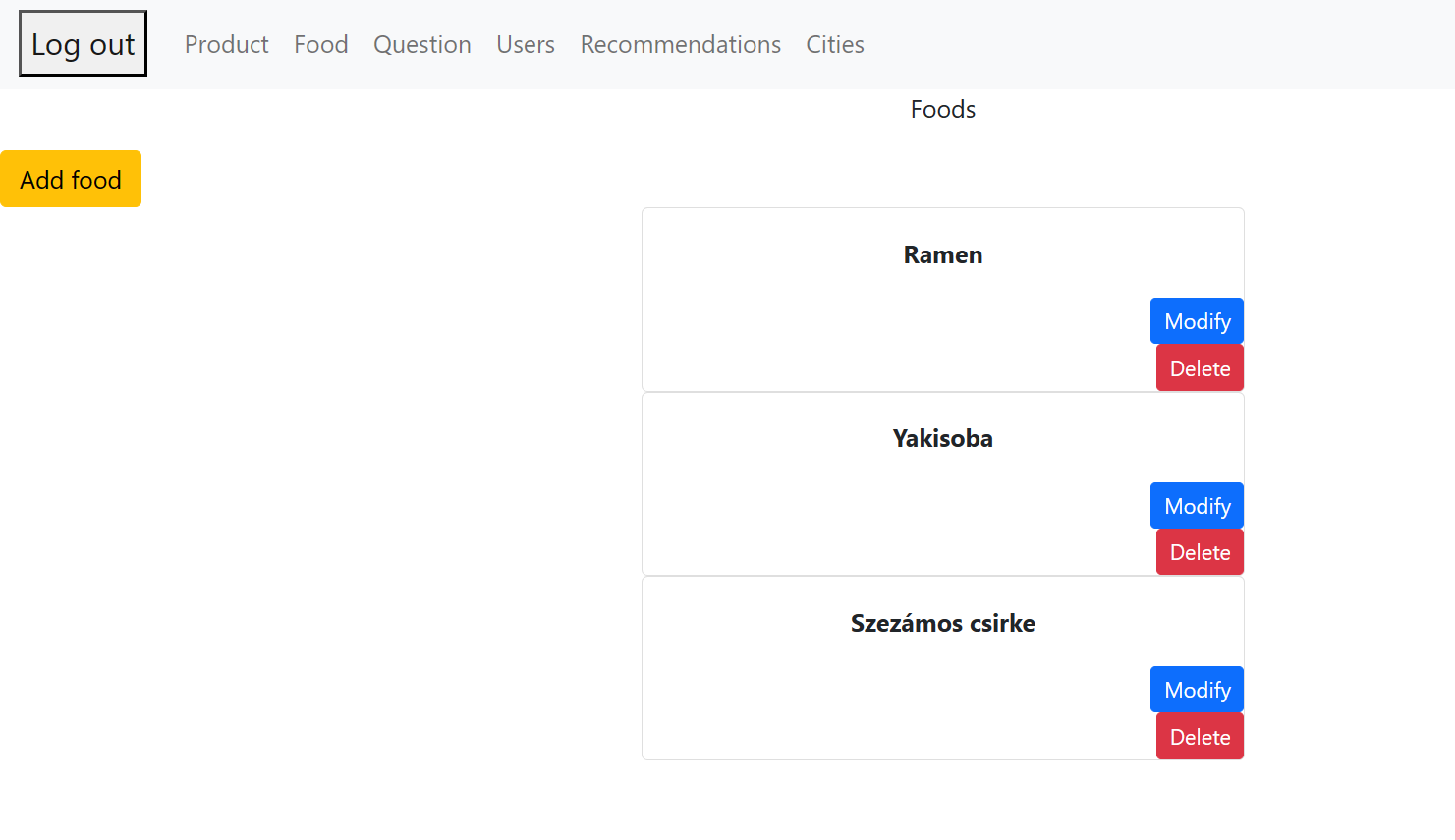
A navigációs sorban még elérhető egy kijelentkezési gomb, aminek az a célja, hogy biztonságosabbá tegye az adminisztrátori munkát. Továbbá elérhető innen az ételajánló fő szolgáltatásunk. Erre azért van szükségünk, hogy ha valami változtatást viszünk végre az adatbázisban, akkor azt másik felhasználóba való bejelentkezés nélkül is le tudjuk ellenőrizni, meg tudjunk győződni róla, hogy minden megfelelően működik.

Az előbb említett kezdőoldalt a kezdeti bejelentkezés után nincs lehetőségünk elérni, csak ha közvetlen linket használunk. Ez azért lett így kialakítva mivel a navigációs sáv ugyan azt a szerepet tölti be, mint a kezdőlap így ennek a kezdeti figyelemfelhíváson kívül más lényeges feladata nincs.

**4.2.5.2. Étel adminisztrációs komponensek**

Amikor az ételeket szeretnénk kezelni akkor a megfelelő menü kiválasztása után a **4.2.4.1. fejezetben** leírtakhoz hasonló **(2. ábrán látható)** lapon kell kiválasztani pontosan milyen város és azon belül milyen konyhának az ételeivel szeretnénk foglalkozni.

Miután ez megtörtént akkor a kiválasztott adatok alapján egymás után kilistázott ételek jelennek, meg amik külön-külön egy dobozban találhatók, és az étel nevén kívül két gombot tartalmaznak (6. ábra).



*6. ábra – Ételek listája és navigációs sáv*

Az egyik gomb az étel törlését teszi lehetővé az előbb kiválasztott városból és konyhából, ha létezik máshol is ugyan olyan nevű étel az nem kerül befolyásolásra. A másik gomb, amit meg lehet találni az ételek mellett az az adott étel módosítását teszi lehetővé.

Mikor módosítani szeretnénk egy ételt, akkor a konyhájához tartozó kérdésekhez kötött válaszokon tudunk módosítani, vagy azokon az éttermeken, ahol azt felszolgálják. Ezeket az opciókat a kezdőoldalhoz hasonló felépítésű oldalon keresztül tudjuk elérni, amit a már említett dobozonként megtalálható módosítás gombon keresztül lehet elérni.

Amikor a válaszokon szeretnénk módosítani, akkor a 3. ábrán láthatóhoz hasonlóan jelennek meg a kezdetben megadott konyhához tartozó kérdések. Ebben az esetben viszont annyi a különbség, hogy a kérdésekhez tartozó válaszok már alapértelmezetten ki vannak töltve a megfelelő értékekkel. Ez úgy lett elérve, hogy már az oldal betöltésekor le lettek kérdezve az adott étel adatai és azok alapján az űrlap létrehozásakor automatikusan a megfelelő opciók értékét állítjuk be a hozzátartozó lekérdezett adat értékére, és így a már létező válaszok maguktól megjelennek a felhasználó számára, ezzel megkönnyítve a módosítás menetét.

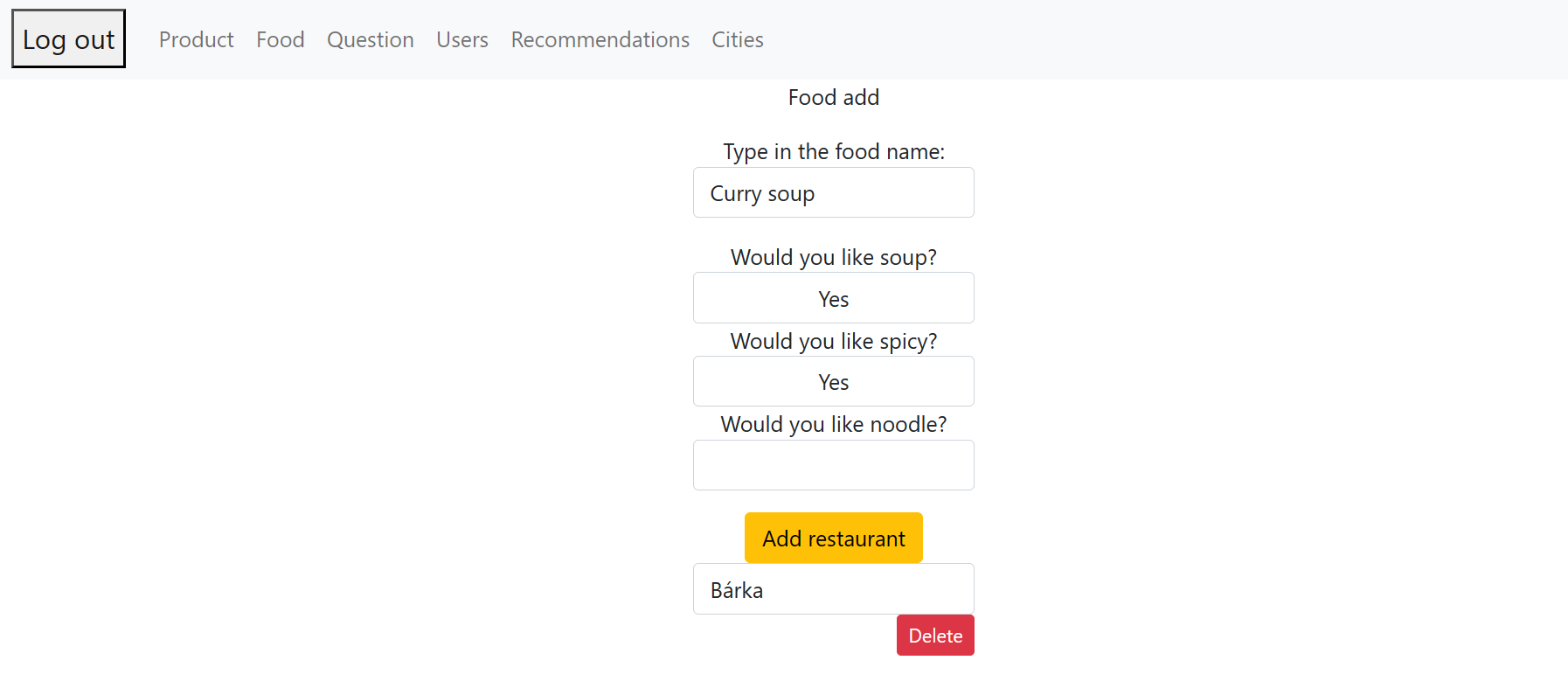
Abban az esetben, amikor az ételhez tartozó éttermeken szeretnénk módosítani, akkor az előbb leírtakhoz hasonlóan függőlegesen egymás után előre kitöltve listázódnak ki az előzőekben felvitt éttermek. Mikor az éttermeken szeretnénk módosítani, akkor lehetőségünk van azoknak csak a nevét átírni mivel minden étterem módosítható szövegdobozban található meg. Ezen kívül lehetőségünk van egy az egyben kitörölni egy éttermet, amit minden étterem szövegdoboza alatt jobb oldalon megtalálható gombbal lehetséges.

Lehetőségünk van az éttermek listájának tetején megtalálható gomb segítségével új üres szövegdoboz beszúrására a lista végére. Ennek az a feladata, hogy lehetőséget adjon új éttermek hozzáadására. Biztonsági okokból kifolyólag nem lehet addig véglegesíteni a változtatásokat mindaddig, amíg nincsen minden szövegdoboz kitöltve. Ez azt jelenti, hogy minden, az űrlaphoz tartozó, a weboldalon megjelenített szövegdoboznak valamilyen formájú karaktert vagy karaktersorozatot kell tartalmaznia. Viszont az lehetséges, hogy minden éttermet, ami az ételhez tartozik, azt kitörlünk, mivel előfordulhat az, hogy minden étterem megszűnik, ami felszolgálja az adott ételt, de nem szeretnénk azt teljesen kitörölni a rendszerből, mivel még vannak a városban éttermek, amik esetleg a későbbiekben felszolgálhatják.

Az éttermek módosítását kiszolgáló űrlap a már korábban többször is említett és leírt *form array* használatával lett megoldva, annyi különbséggel hogy a tömb végére beszúrásra kerülő elem nem teljesen automatikusan történik az oldal betöltésekor. Mivel elméletileg akármennyi éttermet adhatunk egy ételhez, így csak a már meglévő éttermek kerülnek automatikusan a tömbbe. A tömbhöz fűző metódust elérhetővé van téve a felhasználó számára is. Továbbá az éttermet törlő gomb nem azonnali változást tesz az adatbázisban, mivel az csak index alapján törli az előbb említett helyi tömbből az adott elemet.

Az előzőekben említett módosítások véglegesítése után minden esetben a folyamat legelején kiválasztott város és konyha listázott ételeinek az oldalára leszünk visszairányítva. Ezen az oldalon megtalálható még a bal felső sarokban az a gomb, ami arra az oldalra irányít, ahol új ételeket adhatunk az adott város és konyhához.

Az oldalon, ahol az adatbázishoz tudunk új ételt hozzáadni az össze szükséges adat megadására lehetőségünk van. Ezek az adatbeviteli mezők egymás alatt helyezkednek el kezdve egy szövegdobozzal, amibe az étel nevét kell megadni, példa a következő ábrán (7. ábra). Ezek után egymás után sorba függőlegesen a konyha, amibe az új étel tartozik, majd a kérdések és a hozzájuk tartozó válasz legördülő ablakkal. Mindezek után található meg az a rész, ahol az ételt felszolgáló éttermeket lehet hozzáadni. Az itt megtalálható űrlap részegységei ugyan-úgy működnek, mint a már előzőleg bemutatott módosításokért felelős komponensek. Annyi az egyedüli változás, hogy az input adatok nincsenek automatikusan inicializálva semmilyen értékkel.



*7. ábra – Étel hozzáadása űrlap*

**4.2.5.3. Kérdések adminisztrációs komponensek**

Mikor a konyhákhoz tartozó kérdéseket szeretnénk adminisztrálni az adminisztrációs kezdőlapon vagy a navigációs sávban megtalálható releváns gombra kattintás után egy egyszerű legördülő ablakot tartalmazó kezdőoldalra leszünk irányítva. A legördülő ablakra kattintva megjelenik az összes különböző konyhák neve, amit az adatbázis abban a pillanatban tartalmaz.

Mivel nincsen lehetőségünk kliens oldalról a szervertől lekérdezni, hogy milyen különböző konyhák léteznek, ezért azt a kliens oldalon kellett szétválogatni. Ez úgy lett megoldva, hogy az összes várost és azok adatait lekérdezzük, mivel azok tartalmazzák az összes lehetséges konyhát, és a kapott adatokból kiválogatjuk csak a különböző nevű konyhákat egy különálló tömbbe, aminek az elemeit megadjuk a legördülőablakban választható opcióknak.

Ennek a részletes logikai menete az, hogy először a legördülő ablakban választható konyhákat tartalmazó tömb kezdőértékeinek megadjuk az első városnak az összes konyháját. Ezután a maradék városokon végig megyünk egy ciklussal, amin belül egy második ciklus segítségével egyenként végig járjuk az adott városhoz tartozó összes konyhát, amiket megvizsgálunk, hogy már megtalálhatóak-e a választható konyhák tömbjében. Ha pedig még nem tartalmazza őket a tömb, akkor azokat hozzáadjuk, és amikor az adott városnak az összes konyháját ilyen módon ellenőriztük akkor tovább lépünk a soron következő városra, ahol mindezt megismételjük mindaddig, amíg el nem fogynak a városok. A tényleges megvalósítás kódban a cikluson kívül deklarált és inicializált változók nélkül:

for (let i = 1; i < this.cityAll.length; i++) {

for (const cityCuisine of this.cityAll[i].cuisines) {

let notInCuisines = true;

for (const cuisine of this.cuisines) {

if (cuisine == cityCuisine) {

notInCuisines = false; }}

if (notInCuisines) {

this.cuisines.push(cityCuisine); }}}

Miután a felhasználó kiválasztotta a számára megfelelő opciót, a releváns kérdéseket kilistázott oldalra lesz irányítva. Ezen az oldalon szintén külön-külön dobozokban egymás alatt találhatóak meg az előzőekben kiválasztott konyhához tartozó kérdések listája. A dobozokban megtalálható még az adott kérdéssel végezhető műveletek gombjai. Jelenleg ezek még egy törlés gombból álnak, ami az adott kérdést rögtön kitörli az adatbázisból, és egy módosítás gombból, ami tovább irányit az adott kérdést módosító egyszerű űrlapot tartalmazó oldalra. Továbbá még ezen az oldalon a bal felső sarokban található meg az a gomb, ami segítségével egy új kérdés felviteléhez szükséges űrlaphoz leszünk irányítva.

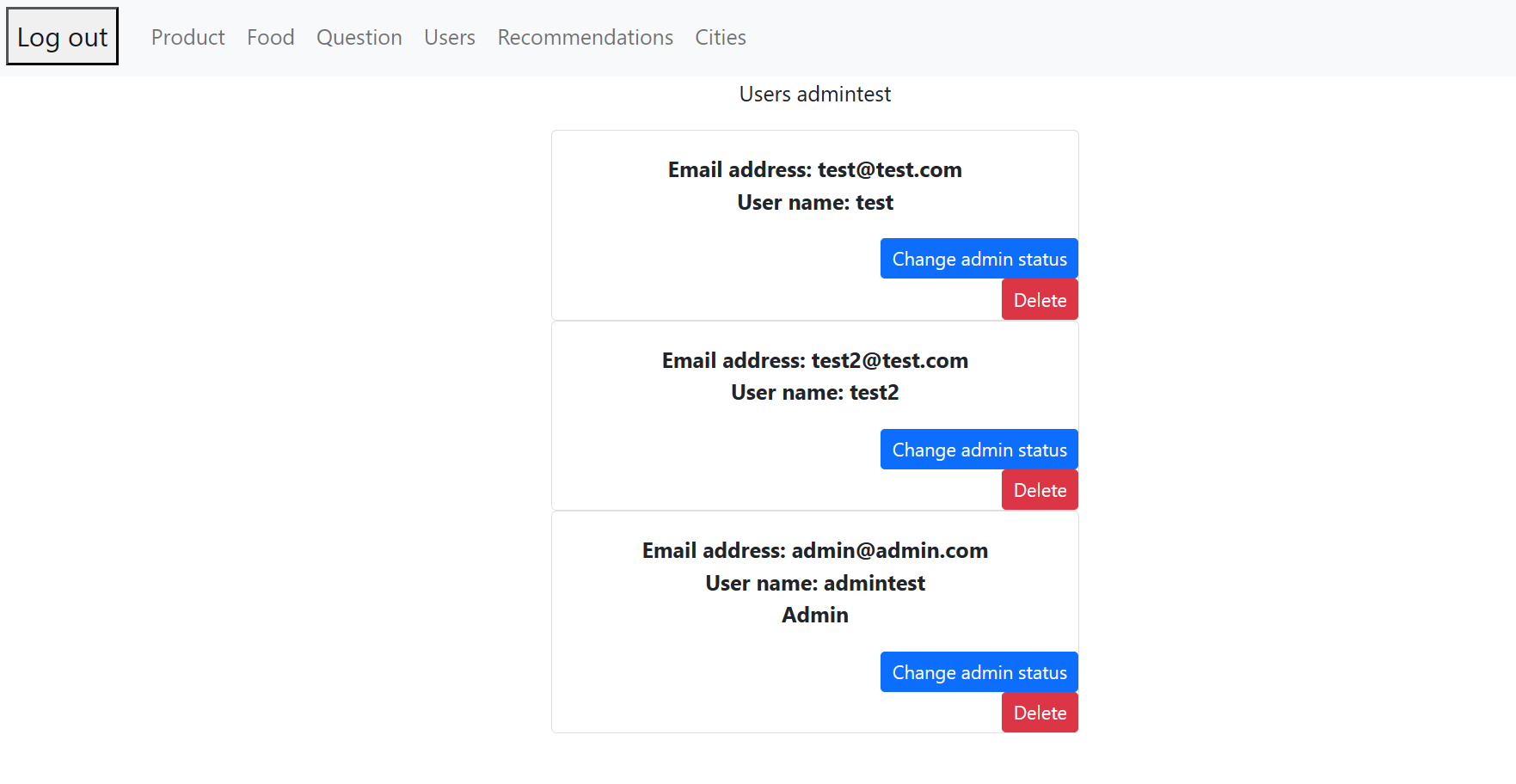
Amikor a felhasználó módosítani szeretne egy kérdést bármilyen okból kifolyólag akkor nincsen más dolga, mint az előbbiekben leírtak alapján kiválasztani a megfelelő konyhát, majd a megjelenő listából kikeresni a módosítani kívánt kérdéshez tartozó megfelelő gombra kattintva kitölteni az oldalon található szövegdobozt a már módosított kérdéssel. Segítségül közvetlenül a szövegdoboz felett megtalálható a régi módosítani kívánt kérdés teljes hosszában, ami a könnyebb módosítás érdekében kimásolható formában van megadva.

Mikor kérdést szeretnénk, hozzáadni egy konyhához akkor a bal felső sarokban megtalálható gombra kattintva megjelenő egy szövegdobozt tartalmazó űrlapot kitöltve tehetünk meg. Mivel nincsen szükségünk más adatra a felhasználótól így a bemeneti mezőn és a hozzátartozó utasításon kívül nem található más meg ezen az oldalon. Fontos megjegyezni azt, hogy szinte lehetetlen leellenőrizni, hogy a beírt kérdés helyes-e. Ezért a felhasználónak fokozottan oda kell figyelnie, hogy helyesen adja meg az új vagy módosított kérdést, ne tartalmazzon helyesírási hibát és a használt mondat végi írásjel megfelelő legyen.

**4.2.5.4. Felhasználók adminisztrátori komponensek**

Amikor egy adminisztrátor szeretné, kezelni az adatbázisban lévő felhasználói adatokat akkor azt a kezdőoldalon vagy a navigációs sávban megtalálható megfelelő gomb megnyomásával teheti, meg mint eddig minden adminisztrációs műveletnél. Miután ez megtörtén a felhasználó tovább lesz irányítva egy oldalra ahol az előzőkhez hasonlóan egymás alatt külön-külön dobozokban vannak megjelenítve egy-egy felhasználó lényeges adtai.

Ezek az adatok nem tartalmaznak semmilyen adatot, ami nem kerülhetne a nyilvánosság számára, okozna biztonsági problémákat. A két fő megjelenített adat minden felhasználónál az az adott felhasználó e-mail címe és felhasználó neve, amivel regisztrált, látható a következő ábrán (8. ábra). Ezekre azért van, szükség mivel ezzel a két adattal lehet pontosan beazonosítani egy felhasználót a leghamarabb mivel jelenleg egy felhasználónak csak az e-mail címének kell teljesen egyedinek lennie viszont sok eset létezik ahol egy-egy e-mail cím nem teljesen egyforma, de akkora a kettő közötti hasonlóság hogy azt elsőre nem lehet észrevenni csak gondos átolvasás után. Ezért van az, hogy az e-mail cím mellett még a felhasználónév is meg van jelenítve, aminek nem kell feltétlen egyedinek lennie, mivel annak az esélye, hogy két felhasználó e-mail címe szinte megkülönbözhetetlen legyen és ugyan azt vagy nagyban hasonló felhasználó nevet válasszon szinte elenyésző.



*8. ábra – Felhasználók kezelőfelülete*

Az előbb említett felhasználói adatok után még egyes esetekben megtalálható egy *Admin* felirat, ami feladata az, hogy az egyes adminisztrációs jogokkal rendelkező felhasználói fiókokat könnyebb legyen beazonosítani. Jelenleg adminisztrációs jogot tud adni és elvenni minden felhasználónak kivéve saját magát minden adminisztrátor. Ezt az egyes felhasználók dobozában megtalálható megfelelő gomb megnyomásával történik, ami után az adott felhasználó *Admin* státusza megváltozik. Ez a változás szemmel látható hisz megjelenik a felhasználó adatai alatt az *Admin* felirat vagy a jog elvétele esetén eltűnik az.

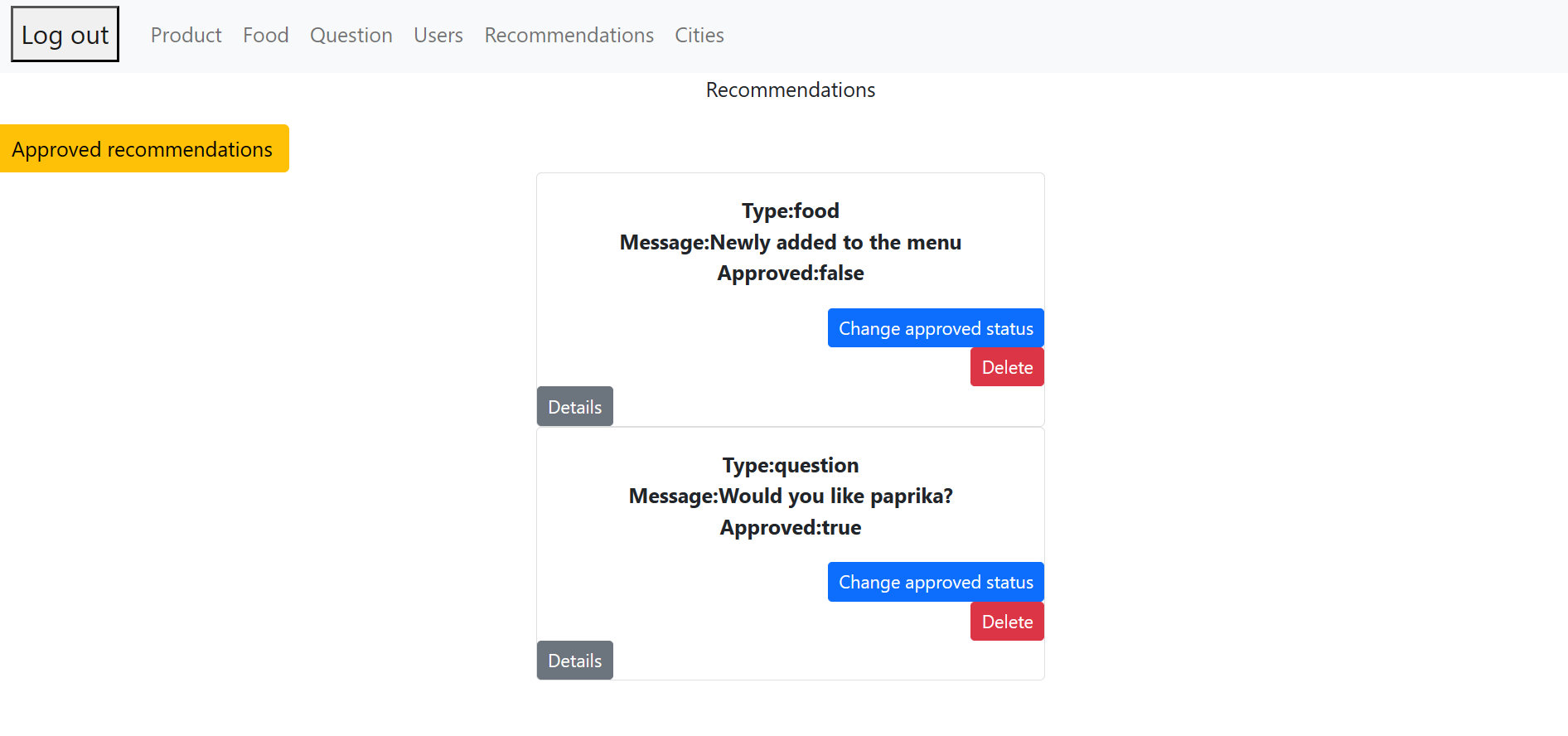
Ugyan itt még felhasználókként megtalálható egy törlés gomb, ami jelenleg lehetővé teszi az adott adminisztrátor számára, hogy bármilyen felhasználói fiókot kitöröljön az adatbázisból a sajátján kívül. Adminisztrációs státuszváltoztatást és fióktörlést biztonsági okokból és az adminisztrátor jövőbeli tehetetlenségének az elkerülésének az érdekében nincsen engedélyezve, hogy az adott felhasználó elvégezze azt saját magán.

**4.2.5.5. Javaslatok adminisztrátori komponensek**

Hogyha szeretnénk kezelni a felhasználók által beküldött javaslatokat az a megfelelő gombok segítségével navigálva tehetjük meg ugyan olyan módon, mint a többi adminisztrációs oldal esetében. Mikor megnyitjuk a javaslatokat kezelő fő oldalt akkor az összes jelenleg beküldött javaslattal találhatjuk magunkat szemben, amik hasonlóan az előző kezelőoldalakhoz van kilistázva.

A különböző javaslatok dobozainak a felső részében középre igazítva találhatjuk meg az általánosságba vett legfontosabb adott ajánlathoz tartozó adatokat. Ez magába foglalja milyen típusú az adott javaslat, mint például étel, kérdés vagy étterem, látható még a felhasználó által hozzáfűzött üzenet vagy leírás továbbá még tartalmazza azt is, hogy az adott javaslat jóvá lett e már hagyva vagy pedig nem. Alapértelmezetten minden egyes felhasználók által beküldött ilyen űrlap hamis értéket vesz fel, amit ha megszeretnénk, változtatni azt a minden dobozban megtalálható státusz változtató gomb megnyomásával tehetünk. Ennek a gombnak a segítségével az adminisztrátor megváltoztathatja bármelyik javaslatnak az elfogadott státuszát, ami azt jelenti, hogy ami még nincs, elfogadva azt elfogadhatja, amelyik pedig már el volt fogadva az akár vissza is utasíthatja valamilyen okból kifolyólag.

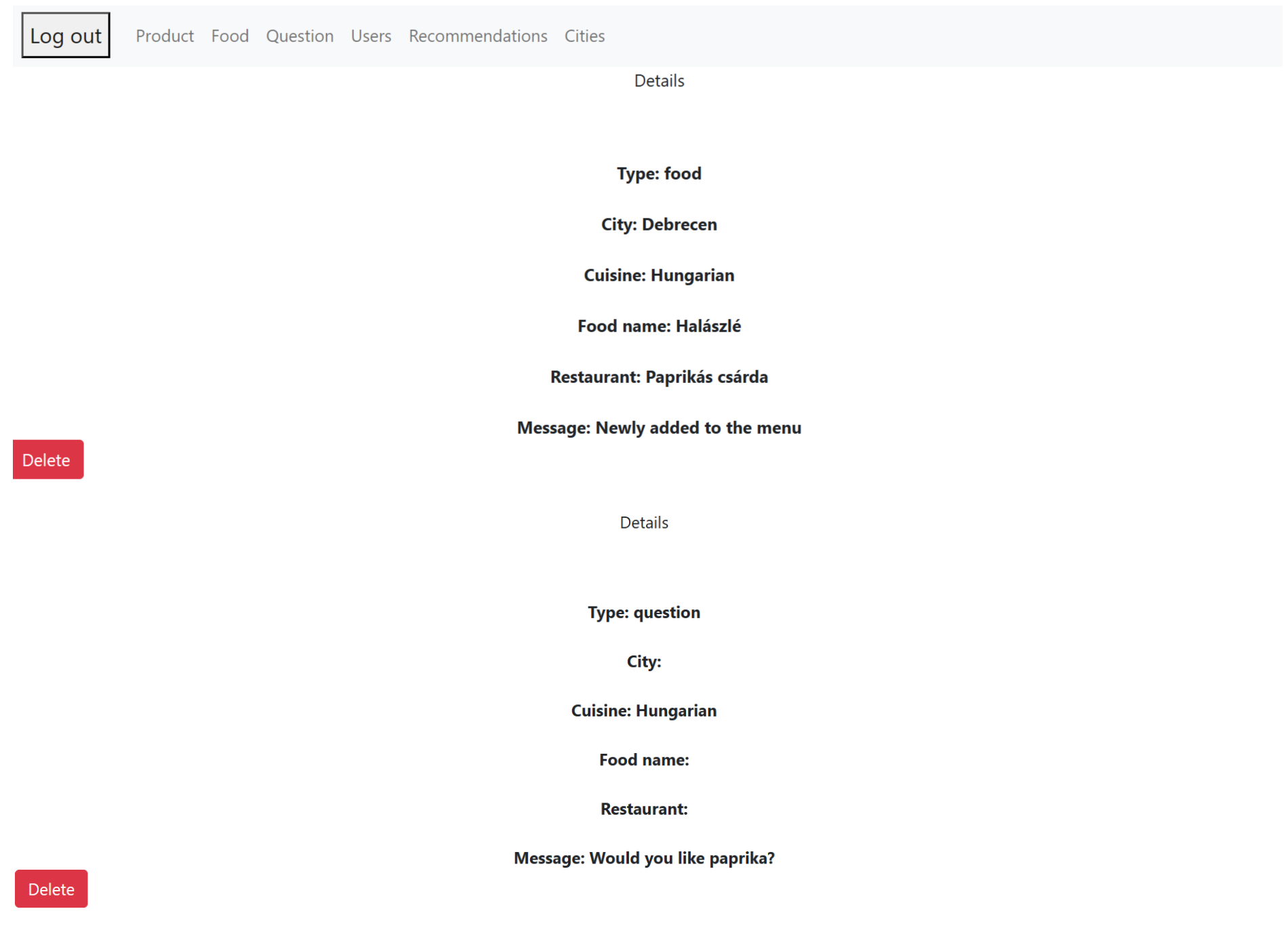
Ennek a státusznak a változtatása valós időben történik az adatbázisban is továbbá mikor egy már elfogadott javaslatot elutasítunk, akkor a nem kerül törlésre hisz lehetséges, hogy a későbbiekben még ismét el lesz fogadva. A státusz változtatása a háttérben nem áll másból csak egy egyszerű *put* *http* kérésből, aminek nem kell megadni csak az adott javaslat id-ját, amiket még az oldal betöltésekor lekérdeztünk szervertől, mivel a szerverünk biztosítja számunkra ennek a műveletnek az egyszerű elvégzését. A törlésre lehetőséget kap az adminisztrátor ezen az oldalon, aminek a gombját szintén a javaslatokhoz tartozó dobozban található meg ennek megnyomása hatására az adott javaslat rögtön törlésre kerül az adatbázisból. A következő ábrán (9. ábra) látható a javaslatok kilistázása.



*9. ábra – Felhasználók javaslatai*

Továbbá lehetőségünk van az egyes beküldött javaslatokat részletesebben is megtekinteni, amit szintén az egyes dobozokban a bal alsó sarokban megtalálható *részletek* gombra kattintva tehetünk, meg ami segítségével megtekinthetjük az egyes javaslatok összes adatát. Ugyanekkor még megtekinthetjük az összes elfogadott javaslatot az oldal bal felső sarkában megtalálható gomb segítségével.

Ha szeretnénk megtekinteni az adott javaslat részleteit, azt az előzőekben leírt gomb segítségével tehetjük, meg ami tovább irányit minket egy oldalra, ami megjeleníti az összes adatát a beküldött javaslatnak. Ez az oldal tartalmazza egy javaslat összes lehetséges adatát figyelmen kívül hagyva, hogy azok mindegyike ki van e töltve. Ez azért van így mivel a típustól függően nem mindegyik tartalmazza ugyan azokat az adatelemeket viszont több szempontból is előnyös egy egységes megjelenítés használata. Az egyik főbb szempont a megjelenés egyszerűbb kivitelezése és egységes modell használata a másik pedig a felhasználó gyorsabban tud alkalmazkodni az adatok kiolvasásához, ha nem változik a megjelenített adatok mennyisége és elhelyezkedése, még akkor is, ha egy-egy adott mező üres, hiszen egy idő után megszokjuk, hogy pontosan hol és milyen adatra van szükségünk. Ezt a folyamatot az is elősegítheti, hogy már a kilistázásnál látjuk, és előre tudjuk milyen típusú javaslatot fogunk megnyitni.



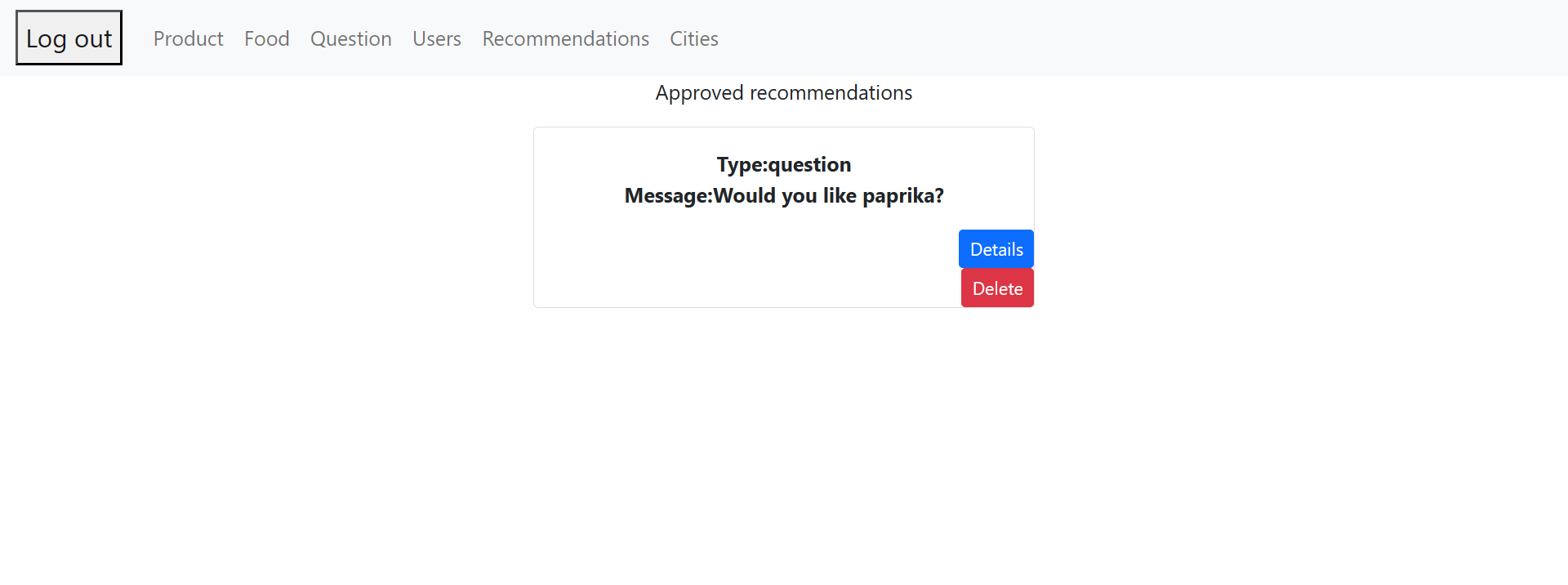
*10. ábra – Javaslatok részletei*

A fentebbi ábrán (10. ábra) látható hogy ténylegesen hogyan is néz ki egy-egy különböző javaslatnak a részletei. Mint az láthatjuk az első és egyik legfontosabb adat az az adott javaslat típusa, ami segítségével tudhatjuk, hogy ténylegesen mit is kell keresnünk az adatok közül. A másik fontos adat, ami minden esetben tartalmaz valamit az az üzenet vagy leírás. Ez a rész tartalmilag sok mindent tartalmazhat, hiszen a felhasználó sokféleképpen értelmezheti, hogy pontosan mire is lennénk kíváncsiak. Az esetek nagyrészében ha a felhasználó megfelelően értelmezte a kérésünket, akkor itt csak rövid leírások lesznek számunkra, hogy pontosan miért is tette ezt a javaslatot a felhasználó esetleg kiegészítő információk, amikre szükségünk lehet. Bár erre lehet kivétel, is mint azt a feni ábra (10. ábra) második példájánál láthatjuk ahol egy kérdés hozzáadását javasolták és a tényleges kérdést az *üzenet* mezőbe adták meg mivel nincsen lehetőség más helyen ezt megadni mivel ezt a funkciót tökéletesen el tudja látni nagyobb nehézségek nélkül ez az egy mező is így nincs szükségünk felesleges létrehoznunk erre egy dedikált adattagot.

A többi mezők azért kerültek külön adattagokba megjelenítve mivel azokat egyszerre több típus is használhatja és olyan fontos adatokat tartalmaznak, amik gyors és egyszerűbb megtalálása nagyban elősegítheti az adminisztrátorok dolgát. Ezért mivel a kérdéseknél ténylegesen szükséges adat csak kettő van ezért az, hogy nem vettük magát a kérdést külön adattagként az jelentősen nem csökkentené az adminisztrátorok idejét az egyes adatok kiolvasásánál.

Továbbá még észre lehet venni, hogy itt, nincsen megjelenítve az elfogadottsági státusza a javaslatnak. Ez azért van így mivel ennek az oldalnak az elsődleges célja az, hogy azokat az adatokat jelenítse meg részletesen, amikhez az első körökben nincs szükségünk, nem fontosak. Ugyanakkor azért sincsen, megjelenítve mivel erre kifejezetten van egy külön oldalunk, amiről majd a későbbiekben lesz szó. Még megfigyelhető az is hogy az egyetlen gomb, ami az oldalon megtalálható az egy törlési gomb, ami azért került ide, hogy az adminisztrátornak az adott javaslat törlését el tudja végezni helyben és ne kelljen azt az előző esetlegesen hosszú listából újból megkeresnie és ott törölni. Ha az itt található gomb segítségével töröljük az adott javaslatot, akkor a törlést követően a listát tartalmazó kezdőoldalra leszünk irányítva.

Hogyha nem szeretnénk egyenként megkeresni a már elfogadott javaslatokat, akkor a 9. ábrán látható kezdőoldal bal felső sarkában megtalálható gombra kattintva kilistázhatjuk az összes elfogadott javaslatot a könnyebb kezelésük érdekében. Ezt a listát a következő ábrán (11. ábra) láthatjuk ahol csak egy javaslat található annak ellenére, hogy több javaslat is van függőben (9. ábrával egyidőben készült).



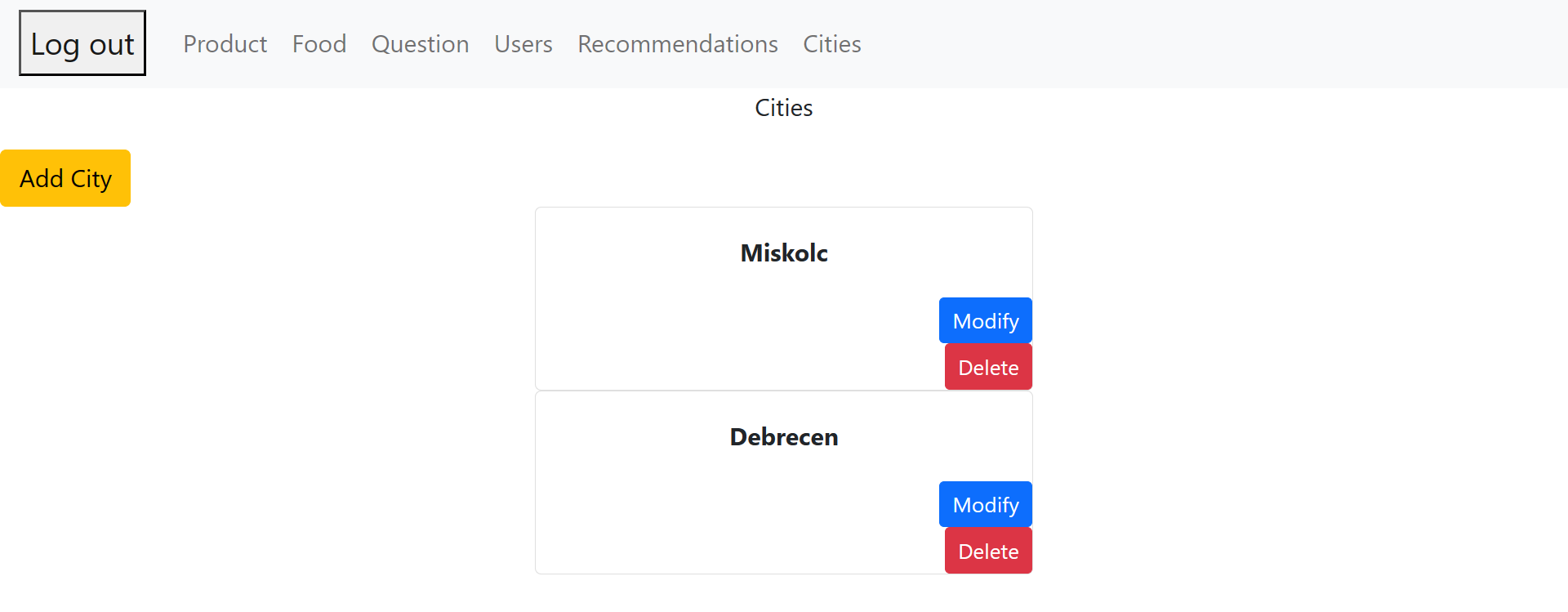
*11. ábra – Elfogadott javaslatok*

Az előző ábrán (11. ábra) látható hogy itt is ugyan azok az adatok vannak megjelenítve, mint az összes javaslat listázásánál kivéve a státuszt jelölő adattagot. Ez azért van így mivel ebben a listában nincsennek benne csak a már elfogadott javaslatok ezért az a mező fölöslegessé válik. Viszont az is megfigyelhető hogy ugyan úgy kevés információ áll a rendelkezésünkre az egy-egy javaslatról, aminek az az oka, hogy ha esetlegesen sok lenne a felhalmozott javaslat, akkor könnyebb legyen megtalálni a számunkra relevánsat. Ebből kifolyólag lehetőségünk van itt is megtekinteni az adott javaslatnak a *részleteit* az adott dobozban megtalálható megfelelő gomb használatával, ami tovább irányit minket a már az előzőekben bemutatott oldalra, amire a 10. ábrán láthatunk példákat. Még ugyanitt megtalálható egy gomb, amit a felhasználó az adott elfogadott javaslatnak a törlésére használhat, így nem kell azért feleslegesen megnyitni a *részleteket* vagy kikeresnünk a főoldalon, hogy ki tudjuk azt törölni bármilyen okból kifolyólag.

A háttérben ennek az oldalnak a működése egyszerű mivel nem volt szükség csak a megfelelő modellel definiálni egy tömböt, amit az oldal betöltésekor feltöltünk, a megfelelő *http* lekérdezés segítségével mivel a szerver számunkra biztosítja ezt a szolgáltatást. Fontos megjegyezni, hogy a jelenlegi állapotban (az írás idejében) ha erről az oldalról érjük el egy javaslatnak a *részleteit* és az itt található gombbal töröljük az abban az esetben is a javaslatok fő oldalára leszünk irányítva (9. ábra), viszont ha a helyi (11. ábra) törlés gombot használjuk akkor, mint a többi hasonló esetben nem leszünk átirányítva sehova és élőben közvetlenül frissítjük így az adatbázist.

**4.2.5.6. Városok adminisztrációs komponensek**

Miután a navigációs gombok segítségével sikerült elérnünk a városokat kezelő kezdőoldalt szembesülünk a következő ábrán (12. ábra) látható listával. A listának az elemei ugyan úgy, elemenként egy-egy dobozban a hozzájuk tartozó gombokkal együtt egymás alatt helyezkednek el, mint az összes többi hasonló adatokat kilistázó oldal.



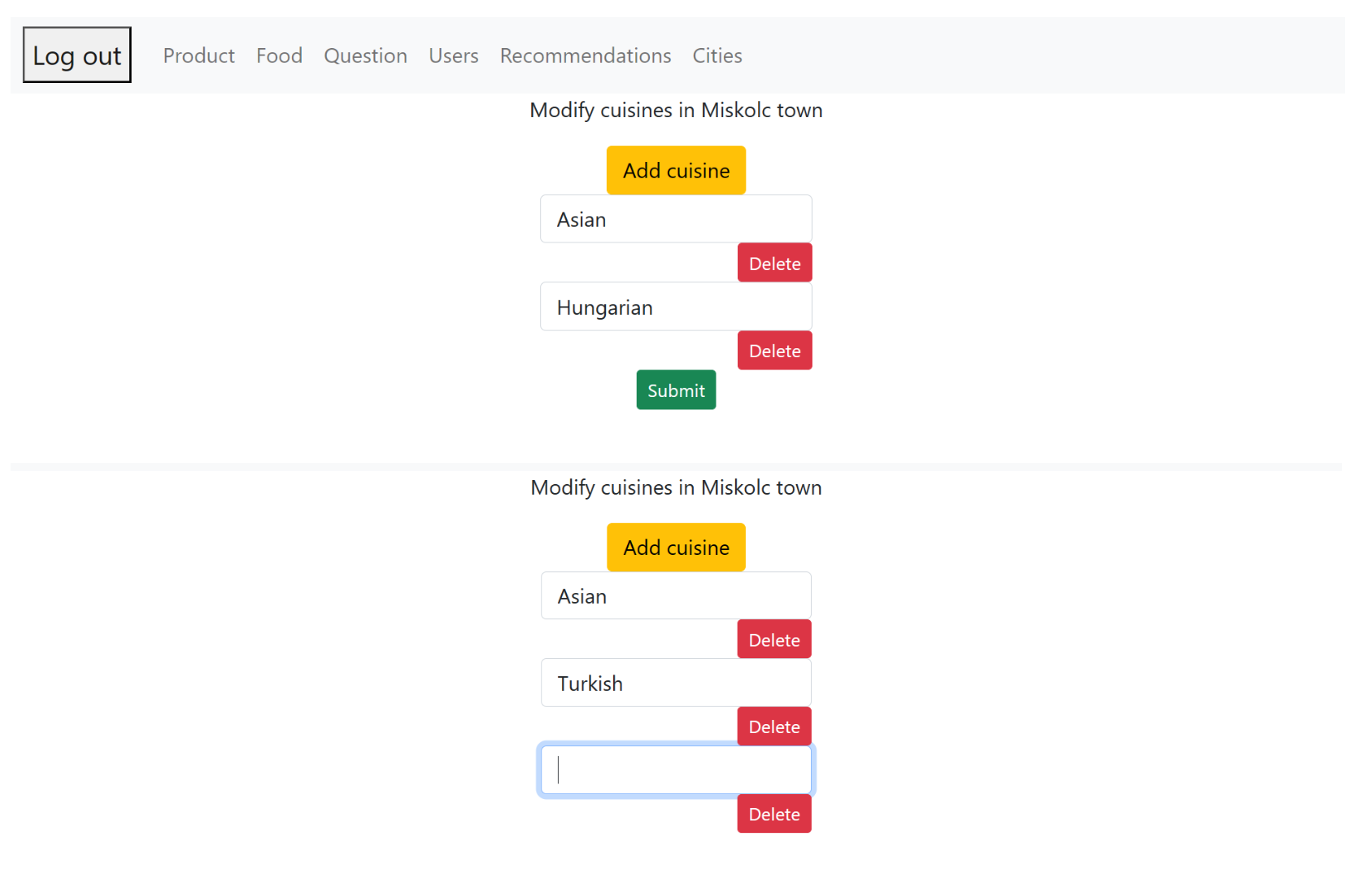
*12. ábra – Városok listája*

A lista elemeinél adatként egyedül csak a városoknak a neve van megjelenítve. Mivel az alkalmazás egyik fő célja a nagyvárosok integrálása a rendszerbe ezért a városnevek egyezésének a problémája elenyésző. Természetesen előfordulhat, hogy két ugyan olyan nevű várost szeretnénk felvinni a rendszerbe, amit nem tehetünk meg mivel fontos, hogy egyértelműen megkülönböztethetők legyenek egymástól a városok. Ebben az esetben egy egyszerű megoldás lehet, hogy megkülönböztető jeleket adunk a városok neve elé vagy után, például megye nevek rövidítése, ország azonosító vagy a város irányítószáma.

Mint azt láthatjuk az előző ábrán (12. ábra) minden város rendelkezik egy *módosítás* gombbal, ami segítségével megváltoztathatjuk utólag az adott városban megtalálható konyhákat. Miután megnyitottuk az adott városhoz tartozó módosítási felületet megfigyelhetjük, hogy a lap tetején lévő feliratban, ami jelzi, hogy éppen milyen oldalon vagyunk, megtalálható a városnak a neve, amit éppen módosítani fogunk.

Ezen kívül még megtalálható kezdetben egymás alatti szerkeszthető szövegdobozokban a módosítás pillanatában a városhoz tartozó konyhák is. Továbbá a szövegdobozok felett megtalálható egy gomb, ami segítségével újabb üres szövegdobozokat tudunk hozzáadni a már meglévőekhez. Ezek az új üres szövegdobozok mindig a legalsó/legutolsó után kerülnek beszúrásra, amit a 13. ábrán is láthatunk.

Lehetőségünk van egy már meglévő konyhát átírni, a konyhákat tartalmazó szövegdobozok törlésére, amit minden szövegdoboz alatt megtalálható gomb segítségével tehetünk meg. Ezek a törlések nem véglegesek, viszont ha egy szövegdobozt a gomb segítségével eltávolítottunk azt nem tudjuk tartalmával együtt visszahozni. Fontos megjegyezni, hogy minden nemű változtatás csak a megfelelően kitöltött űrlap beküldése után válik véglegessé. Az előbb leírtaktra példát lehet látni az alábbi ábrán (13. ábra):



*13. ábra – Város konyháinak a módosítása*

A fentebbi ábrán (13. ábra) látható űrlapok felépítése és működés megegyezik azzal az űrlappal, amit az ételeket felszolgáló éttermeknek a módosításánál találhatunk. Ezeknek az űrlapoknak a működése viszonylag összetett annak ellenére, hogy ténylegesen csak egy szövegeket tartalmazó tömb lesz a kimenete. Ez a bonyolultság abból adódik, hogy a tömbnek váltakozó hosszúságúnak kell lennie azon belül pedig tudnunk, kell kezelni minden elemét a tömbnek külön-külön.

A megoldásom erre a problémára a már említett *form array* használata volt viszont ennek a használata nem annyira egyszerű ebben az esetben, mint egy egyszerű form-nak ahol csak a kért adattagokat kell megadnunk és azok követelményeiket. Ahhoz hogy számunkra megfelelően működjön először egy *form group*-ba létre kell hozni egy tömböt ami majd összefogja az elemeinket, az alábbi kód ezt be is mutatja:

cuisinesForm = this.fb.group({

cuisines: this.fb.array([])

});

Ezután létre kell hoznunk még egy *form group*-ot ami tartalmazni fogja egy-egy elemnek a leírását, követelményeit, amit az előbb leírt tömbünk fog tartalmazni ebben az esetben ez a 13. ábrán látható egy-egy szövegdobozt jelenti. Ezután létre kell hoznunk egy függvényt, ami segítségével egy üres elemet hozzá tudunk adni a tömbhöz, ezt az alábbi kód segítségével valósítható meg:

addCuisine() {

this.cuisines.push(this.cuisineForm());

}

Továbbá ha felszeretnénk dolgozni az űrlapban megtalálható tömböt, akkor azt át kell alakítanunk *form array*-é amit meglehet tenni egyenként minden alakalommal amikor szükségünk van rá viszont mivel többször is van rá szükségünk és az ehhez szükséges végleges kód túl hosszú lenne ezért azt egy get függvénybe csoportosítottam ki amit a rendszer automatikusan egy listaként fog kezelni. Ennek a használta az előző kódrészletben meg is figyelhető, a get metódusnak a felépítése pedig a következő:

get cuisines(): FormArray {

return this.cuisinesForm.get("cuisines") as FormArray;

}

Mivel nem szeretnénk adatvesztést és a felhasználó barátságát is növeli az alkalmazásunknak a városban már megtalálható, az adatbázisból lekérdezett konyhákkal feltöltjük kezdőértékekkel a tömböt. Ez a már bemutatott kódok segítségével történik az alábbi módon:

for (let i=0;i<this.city.cuisines.length;i++){

this.addCuisine();

this.cuisines.at(i).get("cuisine")?.setValue(this.city.cuisines[i]);

}

Itt lehet látni, hogy először hozzáadunk egy üres elemet a tömbünkhöz majd miután ez megtörtént az előbb hozzáadott elemet feltöltjük, az ugyan olyan indexel rendelkező városhoz tartozó konyhával. Mindezt addig csináljuk, míg az összes elemén végig nem mentünk a lekérdezett konyháknak a tömbjén.

Mikor az adatokat ki szeretnénk nyerni az előbb leírt űrlapból, és azokkal frissíteni szeretnénk az adatbázist, akkor az előbb leírtakhoz hasonlóan járunk el, hogy itt az űrlapban megtalálható tömbön megyünk végig és azok elemeit alakítjuk és helyezzük át egy egyszerű tömbbe, amit később továbbadunk a város modellünknek, amivel frissíteni szeretnénk az adatbázisban lévő adatokat. Ennek a ciklusnak a kódja:

for (let i = 0; i < this.cuisines.length; i++) {

const element = this.cuisines.at(i).get("cuisine")?.value;

cuisines.push(element);

}

Az előzőkben leírt űrlapnak a felhasználó felé való megjelenítését hasonlóan egymásba építve, erősen figyelve a megnevezésekre és a megjelenített elemnek a típusára, kellett megvalósítani. Ez a gyakorlatban azt jelenti, hogy az oldalon megjelenő űrlap egy egybefogó *form*-ból áll, ami tartalmazza az összes adatelemet, amit az űrlap segítségével be szeretnénk kérni. Itt nem csak összetetteb adatok bekérdezése történhet meg, hanem más egyszerűbb, például szöveges inputot is elhelyezhetünk, amire a következőkben is láthatunk példát bár az ételek hozzáadásánál is ugyan ez a módszer van használva azzal a különbséggel, hogy ott nem egy, hanem kettő *form array* van használva az éttermek és a kérdések válaszainak a bekérdezésére.

Az előbb említett összefoglaló *form*-on belül egy külön konténerben kell definiálnunk a megjelenítésre szánt *form array*-t, amit ennek hatására úgy kell kezelni mintha az egy különálló *form* lenne a saját már definiált adatelemeivel. Mivel a *form array* úgy viselkedik, mint egy objektumokat tartalmazó tömb ezért szükségünk van arra, hogy végig járjuk a tömb összes elemét miközben egy indexet is használunk, ami segítségével tudjuk meghatározni, hogy a tömb melyik elemét és annak adattagjait jelenítsük meg.

Az indexekkel ellátott konténerekbe tudjuk csak elhelyezni azokat a gombokat, amelyek az adott elemre hatással lévő függvényeket használnak, mivel ezek az indexek úgy viselkednek, mint a lokális változók így azokat az adott konténeren kívülről nem lehet elérni közvetlenül. Viszont az új elem hozzáadására szolgáló gomb az oldal bármely pontján elhelyezhető mivel annak a hatása nem függ semmilyen változótól, ugyan ezekből az indokokból kifolyólag, ha külön függvény végzi az adatok összegyűjtését, akkor a véglegesítésre szolgáló gombot is bárhova elhelyezhetjük az oldalon. Az előzőekben leírtakra egy tényleges *html* kód példa, ami bemutatja a 13. ábrán szereplő űrlapot:

<button (click)="addCuisine()" class="btn-warning btn btn-md">Add cuisine</button>

<form [formGroup]="cuisinesForm">

<div formArrayName="cuisines">

<div \*ngFor="let cuisine of cuisines.controls;let i=index">

<div [formGroupName]="i">

<input class="form-control" formControlName="cuisine" type="text">

</div>

<div class="text-end">

<button (click)="delete(i)" class="btn btn-danger btn-sm ">Delete</button>

</div>

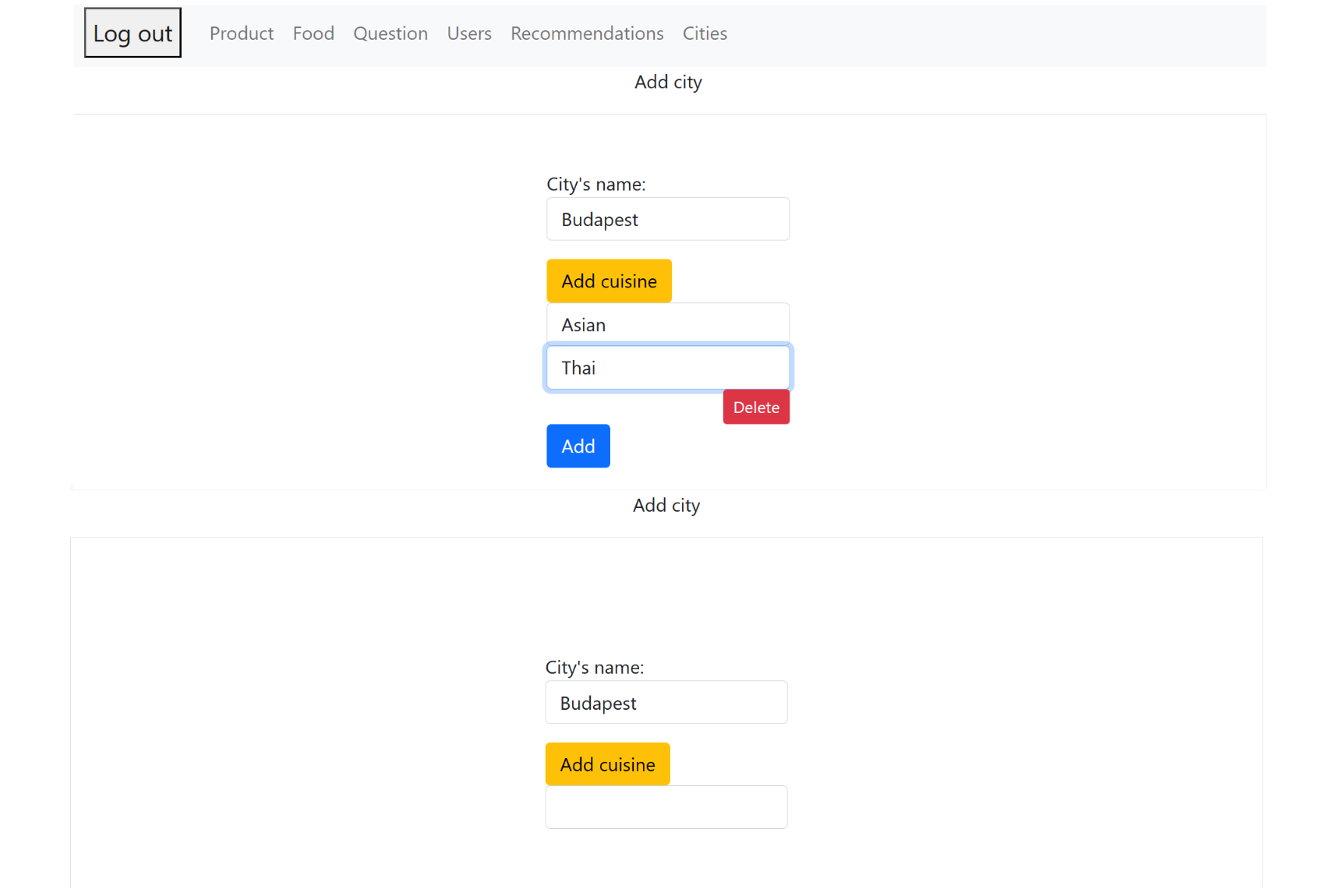
</div>

</div>

<button (click)="done()" [hidden]="cuisinesForm.invalid" class="btn btn-sm btn-success">Submit</button>

</form>

Hogyha szeretnénk új várost hozzáadni a már meglevőkhöz azt a 12. ábrán látható bal felső sarokban megtalálható gomb segítségével tehetjük meg. Amikor megjelenik előttünk az oldal ahol új várossal bővíthetjük az adatbázist, akkor észre lehet venni hogy a kitöltendő űrlap nagyban hasonlít az előbb bemutatott módosításnál használt űrlaphoz. Itt annyi a különbség, hogy a város nevét is meg kell adni és alapértelmezetten már egy konyhának az üres szövegdoboza létre van hozva, amiket kötelezően ki kell tölteni valamilyen karaktersorozattal. Természetesen a konyhák listája felett megtalálható gomb segítségével újabb szövegdobozokat adhatunk hozzá a már meglévőkhöz ezzel egyszerre több konyhával létrehozni az adott várost.



*14. ábra – Város hozzáadás*

Az előbb leírtak megfigyelhetők az előző ábrán (14. ábra) továbbá az is látható hogy az első konyhaként szolgáló szövegdobozon kívül minden másiknak a törlésére lehetőségünk van. Ahhoz hogy véglegesíteni tudjuk a létrehozni kívánt városunkat a város nevének egyedinek kell lennie a már az előzőekben leírtak miatt, továbbá még legalább egy konyhával rendelkeznie kell a városunknak a létrehozása idején mivel ez szükséges ahhoz, hogy a továbbiakban tudjuk vele dolgozni, mert feltételezhető hogy a város létrehozása után ételeket és azzal éttermeket is fogunk létrehozni, amik megtalálhatóak a városban.

Ennek a komponensnek a működése és felépítése ugyan olyan, mint az előbb bemutatott módosításra használt komponens azzal a különbséggel, hogy itt még egy különálló bemenet is létezi és az első konyha bemenetének törlésére nem adunk lehetőséget továbbá az űrlap beküldésekor nem egy már meglévő bejegyzést frissítünk az adatbázisban, hanem egy teljesen újat adunk hozzá ahhoz.

**4.2.5.7. Egyéb fontosabb komponensek**

Az egyik fontos komponens, amiről még nem volt szó az az egész kliensnek az útvonal választásáért felelős komponens. Ebben a komponensben található meg minden url útvonal és a hozzá tartozó komponens pár. Itt tudjuk megadni, hogy az url-ben használt változóknak mi legyen a neve és azok elhelyezkedését. A megadott url-hez lehet társítani az az által elérhető komponenst, amit akkor tudunk elérni, ha a böngészőben az itt megadott url címet megfelelően használjuk, érvényes változók és a pontos cím beírása. Ugyanakkor még be is tudjuk itt állítani címenként egy *guard* használatával, hogy milyen formában és milyen jogosultság bírtokában tudja egy-egy felhasználó használni/elérni az adott címet. Erre egy példa az alábbi kód, ami az ételek listájához vezet, amit csak adminisztrátorok érhetnek el: {path: 'admin/food-list/:town/:cuisine', component: AdminFoodListComponent, canActivate: [AdminGuard]}. Az előző példa kódban látható módon, logika alapján van minden komponens útvonala beállítva a megfelelő *guard-* dal együtt.

Az előbb említett *guard* komponensek fő feladata egy általunk megadott algoritmus alapján eldönteni, hogy az adott funkció elvégezhető-e vagy sem. Ez az esetek nagy részében felhasználói adatok feldolgozása után történő döntéshozatalt jelent, ami értéke igaz vagy hamis. Fontos megjegyezni, hogy egy-egy *guard* többféle típusú hozzáférést is tud egyszerre biztosítani viszont érdemes egyszerre csak egyféle hozzáférési státuszt kezelnünk, például adminisztrátori jogok. Az alábbi kód be is mutatja az adminisztrátori *guard* működését: canActivate(

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

if (this.service.isAdmin()) {

return true;

}

return false;

}.

Még érdemes megemlíteni az egész kliensben lévő komponensek megjelenítéséért felelős *app* komponenst. Ennek a komponensnek a fő feladata egy keretet, kiinduló pontot adni az összes többi komponensnek. Az általános megoldás úgy néz ki, hogy ebben a komponensben adjuk meg azokat a részeket, amit minden komponensnél szeretnénk használni, majd doboz szerűen dinamikusan beszúrjuk ebbe az oldalba az éppen használni kívánt komponenst url cím alapján. Ezt az alábbi html kód be is mutatja ahol a navigációs sáv csak adminisztrációs jogokkal rendelkezőknek jelenik meg:

<app-nav [hidden]="!admin"></app-nav>

<router-outlet></router-outlet>

A kékkel jelölt parancsok nem hagyományos html kódok mivel azok egy-egy komponenst reprezentálnak, amit az *Angular* biztosít.

Végezetül érdemes megemlíteni azt a komponens, ami felelős a *http* kérések és azokra kapott válaszok feldolgozásáért. Ez az *interceptor* komponens nem kötelező szerves része egy ilyen alkalmazásnak viszont érdemes használni mivel lehetővé teszi az egyes kérésekre a globális hibakezelést. Lehetséges egyszerre több ilyen komponenst is definiálni viszont egyszerre csak egy komponens dolgozhatja fel az adott kérést vagy választ, ezért ha szükséges lehetőségünk van tovább adni egy másik *interceptornak* a kéréseket.

Ennek az alkalmazásnak az esetében mivel nincs szükség átalakítani a *http* kéréseket és válaszokat ezért csak egy ilyen komponenst tartalmaz, ami feladata az esetleges hibakódok és azok üzeneteinek a megjelenítése a felhasználó felé konzolos és felugró ablakok formájában.

**5. További fejlesztési lehetőségek**

Az alkalmazás fejlesztése és a szakdolgozat írása közben többször is eszembe jutottak olyan ötletek, amikkel javítani, egyszerűsíteni vagy éppenséggel bővíteni lehetne a webes alkalmazást. Elsősorban a back end-ben lehetne javítani a különböző Dto-k és Entity-k tartalmát mivel vannak olyan adatok, amiket át lehetne helyezni egyes metódusok bemeneti változóiból ezzel megkönnyítve ezek, metódusok használatát és tovább összpontosítaná az összetartozó adatokat. Erre egy példa a metódusokból átszervezni a konyhák és városok változóit az adott Dto-ba és új funkciónak lehetne megvalósítani azt, hogy kibővítjük a városok adatait ezzel megegyszerűsítve azok azonosítását és a felhasználók dolgát.

Továbbá még újabb funkciókkal is lehetne bővíteni a szerver által biztosított szolgáltatásokat ezzel lecsökkentve a kliens oldalon lévő adatfeldolgozásokat. Ilyen lehetne például a különböző konyhák lekérdezése vagy új funkcióként a saját jelszó megváltoztatása adminisztrációnak pedig megadni a lehetőséget alapértelmezett jelszavak beállítására.

A kliens oldalon a back end-hez hasonlóan a modelleket és *http* kéréseket átalakítani továbbá az oldalak kinézetét és űrlapokat is felhasználóbarátabbá és szebbre lehetne kialakítani. Továbbá még a *routing*-ot is át lehet alakítani egy egyszerűbben és egységesebben használható formába. Ugyanakkor még a szerver és kliens közötti kommunikációt is biztonságosabbá lehetne tenni *cookie*-k használatával.

Természetesen az előbb említetteken kívül több mindent is lehetne javítani, fejleszteni az alkalmazáson viszont ezek lennének elsődlegesen véleményem szerint a legfontosabbak az alkalmazás jövőbeli fejlesztését tekintve.

**6. Összegzés**

A szakdolgozatom célja egy webes alkalmazás fejlesztése volt, ami képes minden alapfunkcióját a lehető legkevesebb hibával elvégezni így azt akár már piacra is lehet engedni. Véleményem szerint ezeknek a kitűzött céloknak sikerült nagyrészt megfelelni, hiszen az alkalmazás fő funkciója megfelelően működik és lehetőségünk van az adatbázist egyszerűen kezelni a megadott felületeken keresztül, továbbá az esetleges továbbfejlesztését nagyban segítheti a kialakított projekt struktúra, amit a tanulmányim során elsajátítottak alapján alakítottam ki.

Továbbá pozitívum volt, hogy az alkalmazás fejlesztése során több új technológiát és technikákat tudtam elsajátítani miközben a már meglévő tudásomat is bővebben megtudtam érteni. Ugyanakkor fontosnak tartom megjegyezni, hogy a fejlesztés során megtörtént adatbázis váltásból fakadó problémák és az ilyen folyamatokat támogató technikák megismerése és használata nagyban segített megérteni és perspektívába helyezni az oktatóimtól eddig tanultakat és kapott tanácsokat.

Összeségében el lehet mondani, hogy a kitűzött céljaimat sikerült valamilyen formában elérni, ami közben sikerült kisebb rálátást nyernem egy nagyobb alkalmazásnak a fejlesztési folyamatára. Lehetőséget adott arra is, hogy végleg tudatosítsa bennem, hogy ha szeretnék a jövőben is hasonló területen tevékenykedni, akkor szükségem lesz a folyamatos fejlődésre és tanulásra.

**7.Summary**

The goal of my thesis was the development of a web application that was capable of performing its basic functions with the least amount of problems, giving me the opportunity to release it to the market. In my opinion, the set goals have been largely met because the application’s main function perfectly works and we have multiple interfaces through which we can interact with the database. Moreover, the project structure, which I created based on what I learned throughout my studies, can help with potential future developments of the application.

Moreover, it was a positive that, during the development of the application, I had the opportunity to master new technologies and techniques while deepening my existing knowledge. At the same time, I find it important to note that during development there was a database change, which highlighted for me the importance of knowing problem-solving and change-proofing techniques. By using these techniques, I was able to understand and put into perspective the knowledge and advice that I got from my teachers.

Overall, it can be said that I achieved all my goals in some way, which gave me a better perspective on a bigger application’s development process. It gave me the possibility to become aware for good that if I want to operate in a similar field in the future, I will have to continuously learn and develop myself.

**8. Irodalomjegyzék**

[1] InteliJ IDEA és WebStorm

<https://www.jetbrains.com/>

[2] MongoDB

<https://www.techtarget.com/searchdatamanagement/definition/MongoDB>

[3] Spring Boot

[https://www.tutorialspoint.com/spring\_boot/spring\_boot\_introduction.htm#](https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm)

[4] Angular

<https://en.wikipedia.org/wiki/Angular_(web_framework)>

<https://angular.io/guide/architecture>

[5] TypeScript

<https://en.wikipedia.org/wiki/TypeScript>

[6] Bootstrap

<https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)>

[7] Angular Service-k

<https://angular.io/guide/architecture-services>