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***

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***

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***

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***

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***

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***

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***

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 eldönteni mit, egyél. Amelyik alkalmazás pedig segít ételt választani azok csak egy, a felhasználó által megadott listából választanak ki egyet, ami nem oldja meg azt a problémát, ha teljesen 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. Amelyik alkalmazások hasonlóak még azok az ételrendelő alkalmazások, mint például a *Wolt* és a *Foodpanda* viszont ezek az alkalmazások csak az éttermekre szűrnek és azok a szűrési feltételek is eléggé tágak ezzel megnehezítve a dolgunkat. Ugyanakkor ezeknél az alkalmazásoknál néha eléggé pontatlanul vannak megadva, néha teljesen el is hagyva 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 említetteket hibákon szeretne javítani és a ki nem használt lehetőségeket kihasználni ezzel egy a piacon lévő lyukat betömni. Ugyanakkor az alkalmazásom nem próbálná elvenni a már piacon lévők létjogosultságát, hanem azokat kiegészíteni.

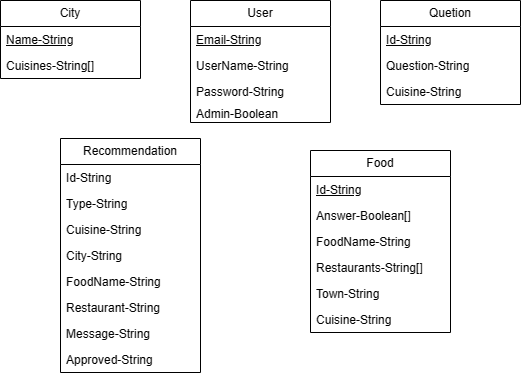
**4. Fejlesztői leírás**

**4.1. Backend**

**4.1.1. Adatbázis**

Az adatbázisom 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. Ezek a kritériumok miatt és az alkalmazás mivoltijából adódóan sikerült egy olyan adatbázist létrehoznom amiben nincsennek relációk, bár van olyan helyzet az alkalmazás futása során, amikor egy-egy adat függhet a másiktól ezek annyira ritkák és specifikusak, hogy egyszerűbb ezeket esetspecifikusan 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 ábrán (1. ábra) 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*. Bár meglehetett volna oldani az adatbázist relációskén a sok tömb miatt az egyes táblákban sok kapcsolótábla jönne létre, ami nagyban lassította volna az adatbázis működését hisz 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 megprogramoznom 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, használni a megfelelő kiterjesztés 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 béli 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 és 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 problémára, specifikusra, amire itt éppen szükség volt.

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

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 elmondottak 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 függenek 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 azok legyenek. 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 annotációja. A Dto-ban lévő kritériumokra példakód:

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

Mindezeken kívül még megtalálható itt 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 fel dolgoz.

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 controllel 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:

@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, 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 és mivel nekünk csak egy adatstruktúrára van szükségünk és a kivételek nagy részénét 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**

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 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 megtalálható, amik a komponens fő funkcióit teszik ki. A következő kódrészletben látható majd 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 van 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 adatokat, egy alapértelmezetten hamis admin adattal kiegészítve, felvisszük az adatbázisunkba.

**4.2.4. Átlag felhasználói komponensek**

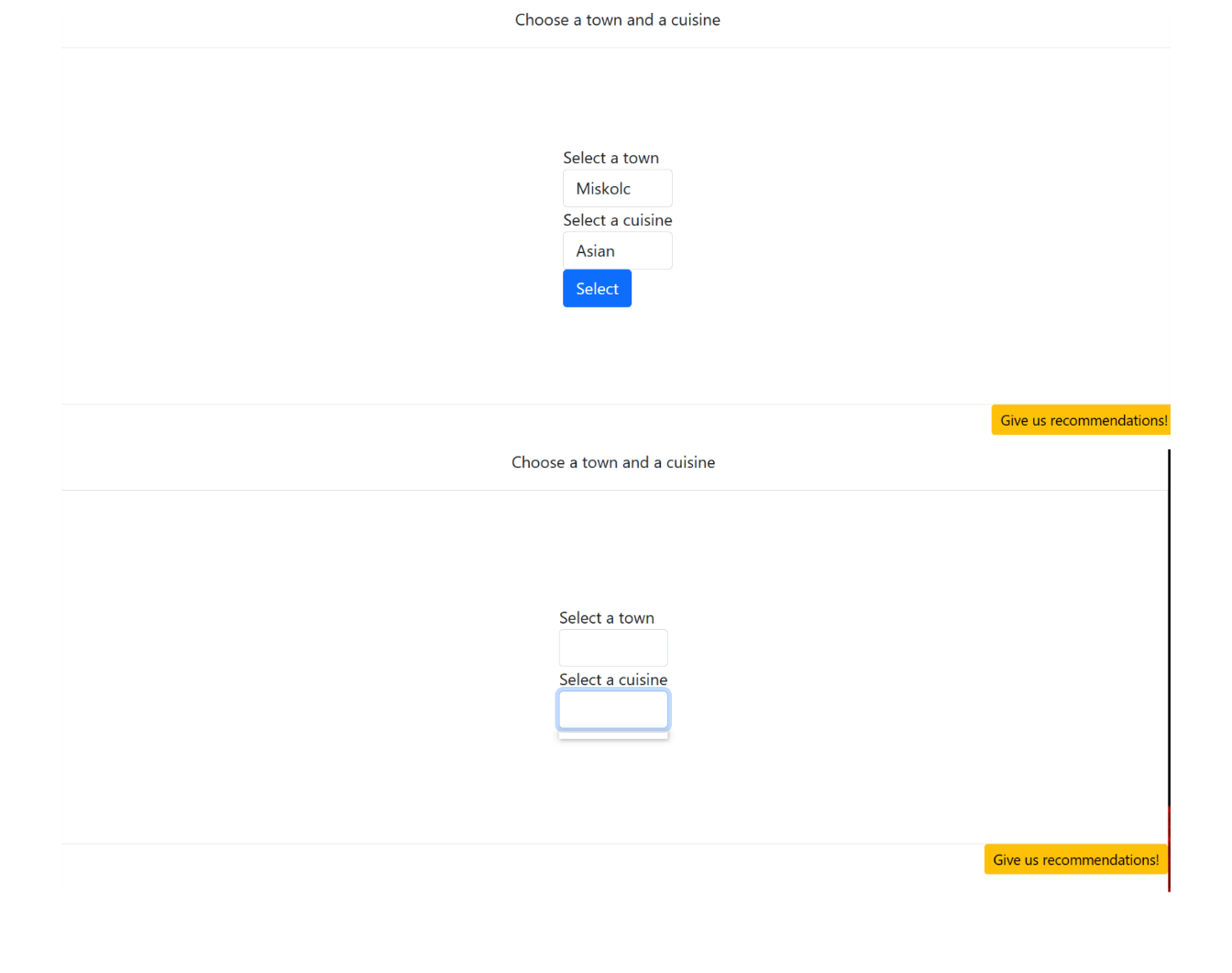
**4.2.4.1. Kezdő oldal**

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 mikor nincs, város kiválasztva akkor nem tudunk konyhát se választani. Ez azét 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.



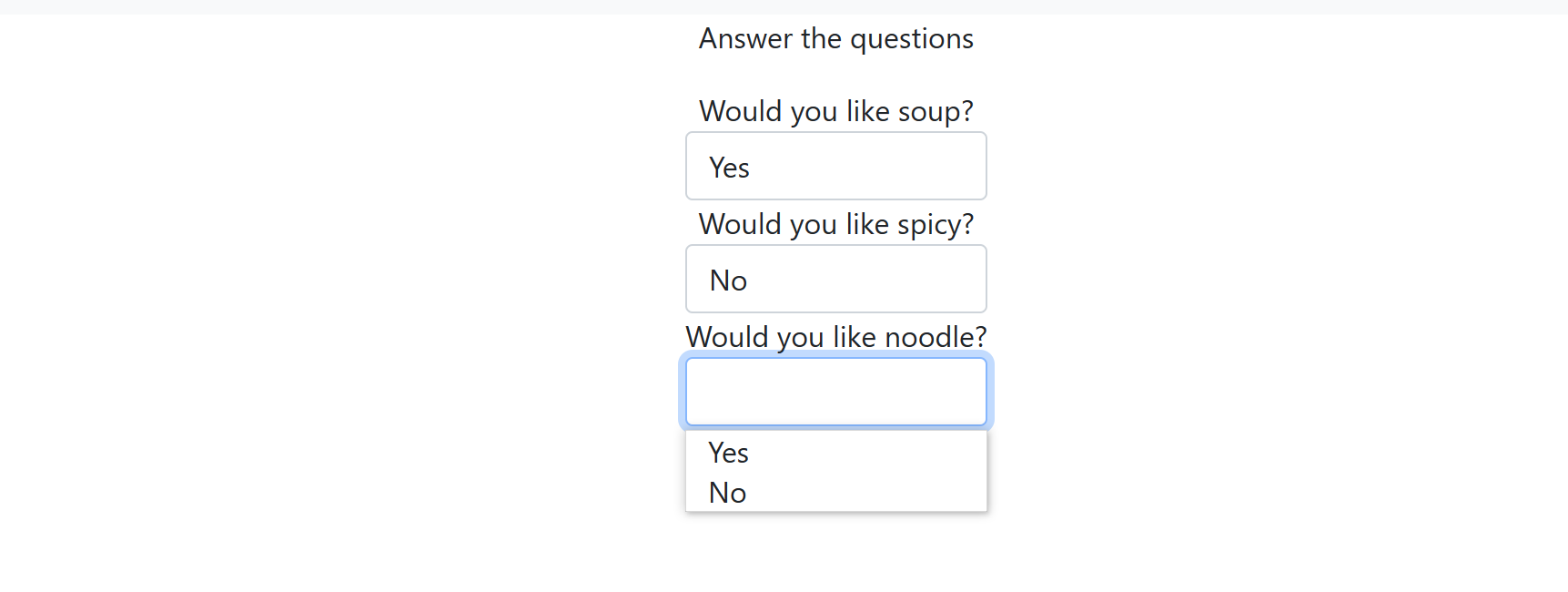
*2. ábra – Legördülő ablakok sajátossága*

**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 elérhető 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 megfelelő sorrendben legyen ezért az adatok kezelésénél nagy szerepe van a tömböknél a megfelelő indexek használatának.

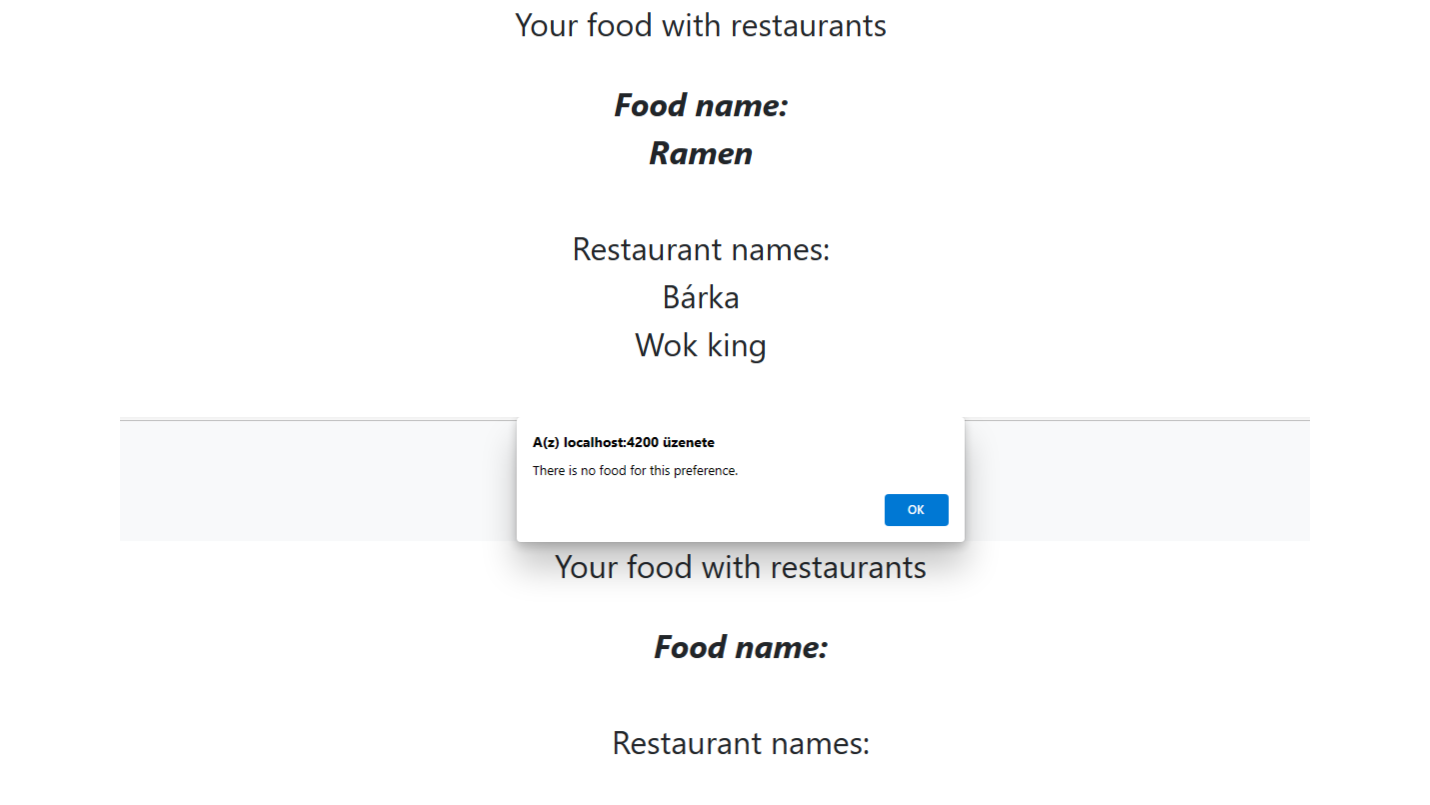


*3. ábra – Kérdőív felépítése, működése*

**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 pedig 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**

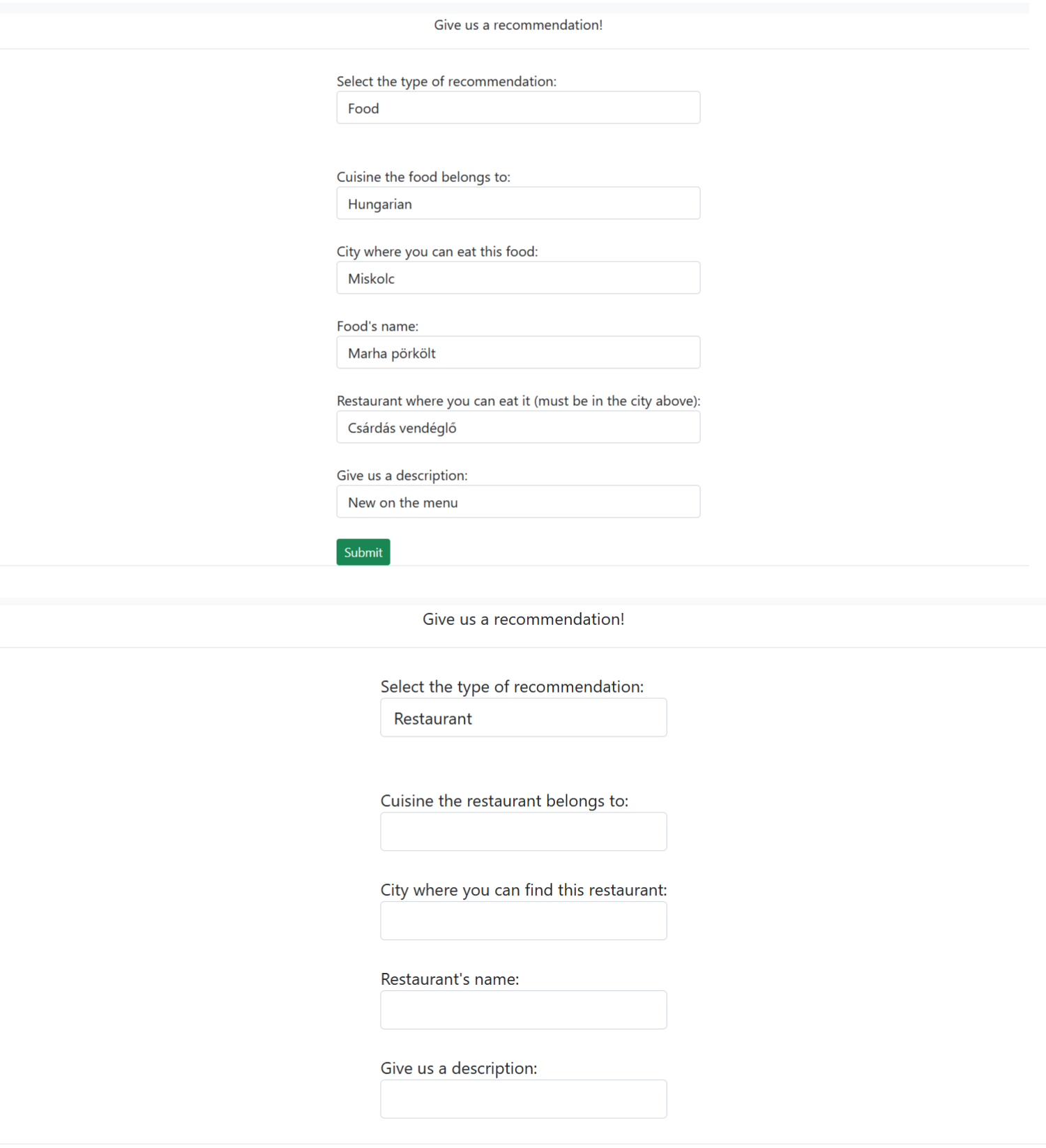
Az ezen az oldalon található űrlap kitöltésével tudunk különböző javaslatokat küldeni az adminisztráció felé. Az űrlap úgy működik, hogy először egy típust kell kiválasztanunk amijen témában szeretnénk küldeni javaslatot majd a témának megfelelő további opciók jelennek, meg amiket kitöltve tudjuk elküldeni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

A téma kiválasztásán kívül a felhasználó szabad kezet kap, abban 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 és mivel egységes modellt használ a komponens a különböző űrlapok elküldésére így több helyen kell elvégezni a kliens oldalon az inputok ellenőrzését. Továbbá mivel 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 *labal*-ek az inputoknál változnak mivel típusonként megfelelő kontextusba kell őket helyezni ezzel szükségessé téve ezt a megoldást.

Maguk a javaslat típusainak a definiálása a kliens oldalon van mivel a szerver nem szolgáltat mást csak a beküldött javaslatok kezelését nem szabályozva annak tényleges tartalmát csak típusát, ami minden adateleménél 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*