ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Alba Regia Műszaki Kar
Természettudományi és Szoftvertechnológiai Intézet

SZAKDOLGOZAT

Alapanyag alapú receptajánló rendszer implementációja

OE-AMK

2024

Hallgató neve: Bory Bence

Hallgató törzskönyvi száma: T001005/FI12904/A-M

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Óbudai Egyetem
Alba Regia Műszaki Kar
Természettudományi és Szoftvertechnológiai Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: Bory Bence
Szakdolgozat száma: SZD2301230926282183[redacted]
Törzskönyvi száma: T001005/FI12904/A-M **Neptun kódja:** [redacted]

Szak: Mérnökinformatikus
Specializáció: Felhő szolgáltatási technológiák és IT biztonság specializáció

A dolgozat címe: Alapanyag alapú receptajánló rendszer implementációja
A dolgozat címe angolul: Implementation of an Ingredient Based Recipe Recommendation System
A feladat részletezése: A hallgató feladata webes alkalmazás készítése receptek ajánlásához. Ennek keretében feladat a különböző receptek, illetve azok összetevőinek összegyűjtése, azokból egy adatbázis készítése, valamint annak kiegészítése webes ajánló rendszerként, amely az ételleírásban található alapanyagok alapján recepteket javasol a felhasználó számára. További feladat a rendszer bővítése speciális kérésekkel, például speciális étrendekre vonatkozó szűrés vagy recept kcal kalkuláció, valamint felhasználóbarát megjelenítés készítése az ajánlásokhoz.

Intézményi konzulens neve: Piglerné Dr. Lakner Rozália

A kiadott téma elévülési határideje:2026. 07. 10.
Beadási határidő: 2024. 05. 15.

A szakdolgozat: Nem titkos.

ÓBUDAI EGYETEM Alba Regia Műszaki Kar Természettudományi és Szoftvertechnológiai Intézet Székesfehérvár

Kiadva: Székesfehérvár, 2024. 05. 09. P.H.

Takács Judit
Intézetigazgató
helyettes

A dolgozatot beadásra alkalmasnak találom: 2024. 05. 15.

.....
belső konzulens

.....
külső konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Alba Regia Műszaki Kar
Geoinformatikai/Mérnöki/
Természettudományi és Szoftvertechnológiai Intézet

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a dolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült dolgozatban található eredményeket az Óbudai Egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja, a titkosításra vonatkozó esetleges megkötések mellett.

Kelt: Gárdony, 2024. 05. 13.

.....

hallgató aláírása

Tartalomjegyzék

1. BEVEZETÉS

Az emberek egyedi lények, akik sokszor nagyban eltérnek egymástól kinézetben, gondolatokban, és mégis vannak dolgok, amikben hasonlítunk, ilyen dolgoknak tekintem például az olyan igazságokat, minthogy az embernek fogyasztania kell a túléléshez. Persze a fogyasztás tág fogalom, fogyasztunk energiát, erőforrásokat, oxigént is, de személyesen a fogyasztásról mindig az étel jut eszembe először, főképp egy japán képregény, a Toriko miatt.

A Toriko röpke nyolc évig futott japánban egy heti képregény antológia, a „Weekly Shonen Jump" lapjai között, de viszonylag nagy követőbázissal bírt, harmincnégy kötete összesen több, mint harmincmillió példányban kelt el a szigetországban. A sorozat egy főképp kalandos, akciódús képregény, ami egy egyedi fantázia világban játszódik, ahol az alapanyagokon és az ínyencségen van a hangsúly.

Amikor 2017-ben elolvastam ezt a művet, kifejezetten megfogott az a tisztelet, amit a szereplők a levadászott alapanyagok és az elfogyasztott ételek felé mutatnak ki, amik nélkül nem csak hogy fizikailag képtelenek lennének élni, de az életük sokkal szürkébb lenne.

Személy szerint mindig is válogatós evő voltam és ez a mű volt az, ami arra inspirált, hogy az elmúlt évek során tágítsam az ízlésemet, megkeressem azon alapanyagokat, amik úgymond harmóniába teszik a lelkemet és a szimpla fogyasztási vágy mellett színesebbé varázsolják az étkezést. Az újonnan kipróbált ételek növelték például a csípősséggel szembeni toleranciámat, emellett olyan egyszerű alapanyagoknak lettem a híve nekik köszönhetően, mint a lilahagyma vagy a lilakáposzta. Alapanyagok, amik, habár másoknak teljesen alapnak számíthatnak, nem szívleltem őket csupán azért, mert nem találtam meg a számomra megfelelő formát az elfogyasztásukra.

Ahhoz, hogy megtaláljam ezeket a megfelelő formákat mindig új helyekre kellett mennem és új ételekkel találkoznom, azért, mert habár szerettem enni és új dolgokat kipróbálni, főzni kedvem sem volt és nem is tudtam. Manapság próbálom fejleszteni

magamat ezen az ágon, de a motivációm gyakran elenyésző, így egy olyan szakdolgozati téma mellett döntöttem, ami a való életben is inspirálhat, segítheti a törekvéseimet, ezért elhatároztam, hogy a feladat során receptekkel és alapanyagokkal fogok foglalkozni, azon belül is receptajánló rendszert fogok készíteni. Ezen belül is olyan rendszert akartam csinálni, aminek legfontosabb funkciója, hogy alapanyagok alapján ad ajánlásokat.

Ennek elkészítéséhez a dolgozat során részletesen fel kell vezetnem a rendszer konkrét feladatait, amik egy adatbázis készítése a recepteknek, illetve egy oldal készítése a receptek és ajánlások megtekintésére. Ezt követően elemeznem kell a megoldás során felmerülő problémákat, aminek a megoldásához szükséges a feladathoz megfelelő eszközök megkeresése, azok kiválasztásának indoklásával együtt. Ezek után egy specifikációban össze kell foglalnom az elkészítés tervezett menetét, amit lekövetve meg tudom oldani a feladatot. A leírtakat végül meg kell valósítanom, elkészítve tehát egy receptekkel foglalkozó oldalt, ahol alapanyagok alapján tudunk dolgozni és különböző szűrésekre ajánlásokat kapunk vissza.

2. A MEGOLDANDÓ PROBLÉMA MEGFOGALMAZÁSA

A szakdolgozati feladatkiírásom a következő: „A hallgató feladata webes alkalmazás készítése receptek ajánlásához. Ennek keretében feladat a különböző receptek, illetve azok összetevőinek összegyűjtése, valamint azokból egy adatbázis készítése, valamint annak kiegészítése webes ajánló rendszerként, amely az ételleírásban található alapanyagok alapján recepteket javasol a felhasználó számára. További feladat a rendszer bővítése speciális kérésekkel, például speciális étrendekre vonatkozó szűrés, vagy recept kcal kalkuláció, valamint felhasználóbarát megjelenítés készítése az ajánlásokhoz.”

A feladat tehát egy olyan weboldal elkészítése, amivel az ember képes recepteket böngészni és képes a saját maga által választott hozzávalói alapján szűrni azokat, azaz a beírt alapanyagok alapján kap szűrt ajánlásokat. Ehhez nélkülözhetetlen valamilyen recept adatbázis. Mivel ilyenhez nem volt hozzáférésem, ezért az internetről kellett adatokat gyűjtenem web scraping technológiával, majd adatbázisba kellett helyeznem azokat. Magán a weboldalon szükséges volt olyan funkciók készítése, mint receptek listázása, azok szűrésére az alapanyagok alapján, illetve speciális szűrők elhelyezése, hiszen manapság egyre növekszenek az igények főképp az ételérzékenységek kiszűrése miatt. Mindez persze mit sem ér, ha magukat a recepteket nem tudjuk megnézni, úgyhogy a recepteknek egy saját oldal is szükséges, amin megtekinthetőek a részleteik. A kinézet természetesen nem lehet primitív, a navigálásnak és az adatok listázásának felhasználóbarátnak kell lennie.

Ennek az elérésére szükséges volt a következők elkészítése:

- Átlátható listázás
- Egyszerűsített felület, ahol minden jelentős funkciót közel találunk egymáshoz
- Lenyitható fülek az éppen szükségtelen elemek rejtésére
- Modernizált felület az összkép miatt
- Konzisztes dizájn

3. A PROBLÉMA ELEMZÉSE

A feladatot részekre kellett bontanom, amiből készíthettem egy áttekinthető vázlatot, mind az eldöntendő kérdések megfogalmazása végett, illetve a feladatvégzések sorrendjének összeállítása miatt. Az elsődleges lépés mindenképp az adatbázis megszerzése és felépítése volt. Az adatbázisban szükségem volt a következőkre: receptek nevei, a receptekhez kapcsolódó képek, a receptek hozzávalói és a szükséges mennyiségek, maga az elkészítés menete. Egyéb információkat a kiválasztott weboldalaktól függően használtam. Ennek az eléréséhez tehát, ahogy arról már az előző fejezetbe szó esett, web scraping használatára volt szükségem. A kód szempontjából a legfontosabb eldöntendő kérdésem pedig maga a nyelv kiválasztása volt, amiben írtam a web scrap-et. A tanulmányaim során például a C# nyelvet használtam, azonban mivel a feladathoz legjobban illeszkedő programnyelv kiválasztását szerettem volna megvalósítani, a feladat lehetőséget kínált olyan nyelvek kipróbálására is, mint a Python. Az adatbázis elkészítése utáni feladat a weblap megvalósítása, ahol egyértelmű választás a HTML és CSS nyelvek használata, emellett elképzelhető, hogy JavaScriptet is érdemes bevetni a folyékonyabb használat és kinézet elérésért. Habár ezek a nyelvek elengedhetetlenek az weboldal elkészítéséhez, magának az oldalnak egy adatbázist is kezelnie kell, amihez választanom kellett egy webes keretrendszert. A hatalmas mennyiségű alkalmazható keretrendszer miatt itt is nehéz volt döntésre jutni.

A szakdolgozatomban felmerülő problémák és a megoldásuk menete tehát a következő:

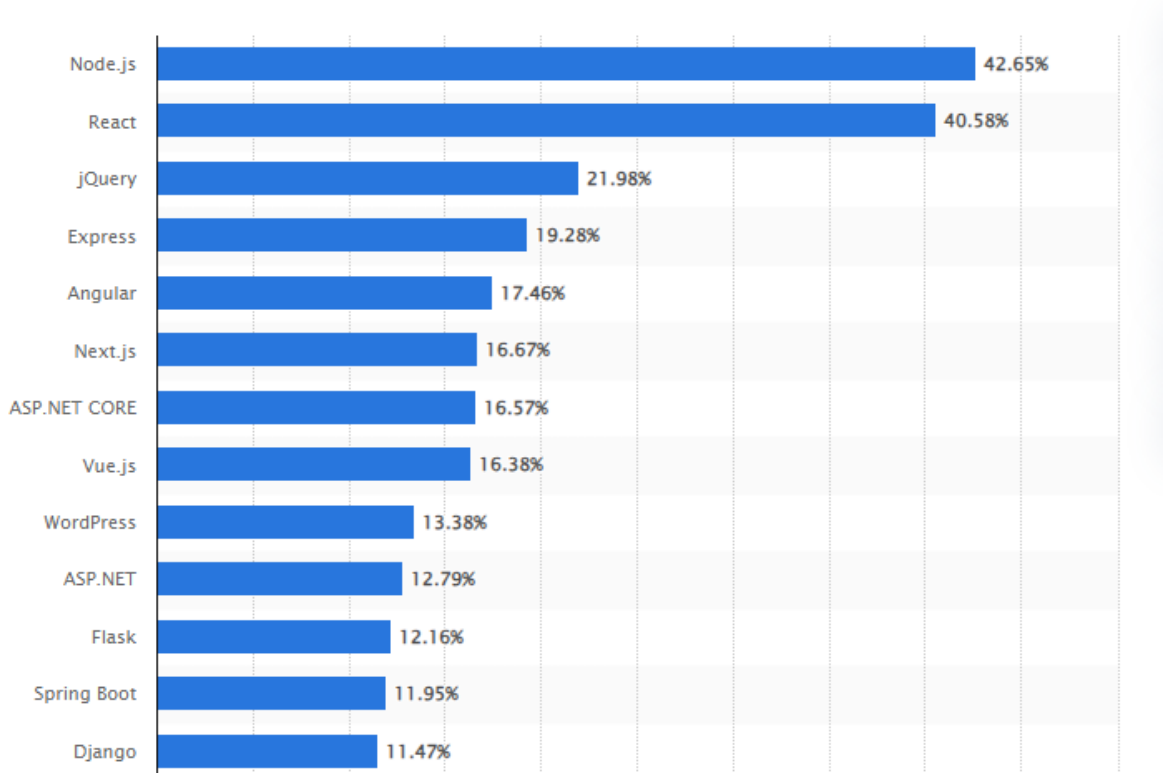
- Recept adatbázis létrehozása
    - Megfelelő forrás keresése az adatbázishoz
    - Az adatok begyűjtése: web scraping
    - Esetleges utómunkák az adatbázison
    - Web scraping, a kódhoz szükséges nyelv kiválasztása és a kód megírása
- Webes keretrendszer kiválasztása

    - Megfelelő keretrendszer megtalálása a könnyű kezelés és az adatbázisom weblapba integrálása végett.
- Weblap
    - Oldal készítése a receptek listázásához
    - Oldal készítése a recept részleteinek megtekintéséhez
- Felhasználóbarátság
    - Szűrők beépítése
    - Felhasználóbarát felület készítése

4. LEHETSÉGES MEGKÖZELÍTÉSI MÓDOK ÁTTEKINTÉSE ÉS MÓDSZEREK KIVÁLASZTÁSA

4.1. Programozási nyelv választása

A weboldal megírása miatt alapvetően több programozási nyelv használatára voltam kényszerítve, ezért hatékonynak látszott minél jobban szűkíteni a kört és olyan fő nyelvet választani, amivel egyszerre megoldható a web scraping, és az adatok kimentése, valamint a keretrendszer a weblap elkészítéséhez.



4.1. ábra A legtöbbet használt keretrendszerek listája, 2023 június
Forrás: statista.com, Lionel Sujay Vailshery, 2024 Jan. 19. [1]

A legelterjedtebb keretrendszereket a 4.1. ábra mutatja, így természetesen ezek egyikét érdemes választani a várhatóan nagyobb mennyiségű fellelhető segítség és dokumentáció végett. A felhozatal azonban még így is bő, tehát érdemes csökkenteni azt.

Logikusan, ha ezt a listát még inkább szűkíteni szeretnénk, valószínű, hogy azt a nyelvet érdemes választani, amihez a legtöbb dokumentációt találjuk meg a lehető legkönnyebben, vagy pedig amiben már van ismeretségünk. Ebből adódóan a számomra legjobb választásnak a következő hármas tűnt: ASP.NET, Flask és Django.

ASP.NET CORE

Egy C# keretrendszer, amivel már volt tapasztalatom, hiszen egy egyetemi félév során használtam. Alkalmas statikus weboldalak fejlesztésére, és habár régen használtam, a közössége mindenképp aktív, és erős dokumentáció található hozzá a Microsoft saját oldalán.

A scrapinget tekintve sok segítőkész alapanyag elérhető, tehát a C# megfelelhet a feladatnak.

DJANGO ÉS FLASK

Mindkettő a hírneve miatt tűnt fel, a hírnevüket pedig egy dolognak köszönhetik, hogy Python alapú keretrendszerek. Saját tapasztalatom alapján manapság a Python mindenképp az egyik legfelkapottabb programozási nyelv. Kezdőbarát nyelvként hirdetik az átláthatósága, könnyen tanulhatósága végett, így az ember nap mint nap újabb Python kurzusokba fut. Afféle hívószóvá avanzsálódott.

Habár korábban még nem dolgoztam benne, a Python az említett gyors tanulhatóságáról, illetve a magas produktivitásáról ismert a kód újrahasználhatósága és az összességében többet kevesebbel hozzáállása és a könnyű áttekinthetősége miatt.

A Flask egy „mikro keretrendszer” [2]. Prioritása a gyorsaság és egyszerűség. Ideális lehet kisebb projektek készítéséhez. Könnyű súlyú és rugalmas keretrendszer, ami számos bővítménnyel és könyvtárral terjeszthető.

Ezzel szemben a Django egy teljes keretrendszer beépített ORM-mel (Object-Relational Mapper), és ideális robosztus projektek készítéséhez. [3] Teljes keretrendszerként olyan

funkciókat kínál, mint a beépített hitelesítés, jogosultságkezelések és ami a megvalósítandó feladathoz fontos: beépített adatbázis-kezelés. A keretrendszer érett, így nagyban terjedt dokumentációval bír. [4]

Habár a Flask egyszerűbbnek tűnhet, a Django beépített funkciói és színes háttere többet kínál a megszabott feladat elvégzésére. A Django emellett jól skálázható, így hatékonnyá téve a hosszútávú fejlesztést és a későbbi bővítéseket is.

A kiválasztott keretrendszerek ismertetett előnyei mellé társulnak a személyes érzéseim is: habár az ASP.NET CORE az a keretrendszer, és a C# az nyelv, amivel tapasztalatom van, így meg tudom mondani, hogy annyira nem szerettem meg ezeket. Ezzel szemben mindig is kíváncsi voltam arra, hogy milyen is a Python. A C#-pal való ellenszenv főképp a viszonylag hosszú szintaxisából ered, ami, habár egyeseknek jól elkülöníthetőséget jelent, személyesen az ellenkezőjét tapasztaltam. Jelen esetben tehát rendelkezésre állt egy ígéretes Python keretrendszer, és egy érdeklődő fél a program nyelv iránt. Ez pedig kiegészíthető a következő két pozitívummal: habár a C# is ideálisnak tűnt a web scrapingre, a Python még több forrással bombázza a témába éppen csak a lábujjhegyét érintő illetőket. Webes útmutatók mellett tényleges könyvek, publikációk találhatók a témában. Ezért tehát evidens, hogy az erőforrásokat és kapcsolódó csomagokat tekintve (mint például BeautifulSoup) a Python a legközkedveltebb nyelv a scrapingre. Emellett a Pycharm, az egyik legközkedveltebb Python IDE ingyenes hozzáférést nyújt diákoknak a Pro verziójához, így ezzel sem kerülünk hátrányba a C# opciókhoz képest.

Mindezek ismeretében a választásom a Python nyelvre, valamint a Django keretrendszerre esett.

4.2. Adatbázis forrásának kiválasztása

A 3. fejezetben taglalt témák összefoglalásával szűkíthetjük és pontokba tudjuk foglalni, hogy milyen weblapot keresünk. Az adatbázis szempontjából ezek a szempontok:

- Magyar nyelvű
- Nagy mennyiségű recept
- Bővíthetőség

Mivel a feladat szempontjából fontos a bővíthetőség, így olyan forrást volt célszerű választani, ami szabadságot ad saját megoldás készítésére a további szűrésekhez, azaz nincs teljesen lecsiszolva. Ez adódhat a választott weboldal funkcionalitás béli hiányosságaiból vagy az adatbázis enyhe rendezetlenségéből. Ezesetben az adatbázist később tisztogatnunk kell majd, de más opció híján ez egy bőven vállalható feladat.

Adatokat természetesen többféleképpen is kinyerhetünk attól függően, hogy milyen opciók állnak a rendelkezésünkre. Szükséges lesz tehát áttekintenünk az adatgyűjtési lehetőségeinket, annak érdekében, hogy eldöntsük milyen forrásokat keressünk. Ezek a lehetőségek az API hozzáférés, a web scraping és a kész adatbázis használata.

API hozzáférés:

- Előnyök:
    - Szabályozott hozzáférés: Az API támogatott hozzáférést nyújt, nem kell HTML parsing.
    - Megbízhatóság: A vizsgált adatok általában jól strukturáltak, dokumentáltak, azaz átlátható és stabil az adatgyűjtés.
    - Hatékonyság: Egyszerre csak a szükséges adatokat szedjük le, amivel csökkentjük a felesleges adatforgalmat és adatfeldolgozást.
- Hátrányok:
    - Korlátozások: Megeshet, hogy az API valamiféle korlátozás mögött van, például kérésszám-limit vagy fizetős hozzáférés.
    - Elérhetőség: Nem minden oldal bír API hozzáféréssel, és mivel alapvetően szűk keretek között keresünk egyet, a sikeres találat esélye igazán csekély.

Web scraping:

- Előnyök:
    - Rugalmasság: Bármilyen oldalról gyűjthetünk adatot, még akkor is, ha nincs API, vagy a keresett információt nem szolgáltatja az elérhető API.
    - Testreszabhatóság: Pontosan azok az adatok kerülnek begyűjtésre, amikre szükség van, és ezek saját definiált struktúrában tárolhatók.

    - Költséghatékonyság: Ingyen megvalósítható, nem szükséges hozzá API kulcs vagy egyéb felár.
- Hátrányok:
    - Komplexitás: A kódot saját kézzel kell elkészíteni, és ügyelni kell arra, hogy esetleges dinamikus tartalmak vagy CAPTCHA mechanizmusok ne akadályozzák az adatok kinyerését.
    - Karbantartás: Ha a script többszöri használatára van szükségünk, akkor folyamatosan frissítenünk kell azt a weboldalban történtő változásoknak megfelelően.

Kész adatbázis használata

- Előnyök:
    - Azonnali használhatóság: Azonnal használható adatbázis, ahol nem kell scrapelnünk.
    - Minőség: Várhatóan egy jól rendezett és ellenőrzött adatbázist kapunk, esetekben még az API-nál is kevesebb utómunkával járhat.
- Hátrányok:
    - Költségek: Ritka a minőségi és publikus adatbázis.
    - Rugalmatlanság: A rendelkezésre álló adatok vagy azok mennyisége korlátozott lehet. Az oldalakkal szemben nincs létező vizuális reprezentáció, ahol akcióban láthatjuk az adatokat.

A kész adatbázis használata és az API lényegileg ugyanazokat az előnyöket biztosítják számunkra, hasonló hátrányokkal, viszont, ha a kettő közül kell választani, személyes választásom az API-ra esne, hisz az, hogy az oldalon már demonstrálva van az adatbázis megbízható működése sokkal bíztatóbb, és akár ötleteket is adhat a projekthez.

A támogatott API hozzáférés azonban túlságosan is ritka, így a web scraping lett az adatgyűjtéshez választott módszer.

A leszűrt információink alapján ennek használatához meg kell keresni a forrásként használt weboldalt, aminek megfelelő HTML-lel kell bírnia.

A web scraping szempontjából ez a következőket jelenti:

- Átlátható felépítés
- Navigálhatóság
- Kevés vagy szűrhető szemét

Ezért tehát olyan forrást kell választani, ami felépítésben jól tagolt, hogy könnyebb dolgunk legyen az átfutandó HTML elemek megadásakor.

A navigálhatóság azért fontos, mert nem csak egy statikus oldalt fogunk megvizsgálni, listák között kell mozognunk, bejárnunk minden recept saját oldalát különböző számozott oldal és kategóriák között.

Személyes döntésem végül a topreceptek.hu oldalra esett a következők miatt. Az oldal dizájn egyszerű és barátságos, a tartalom pedig egyértelműen a receptekre fókusznál. Más oldalakkal szemben itt alig vannak egyéb rovatok, cikkek, amik megnehezítenék a navigációt. Az oldalon lévő receptek kategóriákba vannak szedve, a kategóriák pedig mind megtekinthetők saját URL-jeiken, ami megkönnyíti a böngészést az adatgyűjtéshez.

Magát a recepteket tekintve csak az alapok vannak meg. Képpel ellátott oldal, részekre szedett alapanyagok és mennyiségek, számozott lépések. Ez hagy helyet a kibővítéshez, de elég tartalommal lát el egy lényeges adatbázis létrehozásához. A HTML struktúra könnyen átlátható, a különböző tartalmi elemek, mint a receptek nevei vagy a hozzávalók pontosan elérhetőek, illetve a DOM fájuk nem komplikált, hatékonyan el lehet igazodni rajta.

Összességében az oldal:

- Jól struktúrált HTML-lel,
- részletesen kategorizált tartalommal,
- bővíthető elemekkel

rendelkezik, amik hatékony és a feladatnak megfelelő alapot biztosítanak.

A választott oldal tehát a topreceptek.hu, a szükséges scripteket Pythonban írom, a weblapomat pedig a Django keretrendszer hasznosításával készítem el.

5. RÉSZLETES SPECIFIKÁCIÓ

5.1. Forrásból gyűjtendő adatok

Ahogy az már az előző fejezetben említésre került, a forrás jól struktúrált HTML-lel és kategorizált tartalommal bír. Most vizsgáljuk meg ezt részletesebben, hogy mit kell kinyernünk a forrásból és hogy tehetjük meg ezt.

Először is a forrásunk elérését kell biztosítani. Ebben segítséget nyújt a Python egyik alapkönyvtára, az *urllib* [5]. Ez a könyvtár az, ami lehetővé teszi a webes adatok lekérését, szerepe tehát létfontosságú. Maga a modul több almodulból tevődik össze, amik saját funkciókkal bírnak, amelyeket használni tudunk. Ezek az almodulok a következők:

- *urllib.request*

  URL-ek lekérésére és a válaszok feldolgozására szolgáló almodul.

- *urllib.parse*

  URL-ek elemzésére és módosítására szolgáló almodul. Képes darabjaira szedni és elemezni URL-eket, illetve a darabokból összeállítani őket. Hasznos funkciója, hogy képes egy alap URL definiálására, valamint relatív URL-ek átalakítására abszolút URL-é az alap URL alapján.

- *urllib.error*

  A kivételkezeléshez szükséges osztályokat definiáló modul.

- *urllib.robotparser*

  Robot.txt-ket vizsgál meg ezzel felderítve, hogy végrehajthatóak-e a kívánt URL lekérések.

A weblapot tehát el tudjuk érni, a következőkben megnézzük, hogyan épül fel az oldal, és milyen URL-eket kell behívnunk, megvizsgálnunk.

Ahogy már említve lett, az oldal kategóriákra van tagolva, a kategóriák pedig mind saját URL-lel bírnak, tehát be kell járni az összes kategória URL-jét. Ehhez megoldást nyújt az említett *urllib.parse,* azon belül is az *urljoin.* Az *urljoin* képes teljes URL-ek képzésére

azáltal, hogy kombinál egy alap URL-t, amely ez esetben a kategóriákat listázó cím, és egy másik URL-t, ezek maguk a kategóriák címei.

Amikor az oldalon vagyunk a következő teendőket kell elvégezni:

- A receptek elérési útjának (HTML elemzése) meghatározása
- Belépés a receptek tényleges oldalára
- A recept részleteinek felfedezése és kimentése, ehhez külön függvény létrehozása
- A művelet ismétlése az oldalon felsorolt összes recept esetén
- A kategóriák oldalara vannak bontva, ezért az oldalak közötti váltás szükséges
- Oldalváltás után ismételt receptadatok kinyerése

Tekintve, hogy HTML parsing-gal kell majd foglalkozni, ehhez használható a közkedvelt BeautifulSoup Python könyvtár.

Maga a BeautifulSoup egy nyílt forráskódú, HTML és XML dokumentumok elemzésére és feldolgozására használt könyvtár, amit előszeretettel használnak web scraping feladatoknál. Előnye, hogy:

- Megkönnyíti a navigációt azzal, hogy BeautifulSoup objektummá alakítja a bekért dokumentumokat és újabb, átláthatóbb fa struktúrába helyezi őket.
- Sokoldalú, képes kezelni akár az esetlegesen rosszul formázott dokumentumokat is

Az oldalakra tehát eljuthatunk az *urllib.parse* segítségével, azon belül is használhatjuk az *urljoin-t* hogy minden kategóriát bejárjunk. Amikor már az oldalon vagyunk, a BeautifulSoup-nak köszönhetően kapunk egy jól bejárható és rendezett HMTL dokumentumot.

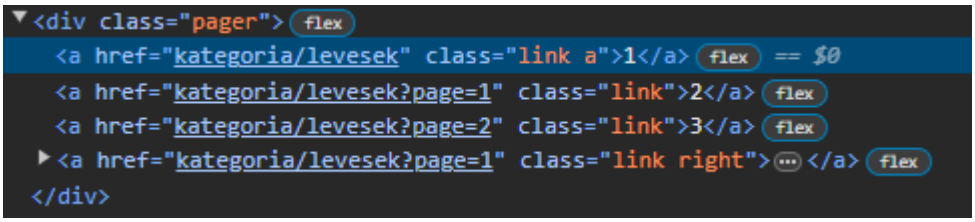
Ehhez meg kell határozni, hogy milyen HTML elemeken kell végigmenni:

- A receptek eléréséhez a hiperhivatkozás elemeket kell megvizsgálni. Ezekből egy oldalon több tucat is elhelyezkedik, de szerencsére, habár van több, tartalmilag közel megkülönböztethetetlen ajánlásunk a felsorolt receptjeinken kívül, feltételezhetően a formázás végett, a szükséges receptek egyedi class elemekkel bírnak (5.1. ábra).

5.1. ábra Receptek linkjei és azonosítója a HTML-ben [topreceptek.hu]

Ezen azonosító segítségével szűrhetőek a megfelelő elemeket, amelyek hiperhivatkozásait el kell tárolnunk későbbi megnyitás és HTML parsing miatt. Maguk a hivatkozások nem teljes URL-ek de ez nem probléma, mert az *urllib.parse*-szal abszolút URL-t tudunk majd készíteni belőlük.

- Az oldalak közötti lapozáshoz meg kell vizsgálnunk a léptetőt. Mivel a weblap jól tagolt, ez könnyen felfedezhető, egy egyedi div elemen belül megtalálhatjuk. Az oldalváltást URL változások jelölik, amiket az *urljoin*-nal le tudunk kezelni.



5.2. ábra Lapozó felépítése [topreceptek.hu]

A felépítésnek köszönhetően (5.2. ábra) le tudjuk olvasni, hogy a kategóriánkon belül összesen hány oldalunk van, amivel már működésre tudunk bírni egy ciklust. Ügyelnünk kell viszont arra, hogy a „page=1” a második oldal lesz, tehát az alapoldalon kell majd kezdeni, amit az előző pontban összeállítunk, aztán a „page=utolsó oldal mínusz egy” URL-ig folytatni a ciklust.

- Ha már bejártuk a forrás összes kategóriáját, azok összes oldalát és a receptekre ugrunk, rátérünk a scraping legfontosabb és legnagyobb feladatára, a tényleges recept adatok eltárolására, és azok adatbázisba való beillesztésére.

Számunkra a következő adatok szükségesek:

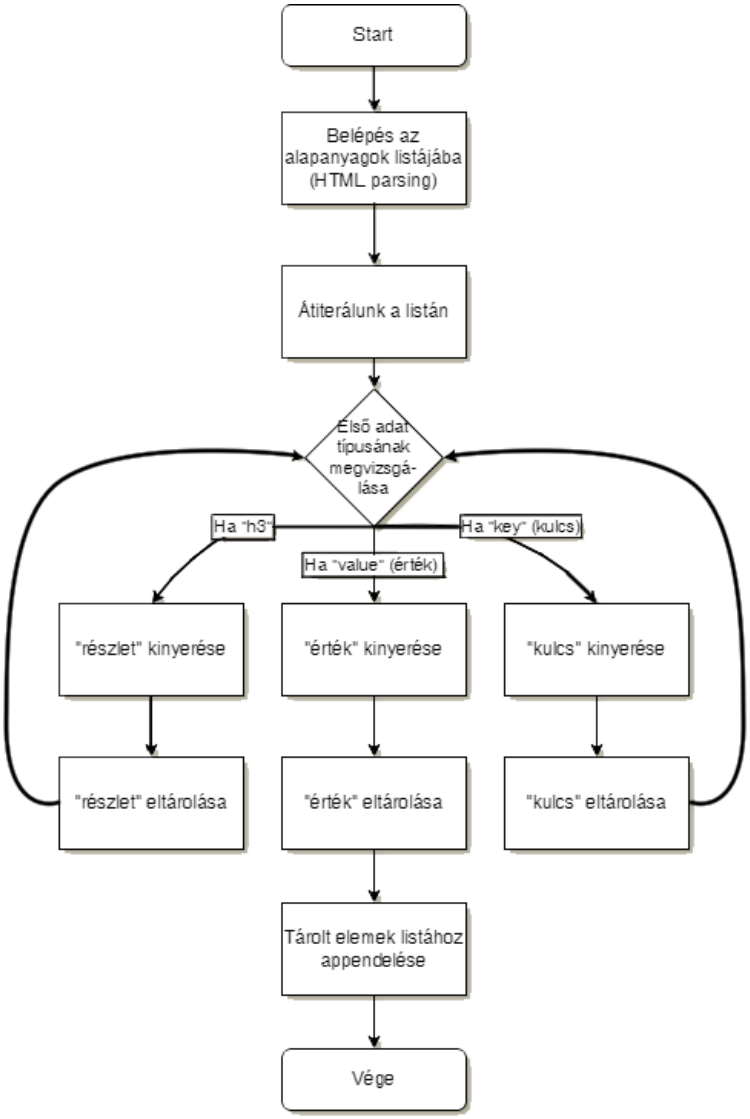
- Recept neve
- Recept képe
- Kategória tagek szűréshez
- Elkészítési idő
- Elkészítés lépései
- Alapanyagok [Alapanyag, Mennyiség, Recept része]

Ezek mind definiálható HTML elemekbe vannak szedve, amiken HTML parsing-gal végig lehet menni, használva a BeautifulSoup által nyújtott függvényeket, mint a *find* és a *find_all* [6]. Az eltárolandó adataink nagyrészt egyszerű stringek lesznek, leszámítva a képeket, amik hiperhivatkozások.

Problémába az alapanyagok lekérésénél ütközhetünk, specifikusan a részekre tagolásnál.

5.3. ábra Alapanyagok felépítése és részletezés [topreceptek.hu]

Az alapanyagok és mennyiségek szótárban tároltak (5.3. ábra). Maguk a szótárak olyan adatszerkezetek, amik az ábrán is látható kulcs-érték párokat alkotnak és tökéletesek az efféle adattárolásra és lehívásra a gyors hozzáférésük és dinamikusságuk miatt. Az adattároláskor ez számunkra is nagyon jó szerkezetet biztosít, ezért a továbbiakban szótárt használunk. Maguk az adatok könnyen kinyerhetőek HTML parsing segítségével, a probléma csak ott jelentkezik, amikor megpróbáljuk begyűjteni és helyesen párosítani a hozzájuk tartozó h3-as elemeket is, amik jelzik, hogy a recept melyik részéhez kellenek.



5.4. ábra Példa folyamatábra az alapanyagok helyes kinyeréséhez

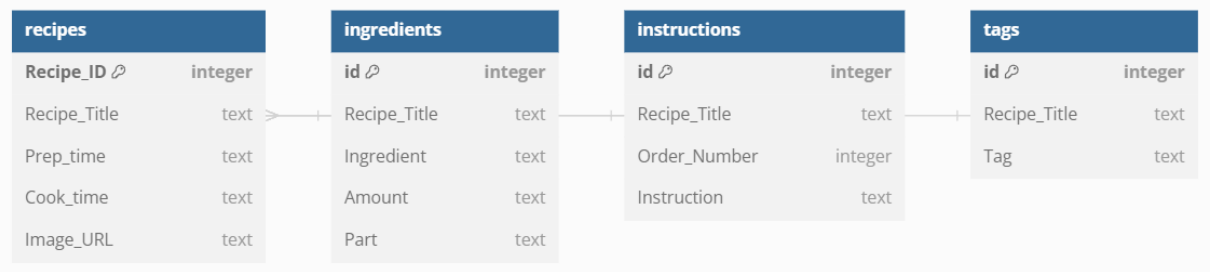
Ennek a lekezelésére ahhoz hasonló felépítésű programot kellett írni, mint amit az 5.4. ábra mutat, ahol induláskor a legelső elemet vizsgáljuk, az első kulcs-érték elem mentését követően pedig a soron következő elemet kell megvizsgálnunk. Ilyenkor elágazásokkal kell dolgoznunk, amik megnézik, hogy van-e társított h3 elem, azaz „részlet”, és ha igen, azt lementik egy változóba, ami addig tárolja az első h3 elem értékét, amíg ilyen elemmel újra nem találkozik. Ilyenkor tehát a tényleges tárolt adataink, amiket később adatbázisba kell mentenünk a következőképpen néznek ki: [részlet1], [mennyiség1], [alapanyag1]; [részlet1], [mennyiség2], [alapanyag2].

5.2. Létrehozandó adatbázis

A web scraping során összeszedett adatainkat végül adatbázissá kell alakítani. Ez a web scraping script végén végezhető el a különböző append-elt elemek általunk generált adatbázisba való beillesztésével. A Python erre segítséget is nyújt az *SQLite3* támogatásával, aminek köszönhetően könnyen léphetünk interakcióba különböző adatbázisokkal.

Az *urllib*-hez hasonlóan az *SQLite3* is beépített modul, ami valójában a C-s *SQLite* könyvtár [7] implementációja. Ez előnyös a kis mérete és önállósága miatt, hisz a rendszer önálló, szerver-mentesen megtekinthető, helyi lemez alapú adatbázist biztosít.

A tárolásra kerülő, korábban felsorolt elemekből táblákat kell alkotni a szegmentálás végett. Az adattáblákat az 5.5. ábra mutatja be. Az adatmodell kialakításában elengedhetetlen az átláthatóság.



5.5. ábra Adatbázis felépítése

Normalizálás szempontjából a különböző adattáblák mind rendelkeznek egyedi azonosítóval, a kapcsolódó adatok saját táblákkal bírnak, illetve egy táblán belül nincsenek ismétlődő csoportok, tehát az első normálformát elértük. Második normálformának már sajnos nem felel meg, mert Recipe_Title felborítja a funkcionális függőséget minden olyan táblában, ami nem a recepteké.

Ez megoldható lenne például a Recipe_ID idegen kulcsként való használatával, de az adatbázis kezelésekor való könnyített átláthatóság miatt egyelőre figyelmen kívül hagyjuk ezeket az optimalizációkat.

5.3. Django keretrendszer felhasználása

Ahogy korábban ismertettem, a Django egy nyílt forráskódú Python keretrendszer, ami sajátosságainak, illetve a hosszú élettartama alatt gyűjtött nagy méretű közösségének köszönhetően jól használható és előnyös rendszerként szolgál a back- és front-end webfejlesztéshez.

A keretrendszer használatával tehát gyors és hatékony weboldal összeállítás lehetséges.

5.3.1. Django felépítése

A Django keretrendszer felépítését tekintve MVT (Model-View-Template) architektúrát követ, szemben mondjuk az APS.NET MVC (Model-View-Control) architektúrájával. A két rendszer eléggé hasonlít egymásra, mindkettő külön komponensekre bontja a kódot a könnyebb karbantartás, tesztelhetőség és összességében átláthatóbb szerkezet létrehozásának céljával.

Komponenseik hasonlítanak, de nem egyeznek: Model-View-Contoller kontra Model-View-Template. A Model komponens az egyetlen, ami lényegileg ténylegesen közel egyező, az MVT view komponense jobban hasonlít az MVC controllerjére, amíg a template a view-hoz.

A különböző komponensek feladata, működése és benne a feladatunkhoz szükséges teendők a következők:

Modell (ORM)

A modell [8] az a komponens, ami az adatokért, azok kapcsolataiért felel. Konkrétabban, a modellek osztályok a Python kódon belül, amik a használt adatbázis tábláit reprezentálják. Ezeket a models.py scriptben tudjuk létrehozni és módosítani.

Működésükért ORM, azaz objekt-relációs leképezés, felel, aminek a fő funkciói az említett modellek, illetve a QuerySet-ek. A QuerySet-eket a View-ban hozzuk majd létre, az adatbázis tartalmának lehívásához szükségesek [9]. Összességében az ORM erős és kényelmes kezelő eszköz az adatbázis adataihoz való hozzáféréshez az SQL nyelv használata nélkül.

A feladatunk itt tehát az lesz, hogy létrehozzunk az adatbázisunknak megfelelő Python osztályunkat a „models.py” -ban, majd feltöltsük ezeket az adatbázisunk adataival.

View és Url

A view [10] különböző HTML sablonokból épül fel, amik a létrehozott modelljeink adataiból dolgoznak. Ehhez a views.py scriptben tudunk különböző függvényeket írni, amik úgymond kérelemkezeléssel foglalkoznak, válaszokat adnak HTTP kérelmekre.

Ilyen függvényeket kell tehát írni, a kívánt adataink megjelenítésére. Az első oldalunkhoz olyan függvényre lesz szükség, ami először felhasználja a receptek és hozzávalók táblázatát. A recepteket meg kell jeleníteni, a hozzávalókra pedig azért lesz szükség, hogy ezek alapján lehessen szűrni. Az elkészítendő függvény tehát a receptek megjelenítéséért és azok szűréséért felelős, így bármilyen egyéb későbbi szűrésre felhasznált táblát meg kell majd hívni, és lekezelni.

A felsorolt recepteknek hiperhivatkozásoknak kell lenniük, amikre kattintva megtekinthetők a recept részletei. Ehhez el kell készíteni a második legfontosabb view függvényt, a részletmegjelenítést. Ehhez először le kell kezelni azt, hogy lássuk a receptet. Ehhez azonosítónak használható a receptek ID-je, amikkel új, és rövid URL-t lehet képezni, habár az ID-t össze kell majd kötni a recept címekkel, és a többi tábla anyagát az alapján lehívni.

Ezután ténylegesen és helyesen le kell hívni az adatokat: alapanyagok a megfelelő részhez legyenek sorolva, a recept elkészítésének sorrendje legyen helyes. Az oldalon lévő teendők többi része már a dizájnban fog megvalósulni.

Ezeknél a lépéseknél fontos az URL-ek készítése, hiszen az URL beírása adja meg azt a HTTP kérést, amire maguk a view függvények adnak választ. Szükség lesz tehát egy URL-re a main oldalnak, illetve egy dinamikus URL-re, ami módosítja a main URL-t és minden receptre való kattintáskor behelyezi a megfelelő ID-t a URL-be, ezzel lekérve a megfelelő recept adatait.

URL-ben emellett le kell majd kezelni a main oldal szűréseit és az esetleges lapozást.

Template-k és statikus elemek

A Django MVT architektúrájában a Template az, ami máshol a view-nak felelne meg, azaz ez az a rész, amivel HTML tartalmat lehet visszajuttatni a kliens felé. A template-kben tehát HTML oldalakat kell létrehozni, amik a behívják és megjelenítik a fent említett view függvények értékeit. Erre különös segítséget nyújt a template nyelv [11], ami megengedi a view-ból hívott adatok listázását és meghívását. A *{{változó}}* szintaxissal meg tudjuk adni a szükséges változókat, amik egyszerűen helyezhetők ciklusokba, feltételes utasításokba.

A HTML oldalakhoz természetesen CSS, esetlegesen JavaScript is társul. A Django-ban ezeket statikus fájlokként lehet meghívni. A statikus fájlok alatt olyan, nem dinamikusan változó fájlokat értünk, amikehez static_root-ot és static_url-t kell beállítani. A static_root egy létrehozandó mappa lesz, amiben a statikus fájlokat lehet tárolni, az url pedig természetesen az az útvonal, amivel template-ben megadható az elérés. Ezeket a központi beállításokban (settings.py) kell konfigurálni. Ilyen statikus elemek lehetnének még például a képek is.

A megvalósítandó rendszerben mindenképp szükséges statikus elemek két CSS fájl, amiket a main és recept részlet oldalakhoz fogunk csatolni.

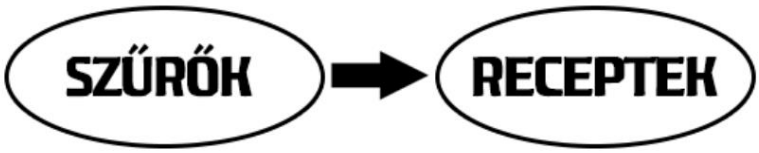
5.3.2. Weblap dizájn

A dizájn minden weboldalnak egy elengedhetetlen eleme. Ez szabja meg azt, hogy hogyan épülnek fel a statikus elemek, illetve ez a felhasználói élmény és elégedettség egyik, ha nem legmeghatározóbb eleme. Legfontosabb tényezői a következők:

- Egyediség
- Vizuális hierarchia
- Átlátható, letisztult elrendezés

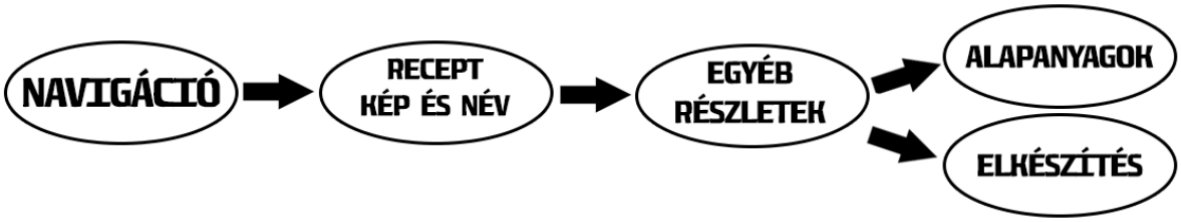
Az egyéniség és karakter fontos, főleg, amikor olyan oldalt készítünk, aminek nagy a vetélytárs bázisa. Célunk vele a saját követőbázis kiépítése, érzelmi hatás keltése a felhasználókban, ami kötődést teremt köztük és az oldal között ezzel növelve az elégedettségüket és főképp a hűségüket.

Vizuális hierarchia – az elemek hierarchiájának úgy kell felépülnie, hogy minden a felhasználó számára hasznos felület könnyedén elérhető legyen, azaz az elemek fontoság szerint legyenek rendezve. Egy olyan projektben, mint a szakdolgozati feladatom, a szűrők könnyű láthatósága elengedhetetlen, így hát fontossági sorrendben ezek az elsők, s csak ezután jönnek a receptek (lásd 5.6. ábra).



5.6. ábra Főoldal vizuális hierarchiája

Amikor a receptek részleteit vizsgáljuk, a legfontosabbak a navigációs elemek, amivel a felhasználó vissza tud térni a főoldalra, habár nem ennek kell a legfeltűnőbb elemnek lennie. A legfeltűnőbb és második legfontosabb elem a recept címe és neve lesz, hogy tudjuk, helyes oldalon járunk-e. Ezt követik a részletek, mint például az elkészítési idő, majd az alapanyagok és lépések. Az alapanyagok és lépések egyaránt fontosak, így egymás mellett kell elhelyezkedniük, ahogy ezt az 5.7. ábra mutatja.



5.7. ábra Receptrészletek oldalának vizuális hierarchiája

Átlátható rendezés – A vizuális architektúrához tartva magunkat, jól tagolt és légző szerkezet, ami a lényeget helyezi előtérbe. Fontos, hogy a receptek felsorolásánál előnyt élvezzenek a képek, hisz az emberi szemet egyből az vonzza be, az fogja eladni a receptet.

Kimondottan ügyelni kell a szűrőkre, ahogy említettem, hierarchia szempontjából ezek a legfontosabb elemek, viszont nem szabad túlzásba esni velük.

5.8. ábra Weblap dizájn példa

Olyan szűrők is lesznek, amire nincs mindenkinek szüksége, tehát ezeket lenyitható fülek mögé kell rejteni, ha más esetben rontanák az összképet vagy a receptek láthatóságát (lásd 5.8. ábra). Ezen kívül fontos, hogy a hierarchia alapján az alapanyagok és utasítások egymás mellett helyezkedjenek el.

6. MEGVALÓSÍTÁS LEÍRÁSA

6.1. Web scraping és adatbázis

A fejlesztéshez először is egy IDE-re volt szükségem. Korábban említésre került, hogy a Pycharm, egy elismert és közkedvelt IDE professzionális verziójához ingyenes hozzáférést nyújt diákok számára a programot fejlesztő JetBrains, így én ezt használtam.

Ahogy a specifikációban említettem, a kódhoz szükséges több, a Pythonban nem alapértelmezett könyvtár, mint a *requests* és a *BeautifulSoup*. Első lépésként ezen könyvtárakat telepítettem, ami könnyű feladat volt az IDE-nek köszönhetően. A beállításokban található egy fül, ahol rengeteg kiegészítő csomag felsorolása látható, ahonnan egyszerűen kikereshetők és telepíthetők a szükséges elemek, de ezen kívül CMD-vel és a Pycharmba beépített terminál segítségével is könnyen telepíthetőek.

A kód első soraiban ezeket a könyvtárakat behívtam egy *import* paranccsal, hogy használni tudjam azokat. Ezután elkezdtem írni a tényleges scriptet.

```
base_url = "https://www.topreceptek.hu/"

categories = [...]

recipe_data = {}
unique_recipe_titles = set()
```

6.1. ábra Web scraping – Inicializálás

Először inicializáltam néhány szükséges üres elemet, amik megalapozzák a scrapinget. Az első ilyen az alap URL és a kategóriák listája az *urljoin* kihasználására. A forrás alap URL-jét a 6.1. ábra mutatja, míg a kategória oldalakra a következőképp lehet eljutni *„…/kategoria/kategoria_neve”.* A kategória listába tehát begyűjthető az összes szükséges kategória neve, amiből később a listán való iterációval abszolút URL-eket lehet összeállítani a későbbi függvényekben, amik elvégzik rajtuk a specifikációban leírt műveleteket.

A látható még két üres lista (6.1. ábra), amelyek közül egy, a *recipe_data* egy szótár lesz a receptek adatainak tárolására, és azok adatbázisba való betöltéséhez. A recept címekre

pedig külön set-et hoztam létre, ami figyeli, hogy az adott recept egyedi-e. Mivel csak olyan recepteket kell scrapelni, amik még nincsenek meg, így ezzel ki tudtam küszöbölni az esetleges duplikációkat, ha mondjuk ugyan az a recept két külön kategóriában is feltűnne.

```
def __init__():
    for category in categories:
        category_url = urljoin(base_url, url: f"/kategoria/{category}")
        result = requests.get(category_url)
```

6.2. ábra Web scraping - Abszolút URL

Ezt követően létrehoztam az *init* függvényt (6.2. ábra). Ez a függvény kezeli a kategóriák és azok oldalainak bejárását, illetve hívja meg a tényleges recept adatokat gyűjtő függvényt. A függvény egy ciklust tartalmaz, ami behívja az említett kategória listánk első elemét, majd abból és az alap URL-ből a megfelelő szintaxis mellet abszolút URL-t készít, amit aztán a *request* könyvtár használatával lekérésre kerül.

Ezt követően megkezdődik az első HTML parsing (6.3. ábra). A HTML tartalmat jobban átlátható *BeautifulSoup* objektummá alakítottam, illetve a parsing-hoz használtam a *html.parser* modult. [12] Ezt az objektumot egy változóba mentettem, amin aztán a parsingot végeztem.

Elsőnek a lapok mennyiségét vizsgáltam meg. Ehhez megkerestem a specifikációban mutatott *pager* osztályú elemeket (5.2. ábra) az első parsing sorral: *doc.find(class=”pager”).* Ezen belül a hiperhivatkozások mennyiségéből lehet kiolvasni az összesített oldalszámot.

```
doc = BeautifulSoup(result.text, features: "html.parser")
pager = doc.find(class_="pager")
if pager:
    page_links = pager.find_all("a")
    total_pages = 1
    for link in page_links:
        try:
            page_number = int(link.get_text(strip=True))
            total_pages = max(total_pages, page_number)
        except ValueError:
            pass
else:
    total_pages = 1
```

6.3. ábra Web scraping – Oldalszám kinyerése

A továbbiakban létrehoztam egy változót, a *total_pages*-t, ami az összesített oldalszámra utal és alapértéke 1. Ezt követően egy ciklusban iteráltam a hiperhivatkozásokon, és integer-ré alakítottam az „*a*” elemek szövegét, azaz az oldalszámot. Az oldalszámot egy új változóba mentettem, majd megnéztem, hogy ez a változó nagyobb-e az eredetileg megadott 1-es össz-oldalszám alapértéknél, és ha igen, felülírtam vele. Amint elfogytak az áttekinthető *„a”* elemek, a ciklus leállt és megkaptam az összesített oldalszámot.

Ezt követően át kell futni a receptek oldalain, amelyhez a 6.4. ábra kódja használható.

```
for page in range(1, total_pages + 1):
    if page == 1:
        url = category_url
    else:
        url = f"{category_url}?page={page-1}"
    scrape_recipe_data(url)
```

6.4. ábra Web scraping - oldalak átfutása

Ehhez egy ciklus végrehajtásával átmegyünk az oldalakon, ügyelve arra, hogy az első oldal URL-je maga az alap kategória url, így először azt kell átnézni, illetve emiatt az összes többi oldal URL-je jelenlegi szám mínusz egy lesz. Így bejárjuk az összes oldalt, és mindegyiken meghívjuk recept adatok kinyerésére szolgáló függvényt.

Ezután az előző függvényhez hasonló módon bekértem az URL-t, request-eltem a HTML-t, majd *BeautifulSoup* objektummá alakítottam azt. Ezúttal a *„-b”* osztályú elemeket hívtam be egy változóba, amik az egyedi receptoldalak linkjeit tárolják. A változónál *find.all*-t használtam, tehát begyűjtöttem az összes linket az oldalon, majd ciklusban végig mentem azokon.

```
for link in recipe_links:
    recipe_title = link.find("div", class_="title").get_text(strip=True)
    if recipe_title not in unique_recipe_titles:
        unique_recipe_titles.add(recipe_title)
        recipe_url = urljoin(base_url, link["href"])
        recipe_result = requests.get(recipe_url)
        recipe_doc = BeautifulSoup(recipe_result.text, features: "html.parser")
```

6.5. ábra Web scraping - egyedi receptek böngészése

Mivel nem szerettem volna ugyanazt a receptet kétszer scrapelni, ezért először kinyertem a recept címét, amit betöltöttem az előzőleg létrehozott *unique_recipe_titles set*-be. Ha új receptnevet találtam, betettem a *set*-be, majd abszolút URL-t hoztam létre neki, mert az oldalon tárolt hiperhivatkozások URL felépítése egyezik a kategóriákéval, tehát csak az URL alaphoz csatolt részét tartalmazza. Ezzel létrehoztam a recept címét és URL-jét, amelyekkel a *requests* és *BeautifulSoup* használatával elvégeztem a szükséges parsingot. Mindezeket a lépéseket tartalmazó kódrészletet a 6.5. ábra mutatja.

A parsing-nál folyamatosan hasonló logikát használtam: új változót hoztam létre, amiben beletettem *find*-dal a kritériumunknak megfelelő HTML elemeket, majd ismét új változót hoztam létre, amibe még jobban szűrtem és ezúttal csak a szükséges tartalmat tettem.

Így szedtem össze az elkészítési időket, képeket, tag-eket és az elkészítés lépéseit.

```
instruction_divs = recipe_doc.find_all( name: "li", class_="-_C", attrs={"x:fn": "instructionMark"})
instructions = [{"number": div["data-n"], "text": div.find("div", class_="li").text.strip()} for div in
                instruction_divs]
```

6.6. ábra Web scraping - recept lépések kinyerése

Ezek közül egyedi a recept lépések kinyerése (6.6. ábra), mert a lépések mellett azok számát is kinyertem. A *find_all* metódus itt hosszabb a megszokottnál, mert minden azo-

nosító elem az oldal más részén is megtalálható, így egyész hosszú felsorolást kellett tennem a helyes azonosításhoz, majd az új listát ciklusosan feltöltenem az utasításokkal és azok lépésszámával.

Maguknak a hozzávalóknak a kinyeréséhez kicsit eltérő megközelítést kellett alkalmaznom, ahogy az a specifikációban is megfogalmaztam.

```
ingredients = []
ingerdientlist = recipe_doc.find_all( name: "div",  attrs: {"class": "key-value"})
current_part = ""
current_ingredient = ""
current_amount = ""
```

6.7. ábra Web scraping - alapanyagkinyerés 1

A 6.7. ábra mutatja be az átlagos parsing-ot, majd a feltöltendő változók létrehozását. Az adatokat a létrehozott, kezdetben üres listában tároltam úgy, hogy belehívtam a három, kezdetben üres stringet. A három stringbe háromféle adat mehet, h3 elem, szótár kulcs és szótár érték, melyet egy listán történő iterálással lehet kinyeri a 6.8. ábra kódrészletének segítségével.

```
for data in ingerdientlist[0]:
    if (data == ' ' or data == '') is False:
        if data.name == "h3":
            if data.contents.__len__() > 0:
                current_part = data.contents[0]
```

6.8. ábra Web scraping - alapanyagkinyerés 2

Ehhez a *data ciklusváltozót használtam, ahol* a *data* alatt a lista első HTML elemét vizsgáltam. Először megnéztem, hogy az első létező elem h3 elem-e. Amennyiben igen, megnéztem, hogy van-e tartalma, és ha van, akkor pedig eltároltam a kihelyezett stringben. Amennyiben nem h3 elem, egy *else* feltétellel megvizsgáltam, hogy kulcs-e. Ha igen, eltároltam a tartalmát, ha nem, megnéztem, hogy érték-e. Ezzel a ciklussal végig mentem az alapanyagokat, mennyiségeket és recept részeket tartalmazó *div* minden egyes elemén.

```
if data.attrs["class"][0] == "value":
    if data.contents.__len__() > 0:
        current_ingredient = data.contents[0]
    else:
        current_ingredient = ""
    ingredients.append({"Ingredient": current_ingredient,
                        "Amount": current_amount,
                        "Part": current_part})
```

6.9. ábra Web scraping - alapanyag append

Az adatok eltárolásra az előre létrehozott üres *ingredients* listát használtam. Fontos, hogy az *append* metódust, ami ténylegesen beleteszi a szótárba az elemeket, azután hívjuk meg, hogy ellenőriztük a *value* osztályú elemet, mert sorrendben ez az utolsó elem. Tehát az esetleges részlet és kulcs már tárolásra kerültek a stringekben, így szótárként *append*elhetőek. Mindez az előbb megkezdett ciklus (6.8. ábra) része, tehát minden alapanyag, tag, stb. kinyerése recept oldalanként történt meg, és a recept címével együtt mentésre került a script legelején létrehozott üres listába.

```
recipe_data[recipe_title] = {
    "Recipe_Title": recipe_title,
    "Ingredients": ingredients,
    "Instructions": instructions,
    "Prep_Time": prep_time,
    "Cook_Time": cook_time,
    "Image_URL": image_url,
    "Tags": tags
}
```

6.10. ábra Web scraping – adattárolás

Az eltárolt adatokat ezután adatbázisba mentettem. Itt került használatra az *sqlite3* és annak metódusai. Elsőképp egy connect metódussal „összekötöttem” a scriptet egy általam elnevezett adatbázissal, ami ezáltal létrejött, és készen állt a feltöltésre. A 6.11. ábra „*c.execute”* utasítását használva a Python kódon belül SQL utasításokat futtattam le, amikkel megalkottam a szükséges táblákat.

```
c.execute('''
    CREATE TABLE IF NOT EXISTS recipes (
        Recipe_ID INTEGER PRIMARY KEY,
        Recipe_Title TEXT,
        Prep_Time TEXT,
        Cook_Time TEXT,
        Image_URL TEXT
    )
''')
```

6.11. ábra Web scraping - adatbázis táblák alkotása

Ezek a táblák természetesen megfeleltek a specifikált sémáknak.

```
for recipe in recipe_data.values():
    c.execute( _sql: '''
        INSERT INTO recipes (Recipe_Title, Prep_Time, Cook_Time, Image_URL) VALUES (?, ?, ?, ?)
    ''', _parameters: (recipe['Recipe_Title'], recipe['Prep_Time'], recipe['Cook_Time'], recipe['Image_URL']))
    recipe_title = recipe['Recipe_Title']
```

6.12. ábra Web scraping - adatbázis táblák feltöltése

A tábláknál továbbra is a korábban említett technikákat használtam, Python ciklusokkal és utasításokkal kiegészítve (6.12. ábra), ahol a ciklussal végig mentem az összes adatgyűjtő függvény által feltöltött *recipe_data* bejegyezésen. Minden táblához megadtam a szükséges paramétereket, és feltöltöttem azokat. A ciklus végén *commit*-oltam és zártam a *connect*-et, a táblák feltöltődtek.

6.2. Weboldal elkészítése

A weboldalt a korábbi fejezetekben többször említett Django keretrendszerben készítettem el, amely használatához először telepítettem a fejlesztői környezetet. Ez nagyon egyszerű, a CMD-ben szimpla *pip install* paranccsal telepítettem, innentől pedig a django-admin-ra hivatkozva készítettem egy saját projektet, a *startproject* utasítással. Ebben a projektben találhatók a szükséges alapbeállítások és scriptek, amik felelnek például a helyi webszerver futtatásáért.

Amíg a projekt a webes alkalmazás konfigurációit tárolja, a funkciók és oldalak a projektek belüli applikációkból fognak eredeztetni. Jelen esetben egy applikációra volt szükségem, amiben létrehoztam a specifikációban említett elemeket, azaz a futtatandó oldal back- és frontendjét.

Mielőtt ezeket elkészítettem, elvégeztem néhány változtatást a projektben, azon belül is a settings.py-ban. [13] A beállítandó dolgok a következők voltak:

- Az applikáció felvétele
- Az adatbázis felvétele

  Behelyeztem a projekt mappába, majd itt hivatkoztam a fájlra

- Útvonal megadása a statikus fájloknak létrehozott mappához

6.2.1. Modellek

A weblapon adatokat szeretnénk látni, ehhez kéréseket küldünk, amikre a view-k válaszolnak és visszaadják a modellek adatait. Ezt levezetve látható, hogy a weblap alapja az adatbázis, jelen környezetben a modellek. Emiatt először ezeket kellett elkészítenem, amit a *view* követ, majd végül a HTML, illetve annak statikus fájljai következnek.

Ezt a feladatot megkönnyíti az *inspect.db* parancs, ami megvizsgálja a beállításokban referált adatbázist és a felépítésének megfelelő modelleket ajánl. [14] Ez főképp legacy adatbázisoknál szükséges funkció, de az elkészítendő adatbázisnál is hasznos. Viszonylag közeli ajánlásokat adott, amiket kevés módosítás után könnyedén megfelelővé tudtam tenni. A kész modellek az adatbázis táblákat reprezentáló osztályok lesznek, melyeknek minden attribútuma a tábla egy oszlopának felel majd meg (6.13. ábra).

```
class Instructions(models.Model):
    id = models.AutoField(primary_key=True)
    recipe_title = models.TextField()
    order_number = models.IntegerField()
    text = models.TextField()

    class Meta:
        db_table = 'instructions'
```

6.13. ábra Instruction tábla modellje

Ezt követően pár konzol utasítással szinkronizálható az adatbázis tartalma a modellekkel. Ehhez a *migrations*-t [15] használtam, amivel kiterjeszthetők a Django modelljei és azok változtatásai a tényleges adatbázisra. Ezen belül is a *makemigrations* és *migrate* parancsokat használtam. Bármilyen későbbi adatbázis változtatáskor ezeket kell majd újra használni a modellek megfelelő frissítését követően.

6.2.2. Főoldal

Korábban többször említettem, hogy a megjelenítéshez egy HTML template szükséges, ami kérést küld a view felé. A view pedig választ ad vissza és a megírt függvényekkel közli a helyes adatokat. Azonban ahhoz, hogy a template egyáltalán elküldje ezt a kérést, ahhoz kell egy URL a HTML eléréséhez.

A Django-ban a manage.py tárolja a webapplikáció futtatásához szükséges beállításokat, a *python manage.py runserver* paranccsal pedig futtathatjuk azt. A parancsra a konzolon visszakapunk egy hiperhivatkozást, amire kattintva az oldalra jutunk, ez lesz a főoldal URL-je.

```
urlpatterns = [
    path('', views.main, name="main")
```

6.14. ábra URL készítés - app

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('website.urls'))
]
```

6.15. ábra URL készítés - projekt

Az URL-eket az urls.py-ban kell beállítani, amiből kettő van. Egy a projektben, egy pedig az applikációban. Az applikáción belüli URL beállítást végző kódrészletet a 6.14. ábra, a projekten belülit pedig a 6.15. ábra mutatja. [16]

View és Template

A következő lépés függvény készítése [17] „main” névvel, amelynek paramétere a *request*. A *request* egy http kérést reprezentál, amit a függvény végén kezel a *return*-ben, amivel egy *render* függvényt ad vissza. A *render* függvény http választ küld vissza a benne megadott paraméterekre, amelyek a request, társítva a html oldallal, amit egy kontextus követ.

```
def main(request):

    return render(request, template_name: 'main.html', context: {
    })
```

6.16. ábra Main függvény létrehozása

Az elkészített függvénybe tartalmat írva receptek hívhatók le és jeleníthetők meg. Ehhez létrehoztam egy QuerySet-et a recepteknek: *filtered_recipes = Recipes.objects.filter(image_url__isnull=False)*. A szűrésre annak ellenére szükség volt, hogy kevés egyéb cikk volt az oldalon, mert néha mert receptajánlások összesítéseit tartalmazta az oldal elkülöníthetetlen HTML elemekkel. Azonban ezek könnyen megkülönböztethetőek, mert nem tartalmaznak képeket. Ezekből nagyon kevés van, kevesebb, mint 10, de egyszerűbb itt kiszűrni ezeket, mint módosítani a scraping scriptet.

Mivel nem szerettem volna túl sok receptet listázni egy oldalon, ezért bevezettem az oldalváltást a paginatorral. Ennek működésére külön gombokat tettem a HMTL-be, ahogy ezt a 6.17. ábra mutatja.

```
paginator = Paginator(filtered_recipes, per_page: 51)
page_number = request.GET.get('page')
page_obj = paginator.get_page(page_number)
```

6.17. ábra Lapozás implementálása

A szűrt recepteket, vagyis a recept tábla elemeket, az említett render-ben behívtam. Ezt követően létrehoztam a megadott HTML-t, a main-t, amiben a listán végig iterálva felsoroltam a címeket (6.18. ábra).

```
<ul id="recipe-list">
    {% for recipe in page_obj %}
        <li> <div>{{ recipe.recipe_title }}</div></li>
```

6.18. ábra Receptek listázása

Ennek eredménye a receptcímek felsorolása az oldalon, de ennél többre van szükség. Ehhez először hiperhivatkozásokká változtattam a felsorolást, amikre kattintva a receptes részlet oldalra jutunk. Ehhez létrehoztam egy új üres függvényt a view-ban, egy új HTML template-t és az URL-t amire a hivatkozások visznek. A függvény neve *recipe_detail*, az oldalé *recipe.html*. Ahogy korábban említettem, az új URL-hez a receptek ID-jét használom az egyszerűség végett. Ehhez a „li” lista elemen belül (6.18. ábra) vettem egy „a” elemet, amiben a hiperhivatkozás *{% url 'recipe_detail' recipe.recipe_id %}*. Ezzel megoldottam, hogy a felsoroláson belül a címek url-ek legyenek, amik visszaadják a recipe_id-t a recipe_detail azonosítóval bíró URL útnak.

Túrófelfújt áfonyával, bevált receptúra alapján. Rizs nélkül, fantasztikus ízzel.
Sajtos panírozott karfiol és egy trükk, amivel a panírozás gyorsabbá tehető.
Sajtos-ropogós csirkefalatok felesleges olajban sütés nélkül

6.19. ábra Receptek listázása URL-ként

Az URL létrehozását a 6.20. ábra mutatja, ahol az URL dinamikusan épül fel [18] a megnyitott recept ID-ja alapján.

```
path('<int:recipe_id>/', views.recipe_detail, name='recipe_detail'),
```

6.20. ábra Dinamikus URL létrehozása

A receptekre kattintva a kapott URL-ek tehát a következőképp néznek ki: *127.0.0.1:8000/1/*. Ahol a loopback cím és a port mögötti, utolsó szám a kiválsztott recept ID-je.

Szűrők

Az oldal ebben a készültségi fázisban igencsak egyszerű, ezért elvégeztem rajta néhány, a specifikációban említett módosítást. Először is változtattam a receptek felsorolásán, amely során szükségem volt a receptek képeire. Másodszor szűrőket készítettem, alapanyag, kategória és különleges diéta alapján.

A képek hozzáadását a címek felsorolására használt sorokban „img" elemben soroltam a receptes táblánk *image_url* elemeit a linkekért felelős „*a*" elemen belül.

A szűrések Django-s kialakításának megkezdése előtt a szűréshez szükséges alapok elkészítésére volt szükség a különleges diétákhoz történő szűréshez.

Ehhez ismét az SQlite3-t használtam. Miután ezt behívtam, szótárakat készítettem *dietary_ingredients* néven (6.21. ábra), amik tartalmazzák az ellenőrizendő étrend nevét, és az ahhoz kötött alapanyagokat. A speciális étrendek a következők: cukormentes, laktózmentes, tojásmentes, gluténmentes és mag nélküli. Ha bármkor új étrendeket szeretnénk betenni, nem lesz több dolgunk a szótár kiegészítésénél.

```
dietary_ingredients = {
    "sugarfree": ["cukor", "csoki", "csokoládé", "puding", "méz", "lekvár", "kakaó"],
```

6.21. ábra Különleges étrendek szótára

Akárcsak a web scraping-nél, most is connection-t nyitottam az adatbázishoz, majd végig mentem a recept táblánk sorain, és új oszlopokat hoztam létre a szótárban definiált étrendjeinknek, ha még nem létezik azonos nevű oszlop abban a sorban (6.22. ábra).

```
existing_columns = set([column[1] for column in c.execute("PRAGMA table_info(recipes)").fetchall()])
for dietary_need in dietary_ingredients.keys():
    if dietary_need not in existing_columns:
        c.execute(f'ALTER TABLE recipes ADD COLUMN {dietary_need} BOOLEAN')
```

6.22. ábra Oszlopok létrehozása az étrendekhez

Egy SQL hívással lekértem a recept neveket és eltároltam azokat egy listába, amin ciklikusan végig iteráltam és létrehoztam egy üres szótárt *diatery_labels* néven, illetve behívtam a hozzávalókat az adatbázisból egy másik listába.

```
for dietary_need, dietary_ingredient_list in dietary_ingredients.items():
    if not any(any(dietary_ingredient.lower() in ingredient[0].lower() for
                   ingredient in ingredients) for dietary_ingredient in dietary_ingredient_list):
        dietary_labels[dietary_need] = 1
    else:
        dietary_labels[dietary_need] = 0
```

6.23. ábra Recept étrendnek való megfelelésének vizsgálata

Ezt követően végig iteráltam az összes étrenden és az azokhoz megadott alapanyagokon, és ellenőriztem, hogy megtalálhatók-e az éppen vizsgált receptünkben (6.23. ábra). Ha a felsorolt alapanyagok bármelyike megtalálható a receptben, akkor az nem felel meg az elvárásoknak, így a *label* értéke 0 lesz, ellenkező esetben pedig 1, ezek pedig a nemrég létrehozott új oszlopokba kerülnek.

```
c.execute('UPDATE recipes SET ' + ', '.join([f'{label} = ?' for label in dietary_labels.keys()])
          + ' WHERE Recipe_ID = ?', _parameters: (*dietary_labels.values(), recipe_id))
```

6.24. ábra Adatbázis frissítése étrendekkel

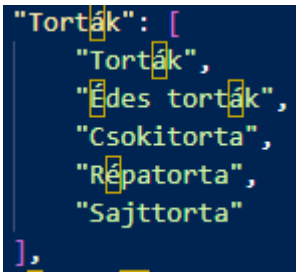
Ezután elkészítettem az új modelleket és migráltam a változásokat a 6.24. ábra kódrészletének segítségével.

Ezt követően a kategóriák számának a szűrését kellett megoldani. A fejlesztés jelen fázisában 196 egyedi kategória volt, ugyanis a receptek rengeteg alkategóriával is el voltak látva, például a tésztaféléken belül volt több spagetti recept, amiknek volt egy spagetti alkategóriájuk. Fő kategóriából összesen 26 db volt, ami egy átlátható mennyiség. A szűrésben látható kategóriák csökkentésére való lehetőségeim a következők voltak:

- változtatok az adatbázison, és csak a fő kategóriákat hagyom bent
- a kódban felsorolom, hogy melyik kategóriákat szeretném látni

Az adatbázison nem akartam változtatni, hiszen az alkategóriák bármikor jól jöhetnek. A választásom tehát a második opció volt, azonban ez nagy helyet foglalt volna a kódban, így az átláthatóság végett jobbnak láttam, ha egy külső fájlban szortírozom a kategóriákat, majd azt hívom meg. Az alkategóriák nem követnek automatizálható logikát így ezt a külső fájlt kézzel kellett megírnom.

Ehhez a fájlhoz a JSON formátumot választottam az olvashatósága, széleskörű támogatása és egyszerűsége miatt. A fájlban csoportokat hoztam létre, ezek a fő kategóriák, amelyek listázásra kerülnek majd, a csoportokon belül a hozzá tartozó kategóriákat tettem, azaz az alkategóriákat és a főkategóriát is. Erre mutat példát a 6.25. ábra.



6.25. ábra Kategória csoportok

Az alapok elkészítése után, elkezdtem a view módosítását, elsőképp a JSON fájl importálásával. Ez a *with open* függvénnyel [19] történt, amin belül megadtam a fájlt, a fájlon végzett műveletet, jelenesetben olvasást és a karakterkódolást (6.26. ábra). A fájlt magát egy Python szótárba helyeztem.

```
with open('kategorak.json', 'r', encoding='utf-8') as file:
    categories = json.load(file)
```

6.26. ábra JSON fájl megnyitása

Az adatbázis kiegészítésével és a JSON fájl beolvasásával teljesültek a feltételek a különböző szűrések kezeléséhez, amelyek közül a legfontosabbat, az alapanyag alapú szűrést valósítottam meg először.

Mivel az alapanyagok alapján szűrtem, először le kellett hívnom azokat egy változóba, későbbi felhasználásra. Ehhez a *request.get*-et használtam, melyet a 6.27. ábrakódrészlet mutat be.

```
ingredient = request.GET.get('ingredient', '')
ingredient_list = [ingredient.strip()
for ingredient in ingredient.split(',') if ingredient.strip()]
```

6.27. ábra Alapanyag kezelése szűréshez

A receptek behívásával ellentétben, a GET nem betölti a tényleges adatainkat, hanem a megadott 'ingredient' értékkel azonos értéket nyer ki a felhasználó általi kérésekből. Tehát egy alapanyag beírására létrehozott beviteli mezőbe írt szöveget használ és az alapján ad értéket neki, például, ha a felhasználó rákeres arra, hogy „sajt" azt tovább viszi a get-be, az URL módosul a következővel „?ingredient=sajt". Ez alapján lehet majd szűrni a felsorolt elemeket. A kódrészlet utolsó sora arra szolgál, hogy több alapanyagot is lehessen használni, azokat vesszővel elválasztva.

Tehát ha van egy ilyen nemüres alapanyag listánk, elkezdhetjük a szűrést. Ahogy azt a 6.28. ábra kódrészlete mutatja, ez egy üres set létrehozásával kezdődik, amibe a megfelelő receptek kerülnek majd. Majd végig megyünk a listába helyezett alapanyagokon és megnézzük, hogy szerepelnek-e a receptben.

```
if ingredient_list:
    matching_recipe_titles = set()
    for ingredient in ingredient_list:
        regex_pattern = r'\b{}\b'.format(re.escape(ingredient.lower()))
        matching_ingredients = Ingredients.objects.filter(ingredient__iregex=regex_pattern)
        titles_for_ingredient = set(item.recipe_title for item in matching_ingredients)
        if not matching_recipe_titles:
            matching_recipe_titles = titles_for_ingredient
        else:
            matching_recipe_titles &= titles_for_ingredient
    filtered_recipes = filtered_recipes.filter(recipe_title__in=matching_recipe_titles)
```

6.28. ábra Receptek szűrése alapanyag alapján

Ehhez egy *regex pattern*-t használtam, ami egy nagyon hasznos eszköz szövegek átkutatására, amiből a határfigyelés alkalmaztam. [20] A határ figyelés alatt a „\b" érhető, ami figyeli, hogy a keresett szó, jelen esetben alapanyag, teljes egészében legyen vizsgálva, azaz a határai között. Ez szükséges, hisz például, ha a sóra szűrünk, nem akarjuk a borsót visszakapni. Emellett kisbetűssé alakítottam az alapanyagot, hogy a kis és nagybetűk ne okozzanak problémát, valamint használtam a *re-escape*-et ami átugrik bármilyen különleges karaktert a tisztább keresésért és a *regex* zavartalan működése miatt.

Egy metódussal kigyűjtöttem az összes létező és *regex*-nek megfelelő alapanyagot a modellből és egy új listába tettem ezeket. Ezt követően megnéztem, hogy ezeket az alapanyagokat pontosan melyik receptből szedte ki, és eltároltam azon receptek címeit.

Ezután feltételes elágazásban, ha a *set* üres, vagyis ez az első vizsgált alapanyag, beletettem az eltárolt recepteket. Ha pedig nem üres, és ez nem az első alapanyag, amit vizsgálunk, akkor a két lista metszetét tároltam el, hogy valóban az összes alapanyagnak megfeleljen a szűrés. Végül visszaadtam ezt a listát az eredeti receptfelsorolásunknak, ezzel a receptek szűrése megtörtént.

```
<form id="filter-form" method="GET" action="{% url 'main' %}">
```

6.29. ábra Form elem a GET kérések lehetővé tételéhez

A HTML-en belül ezeknek a szűrőknek egy *form* elemet készítettem a GET metódussal (6.29. ábra), aminek az URL-es címfelsoroláshoz hasonlóan megadtam, hogy mégis melyik függvénynek adjon vissza kérést.

Az alapanyag szűrésnek ezen belül kellett egy input keresőmező, aminek az értéke *„value={{ ingredients }}”* lesz, a neve pedig *ingredient*. Kommunikál a 6.27. ábra elemeivel. Végül, akár csak a *page_obj* elemeket a címek megjelenítésekor, most is át kell adnunk az *ingredient*-et a *render* függvénynek.

Különleges étrend alapú szűrés

Az alapanyag alapú szűrésnél egyértelmű volt a kereső sáv használata, de itt felmerült, hogy vizuálisan milyen lehetőségünk van a szűrési opciókra. A leggyakrabban használt elemek általában: kereső sáv, lenyitható sáv, jelölőnégyzetek vagy csúszka. Kereső sáv már van és nem is felel meg a feladatnak, a lenyitható sáv elfogadható, de a kategóriákhoz sokkal jobban illik, a csúszka pedig ismét nem felel meg a feladatnak, a választásom tehát a jelölőnégyzetekre esett. Ezt azonban nem mindenki fogja használni, megfontolandó az említett lenyitható fül is (5.8. ábra). Használhatóság szempontjából fontos még, hogy a jelölőnégyzetek bepipálása indítsa el a szűrést, ne kelljen külön gomb hozzá.

A hozzá tartozó kód megvalósításához elsőként a GET paramétereket adtam meg, ahogy ezt a 6.30. ábra mutatja.

```
sugarfree = request.GET.get('sugarfree') == 'on'
lactosefree = request.GET.get('lactosefree') == 'on'
```

6.30. ábra Checkbox állapot lekérése

Itt beállítottam a jelölőnégyzetek státuszát. Ha azok bevannak pipálva, az eredmény 'on' értékű. Ezt ellenőrizve, az eredmények megfelelően True vagy False értékeket kapnak a változók (6.31. ábra).

```
if sugarfree:
    filtered_recipes = filtered_recipes.filter(sugarfree=True)
```

6.31. ábra Recept szűrése étrend alapján

Abban az esetben, ha a változó értéke True, szűrjük a recepteket. Ezt vissza is kell adni az oldalnak a *render* függvényben. Ehhez készítettem egy szótártat, amiben a jelölőnégyzet neveket és az azonos nevű változókat tároltam, például „'sugarfree': sugarfree", majd az egyszerűség kedvéért ezt a szótárat adtam vissza a *render*-nek.

Ezután elkészítettem hozzá a HTML-t. Az összes jelölőnégyzetet egy átfogó *div* elemen belül hoztam létre, ami egy gomb lenyomásával nyílik majd ki, addig is rejtve marad (6.32. ábra).

```
<input type="checkbox" id="sugarfree" name="sugarfree"
{% if checkbox_states.sugarfree %}checked{% endif %} onchange="this.form.submit()">
```

6.32. ábra Étrend checkbox-ok létrehozása

Ez az elem egy checkbox típusú input, ezen belül lekértem az állapotot, amire azért volt szükség, hogy ha bármikor újra töltjük az oldalt frissítés vagy lapozás miatt, az állapot ne vesszen el. Ezen túlmenően az *onchange*-re, azaz a jelölőnégyzeteket érintő változásra kértem egy küldést, hogy a jelölőnégyzet szimpla bepipálása már magában elindítsa a szűrést.

Kategória szűrés

A kategória alapú szűrés is egy GET kéréssel kezdődik, a *'category'* azonosítóra hivatkozva, és új változóba mentjük az eredményt.

Ha történt GET a kategóriánkra, azaz választottunk kategóriát, elkezdjük a szűrést. Maguk a kategóriák a Tag táblában (5.5. ábra) léteznek a recept címek mellett, tehát azt kell megnézni, hogy ahol a változónk értéke megegyezik-e egy kategória Tag-gel, ott mi a recept címe és ezekkel a címekkel kell tovább szűrni a felsorolt recepteket. Az ehhez készített kódrészletet a 6.33. ábra mutatja.

```
selected_category = request.GET.get('category', '')
if selected_category:
    matching_titles = Tags.objects.filter(tag=selected_category).values('recipe_title')
    filtered_recipes = filtered_recipes.filter(recipe_title__in=Subquery(matching_titles))
```

6.33. ábra Kategória alapú szűrés

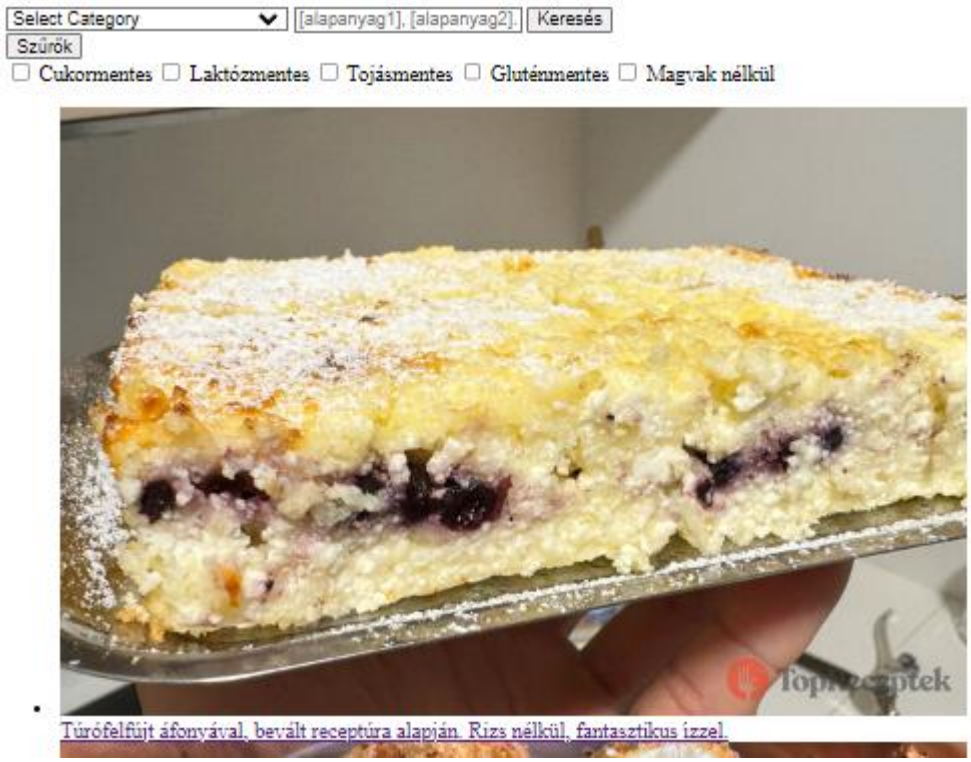
Ezután átadjuk a szükséges komponenseket a render függvénynek, majd áttérünk a HTML kódra, ami közel egyező az étrendszűréshez. Itt azonban nem kézzel hozzuk létre az elemeket, hanem ciklussal listázzuk őket.

```
<select name="category" onchange="this.form.submit()">
    <option value="">Válasszon kategóriát</option>
    {% for category in categories %}
    <option value="{{ category }}" {% if selected_category == category %}selected
    {% endif %}>{{ category }}</option>
    {% endfor %}
```

6.34. ábra Kategóriák felsorolása a weblapon

Minden lap újra töltésnél megvizsgáljuk, hogy a view-s függvény tartalmaz-e mentett kategóriát, és ha igen akkor annak megfelelően járunk el, ami ismételten lapozásnál és frissítéskor szükséges.

A szűrők hozzáadása és egyéb módosítások után a weboldalunk megjelenését a 6.35. ábra mutatja.



6.35. ábra Weboldal kinézete CSS előtt

Egész tömött, az alkotó elemek nagyrésze helyben van, már csak a lapozáshoz kell füleket biztosítani a megjelenítésen történő javítás előtt a statikus elemekkel.

Lapozás

A lapozásnál a *page_obj.number* adja vissza azt, hogy jelenleg hányadik oldalon járunk.

Maga a lapozás felépítése a következő: Amennyiben nem az első oldalon vagyunk, egy „<” megengedi a visszafelé lapozást, egy „<<” pedig az első oldalra ugrást, és ugyanez előre. Az oldalszámok az ikonok között lesznek láthatók. Az oldalszámra a következő esetekben van szükség:

- Ugrás az első oldalra

  Amennyiben az oldalszám nem 1

- Vizuális elemek

Az oldalszám felsorolása alapértelmezésben 1…79, viszont, ha a szám nagyobb, mint 3, akkor 1 … 3 … 79

- Ugrás az utolsó oldalra

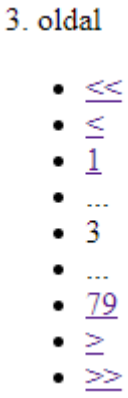
  Ha nem a legmagasabb számú oldalon vagyunk

```
{% if page_obj.has_previous %}
<li><a href="?{% for key, value in request.GET.items %}{% if key != 'page' %}{{ key }}=
{{ value|urlencode }}&{% endif %}{% endfor %}page={{ page_obj.previous_page_number }}"><</a></li>
{% else %}
<li><span class="disabled"><</span></li>
{% endif %}
```

6.36. ábra Példa a léptetésre (visszafelé léptetés)

Egyedül az előre és vissza léptetésnél nem használtam ezeket, ott a 6.36. ábr látható *has_previous*-t, illetve a *has_next*-et használtam, ami megnézi, hogy van-e a jelenlegi előtt vagy után oldal. Az összes léptetésre szolgáló funkció egy feltételes felsorolásban van, amire, ha nem teljesülnek a feltételek, akkor kikapcsolt állapotban van. Például a gomb, amivel az első oldalra ugrunk mindig látható, de ha a feltételek nem teljesek, rá-nyomni nem lehet. A nyilak lényegében hiperlinkek, amelyek az URL megváltoztatásával az új oldalra visznek, egyértelmű tehát, hogy az oldalszámon kívül a többi szűrési paramétert meg kell tartania az új oldalon is.

Ehhez a hiperhivatkozáson belül egy ciklussal végig mentem az összes request-ben lévő paraméteren az oldalszám kivételével, amiket a GET-tel kaptam, és ezeket újra alkottam az újonnani URL-ben a *{{ key }}={{ value|urlencode }}* (6.36. ábr) segítségével, ahol az urlencode az értékek url-nek megfelelő tárolásáért felel. Ehhez aztán hozzácsatoltam az oldalszámot, és kész az új URL, ami oldalt vált a szükséges paraméterek megtartása mellett.



6.37. ábra Lapozás

6.2.3. Főoldal dizájn megvalósítása

A főoldali dizájn megvalósításához a specifikált felépítést követtem (5.8. ábra). Ehhez először létrehoztam a „logo”-t, amihez egy láthatóan elkülönített fejléc szükséges, ami több karaktert ad az oldalunknak. Ezt az elkülönítést legjobban egy egyedi betűtípus használatával tudtam megmutatni. A Goolge Fonts egy tökéletes forrás ilyen betűtípusok kutatására, a választott típusunkhoz kapunk a projektünkbe beilleszthető HTML kódot, aminek a beillesztésével már hivatkozhatunk is a betűtípusra a CSS-ben.

Létrehoztam az alapot a logo-nak, ez egy szimpla div, benne egy h1 elemmel későbbi formázáshoz. Hogy ezt formázni tudjam, létrehoztam egy CSS fájlt abban a mappában, amit a beállításoknál megadtam statikus útvonalként, majd ezt a fájlt behívtam a HTML-be (6.38. ábra).

```
{% load static %}
<link type="text/css" rel="stylesheet" href="{% static 'css/style.css' %}">
```

6.38. ábra Statikus CSS fájl importálása Djangoban

A CSS-en belül egyszerű formázásokat alkalmaztam, középre igazításokkal, szélesség és magasság értékek megadásával, egyszerű háttérszínek beállítása mellett. A méret értékeknél különböző nagyságokat próbáltam ki, amíg meg nem találtam a nekem tetsző mértékeket. A keresőt és kategória szűrést egymást mellé tettem, a négyzeteket pedig lenyitható fülbe helyeztem el. Ehhez JavaScriptet kellett használnom, így előbb megcsináltam

a lapozást és a receptfelsorolás kinézetét. A legtöbb használt kód ismét gyakori CSS sorokból állt, középre igazításhoz, margók beállításhoz, a színek beállításához. Különös figyelmet kellett helyezni arra, hogy melyik elemhez írunk stílust: nem elég csak név vagy id szerint behívni az elemeket, olyan hivatkozásokat írni, mint például konténer specifikus HTML elemén belül több specifikus elem, egyedi azonosító alapján.

Egyedi változásokat kellett készítenem dinamikus esetekben is, mint például a kurzor lebeg az elem felett.

Mindkettőnél fontos az is, hogy először kivegyem a felsorolásokra alapból ráhelyezett és szükségtelen elemeket, mint például a felsoroláskor látható pontok (6.37. ábra).

```
.pagination-container ul,
.pagination-container ul li {
    list-style-type: none;
    margin: 0;
    padding: 0;
}
```

6.39. ábra Lapozás CSS részlet

A lapozáson végzett változtatásaim ennek megfelelően a következők:

- Elemek sorba rendezése, ehhez display:flex beállítása a konténeren, majd inline-block a listaelemeken
- Középre igazítás, margók beállítása
- Lekerekített szélű, az alap háttérszínnél sötétebb szín beállítása a gombokhoz, dinamikus színekkel az oldalváltáshoz és a jelenlegi oldal jelzéséhez

6.40. ábra Lapozás CSS végeredmény

A receptfelsorolásnál beállítottam a következőket:

- Konténerben ismét flex megjelenés, warp-pal, tehát a konténerből túlfolyó elemek új sort kezdenek. Emellett space-around tartalomelhelyezés, tehát a szabad teret egyenlő elosztja az elemek között.

  Soronként három recept van, de ha esetenként nem jön ki a három, a kevesebbet is jól el fogja rendezni.

- Egyes képek kisebbek, mint a többi, így beállítottam az object-fit: cover-t. Ezzel az összes kép egységesen kitölti majd a saját terét, nem látszanak a méretkülönbségek.
- A feliratok pozícióját abszolútra állítottam, ami a legközelebbi elemhez igazítja őket, és engedi, hogy átfedés legyen köztük. Esetünkben a szöveget a képhez igazítja.
- Dinamikus változások a képen lebegő kurzor esetén.

6.41. ábra Receptfelsorolás kinézete CSS után

Az eredti terv megvalósításához már csak a négyzetek lenyitható fülbe való behelyezése hiányzik. A lenyitható fülhöz egy gomb elemet hoztam létre a négyzetek fölött, majd ennek az állapota alapján változtattam a négyzetek láthatóságát [21]. Ezen belül a localStorage tulajdonságot használtam. [22] Ez a gombállapotot helyileg menti, így, ha a felhasználó lenyitva használja az oldalt, akkor ezt megjegyzi és a következő látogatáskor is nyitva marad, így javítva a használati élményt.

A scriptben először létrehoztam egy eseménymegfigyelőt. Ez a megfigyelő megkeresi a lenyitható gombot a HTML fájlban, figyeli azt és kattintás esemény érzékelése esetén bekapcsol egy függvényt (6.42. ábra).

```
var collapsibleButton : Element  = document.querySelector( selectors: '.collapsible');
collapsibleButton.addEventListener( type: 'click', toggleCheckboxes);
```

6.42. ábra Lenyitható fül - Eseményfigyelő kattintáskor

Ez a függvény ment egy szótárat a localStorage-ba, ami megmondja, hogy a lenyitható fül nyitott vagy zárt állapotú, illetve állít a négyzetek láthatóságán (6.43. ábra).

```
function toggleCheckboxes() : void  {
    checkboxes.style.display = checkboxes.style.display === "block" ? "none" : "block";
    localStorage.setItem('collapsibleState', checkboxes.style.display === "block" ? 'open' : 'closed');
}
```

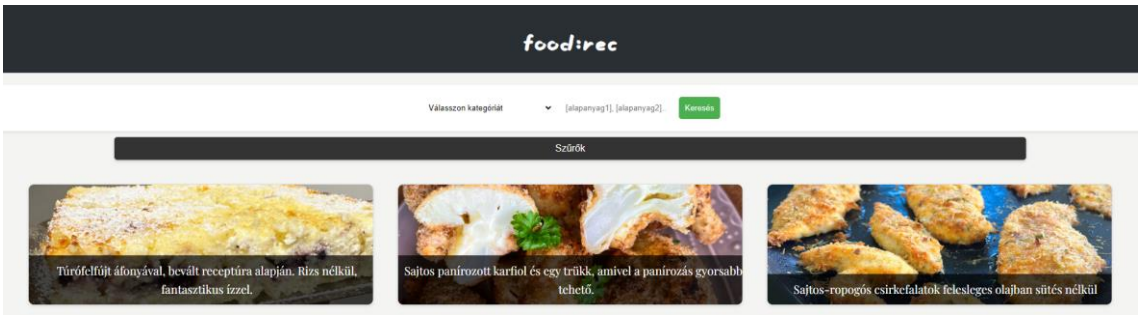
6.43. ábra Lenyitható fül állapotának mentése

Így történik a változások kezelése. Mindez egy nagyobb függvény része (6.44. ábra), amiben látható, hogy a jelenlegi állapot ellenőrzésére a kód elején kerül sor és ha van egy előző változatunk, akkor azt a helyileg tárolt értékek alapján visszaadja.

```
document.addEventListener( type: "DOMContentLoaded",  listener: function() : void  {
    var isCollapsibleOpen : string  = localStorage.getItem( key: 'collapsibleState');
    var checkboxes : HTMLElement  = document.getElementById( elementId: "checkboxes");
    if (isCollapsibleOpen === "open") {
        checkboxes.style.display = "block";
    } else {
        checkboxes.style.display = "none";
    }
```

6.44. ábra Lenyitható fül lenyitása

A főoldal kinézete a változtatásokat követően a 6.45. ábra mutatja.



6.45. ábra Főoldal kinézete

A következő lépés a receptek részleteit mutató oldal elkészítése.

6.2.4. Másodoldal

A Főoldalnál bemutatottakhoz hasonlóan most is a view-val kezdünk, a már előre létrehozott üres függvényünk kiegészítésével. Ez a függvény nem csak a request-et kapja vissza, hanem a recept ID-t is az URL kiépítéshez. Ehhez az ID-hoz tartozó recept részletei szükségesek, úgyhogy először behívtam azt, majd a cím alapján lekértem az alapanyagokat is (6.46. ábra).

```
def recipe_detail(request, recipe_id):
    recipe = Recipes.objects.get(pk=recipe_id)
    ingredients = Ingredients.objects.filter(recipe_title=recipe.recipe_title)
```

6.46. ábra View függvény a receptek részletezéséhez

Az alapanyagokat csoportosítva kell felsorolni. Ehhez létrehoztam egy szótárt, amiben maga a csoport egy lista kulcs elem lesz, és a hozzá tartozó alapanyagok és azok mennyiségei az értékek (6.47. ábra).

```
grouped_ingredients = {}
for ingredient in ingredients:
    part = ingredient.part
    if part not in grouped_ingredients:
        grouped_ingredients[part] = []
    grouped_ingredients[part].append((ingredient.ingredient, ingredient.amount))
```

6.47. ábra Alapanyagok megfelelő részhez való társítása

Az utasításokat az alapanyagokhoz hasonlóan receptcím alapján lehívtam, ezután egy szűrést végeztem és az új változóban csak az egyedi utasításokat tettem ezzel kiküszöbölve bármilyen lehetséges ismétléses hibát.

Az elemeket visszaadtam a *render*-nek (6.48. ábra), és áttértem a HTML-re.

```
return render(request, template_name: 'recipe.html', context: {
    'recipe': recipe,
    'grouped_ingredients': grouped_ingredients,
    'instructions': unique_instructions,
})
```

6.48. ábra Receptrészletek függvény elemeinek renderelése

A HTML-ben először beraktam a *logo*-t. Majd külön konténereket hoztam létre a következő elemeknek:

- Receptcím, kép
- Elkészítési idő
- Alapanyagok, lépések

Az elkészítési időnél két paraméterem volt, előkészületek és elkészítés. Maguk a megjelenítendő adatok csak az idők, tehát például 10 perc, ez elé kell írni, hogy ez mire vonatkozik. Mivel nem minden receptnél volt megadva mindkét adat, így feltételes elágazást készítettem, ami csak akkor jeleníti meg a szükséges kiírást, ha van hozzá adat (6.49. ábra).

Kicsit eltérő még az alapanyagok listázása, ahol dupla ciklust használtam először a szükséges rész kiírására, majd az alapanyag és mennyiség egymás melletti kiírására, ha mindkettő adott.

```
{% for part, ingredients in grouped_ingredients.items %}
    <h3>{{ part }}</h3>
    <ul>
        {% for ingredient, amount in ingredients %}
            {% if amount %}
                <li>{{ amount }} - {{  ingredient }} </li>
            {% else %}
                <li>{{ ingredient }}</li>
            {% endif %}
        {% endfor %}
    </ul>
{% endfor %}
```

6.49. ábra Receptek listázása a megfelelő részhez

Betettem továbbá *hr* elemű elválasztó vonalakat. Az ezzel kapott eredményt a 6.50. ábra mutatja.

6.50. ábra Másodoldal felsorolásai

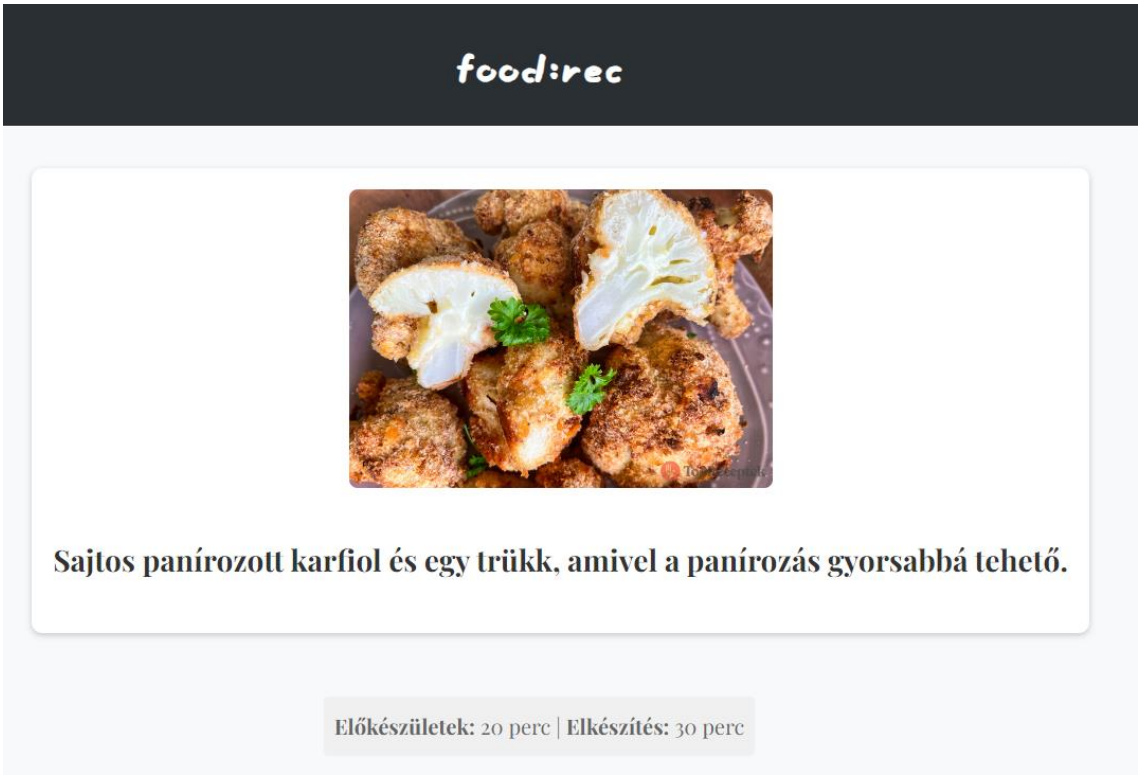
Ezután áttértem a statikus elemekre, ezek a CSS és ismét JavaScript, ezúttal animációk miatt. Az elemek a vizuális hierarchia részletezés óta ismertek (5.7. ábra).

Recept és Képnév

A receptet és képnevet egy középre igazított flexbox-ba tettem, ami engedi, hogy egy konténeren belüli több elem egymás mellett flexibilisen szerepeljen. Megkerestem és beállítottam a megfelelő méreteke és a tetsző betűtípust.

Egyéb részletek

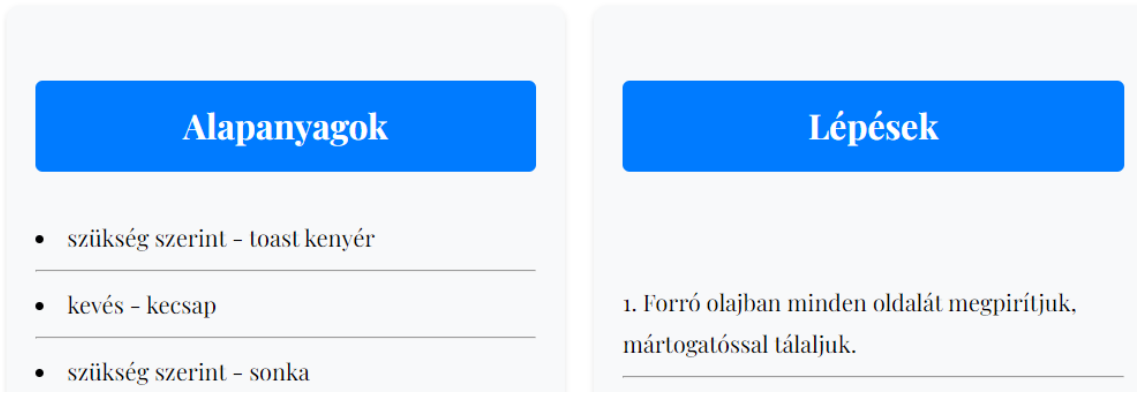
Ezek az elkészítési idők középre és a cím alá való helyezése, színek és betűtípusok választása, illetve az elválasztó vonalak szépítése.



6.51. ábra Másodoldal a CSS után - Receptcímek

Alapanyagok és az elkésztés lépései

Az alapanyagok és az elkészítés lépéseinek megjelenítéséhez ismét flexbox-ot használtam, emellett elkülönített és megkülönböztethető színű háttereket készítettem az elemeknek, távolságok definiálását, betűtípusok beállítását végeztem el.



6.52. ábra Másodoldal a CSS után - Alapanyagok, lépések

Készítettem animációkat is, ezek a következők:

- Recept cím: Fade-in
- Elkészítései idő: Fentről lecsúszás
- Lépések. Becsúszás jobbról

Ezek egyszerű mozgások, amiket keyframekkel [23] valósítottam meg. Az említett JavaScriptek pedig felelnek majd azért, hogy ezek az animációk a helyes időben jelenjenek meg. Ehhez a 6.42. ábra kódjához hasonló eseményfigyelőket hoztam létre, például a lépések akkor csúsznak be oldalról, amikor a teljes elem látható lesz, teljesen belép a *viewport*-ba.

6.2.5. Feladat bővítése

Alapanyag javaslatok

Az alapanyagok könnyebb böngészése érdekében egy új funkciót készítettem, ami automatikus javaslatokat tesz a felhasználó részére, amikor a keresősávban gépel.

Ennél a feladatnál az első probléma, amivel szembe kell néznem az maga az adatbázis, ugyanis 5333 egyedi alapanyagot tartalmaz. Ez főképp abból adódik, hogy a recepteket felhasználók töltötték fel, és nem lettek megfelelően adminisztrálva a publikáláskor tehát vannak olyan eseteink, ahol egyetlen alapanyag mezőben soroltak fel több fűszert, vagy specifikáltak valamit például túró helyett túró (zsírosabb), ezzel egyedivé téve az alapanyag mezőt. Ezek szükségtelenül megtöbbszörözik a javaslatokat, ami így átláthatatlanná válhat. A hiba javítására a következő megközelítést alkalmaztam:

- Alapanyagok csoportosítása, akárcsak a kategóriáknál
- Szűrők beállítása a kategorizálhatatlan eseteben

  Ne jelenlenek meg olyan ajánlások, amikben számok, különleges karakterek szerepelnek

- Elválasztó elemek, például vessző, ’és’ karakterlánc észlelése, a felsorolt elemek külön kezelése

A kategóriákhoz létrehoztam egy új scriptet. A script feladata beolvasni egy SQL lekérdezéssel készített adatbázist, ami tartalmazza az alapanyagokat és azok gyakoriságát,

majd kiépít egy új szótárt belőlük. A szótár készítését a leggyakoribb alapanyaggal kezdtem, létrehoztam hozzá egy csoportot, amibe beletettem önmagát, majd átnéztem az összes különböző alapanyagot, és ha egyezést találtam, akkor beletettem a csoportba. Ciklikusan végig mentem az alapanyagokon, és csoportokat hoztam létre azoknak az alapanyagoknak, amik még nem részei egy másik csoportnak sem.

Ahhoz, hogy az elemeket megfelelő sorrendben nézzük át, használhatjuk a Pythonba beépített lambda kifejezést (6.53. ábra). Ennek három paramétert kell megadni, hogy mit akarunk átnézni, mi szerint akarjuk átnézni és milyen sorrendben. Mivel a szűrt adatbázisban kutatunk, ennek csak két eleme van, az alapanyag neve és a gyakorisága. A második elem, azaz a gyakoriság szerint nézzük át az elemeket, ami az „x[1]”. Az utolsó paraméter a sorrend megadására szolgál, ahol True a csökkenő és False a növekvő, ezek közül a csökkenő sorrendet használtam.

```
sorted_ingredients = sorted(ingredients_data, key=lambda x: int(x[1]), reverse=True)
```

6.53. ábra Alapanyagok átnézése gyakoriság alapján

Ezután ciklust indítottam, ami végignézi az alapanyagokat és feltölti az előre létrehozott üres szótárt. Ehhez mindent kisbetűssé alakítottam a kis- és nagybetű érzékenység miatt, majd megnéztem, hogy a soron következő alapanyag szerepel-e valamelyik csoportban is, ha igen, nem kap saját csoportot (6.54. ábra).

```
if any(ingredient_lower in group for group in ingredient_groups.values()):
    continue
```

6.54. ábra Csoportok ellenőrzése

Itt használtam az *any* függvényt, ami iterálható elemeket néz át és True vagy False eredményt ad vissza az alapján, hogy a keresett elem benne van-e a vizsgált elemekben. Amennyiben az alapanyag már része egy csoportnak, a *continue* utasítást adtam meg, ami kihagyja a ciklus többi részét, egyenesen a következő alapanyag megvizsgálására ugrik.

Ha nincs benne egy csoportban sem, megnéztem, hogy illik-e bármelyik csoporthoz. Ha nem, ismét vizsgáltam, de ezúttal szavakra bontva, azaz mind a csoportot, mind az alapanyagot szavakra bontottam és egyezést kerestem (6.55. ábra).

```
for existing_group, ingredients in ingredient_groups.items():
    for sub_ingredient in ingredients:
        if f" {sub_ingredient.lower()} " in f" {ingredient_lower} ":
            ingredient_groups[existing_group].append(ingredient)
            break
```

6.55. ábra Alapanyagok csoportosítása szavak szerint

Így megoldottam azt, hogy az olyan esetek is mint például a „főtt tojás” is bekerüljön a tojás kategóriába, anélkül, hogy az olyan esetek, mint a borsó bekerüljön a bors kategóriába.

Előfordul viszont, hogy a kategorizálandó alapanyag nem teljesen elkülönített, egyező szó, például a feketebors – bors, úgyhogy egy kis utómunkát kellett végeznem kézzel is.

Visszatérve a kódra, ha az alapanyag egy csoportnak sem felel meg, és nincs csoportja, akkor készül neki egy. Miután pedig készen vannak a csoportok, json fájlba mentettem őket, akárcsak a kategóriáknál tettem.

```
"vaj": [
    "vaj",
    "puha vaj",
    "olvasztott vaj",
    "vaj vagy margarin",
    " vaj",
    "margarin vagy vaj",
    "vaj kenéshez",
    "hideg vaj",
```

6.56. ábra Alapanyag csoportok felépítése

Ezzel az 5333 számot 1384-re csökkentettem, amiből 980 egyedi alapanyag maradt, a maradék 4353 alapanyagot pedig 404 csoportba sűrítettem. Mivel ezek a csoportok emberi átnézésre is szorulnak ezért előfordulhatnak esetlegesen kimaradt összevonások, amik még jobban csökkenthetnék a csoportszámot.

Ezután megírtam a view függvényeket. Első lépésben behívtam a fájlt, akárcsak a 6.26. ábra esetében és egy új query változóba mentettem a GET-től várt visszatérő értéket, ami alapján a szűrés történik majd, akárcsak a többi előzetes szűrésnél. Mint az alapanyag

szűrésnél, a csoportok itt is vesszővel elválasztva jeleníthetők meg a kétcsoportos szűrés miatt, tehát ismét implementáltam a *strip* és *split* függvényeket (6.27. ábra).

Ezután megkerestem a keresésnek megfelelő csoportokat. Ehhez végig iteráltam a szótáron, azon belül is azt nézve, hogy a beérkezett keresés (a q elem), megtalálható-e a csoport nevében vagy első elemében (6.57. ábra). Mindezt kisbetűs összehasonlítással a nagybetűs különbségek elkerülése végett. Ha van találat, a csoportnév bekerül a „találatok" listába, majd ezen lista elemeit hozzákapcsoltam egy másik, ajánlások nevű listához az *extend* függvénnyel [24]. Ez a függvény az *any*-hez hasonlóan iterálható elemekkel dolgozik, specifikusan kiterjeszt egy létező listát az elemek hozzáadásával.

```
for q in queries:
    matches = [group for group, ingredients in
    ingredient_groups.items() if q in ingredients[0].lower() or q in group.lower()]
    suggestions.extend(matches)
```

6.57. ábra Alapanyag csoportok szűrése

Mivel csak a több alapanyagos csoportokat szerettem volna megjeleníteni, ezért kivettem az ajánlások közül az ennek nem megfelelő csoportokat egy új listába (6.58. ábra).

```
initial_suggestions = [s for s in suggestions if len(ingredient_groups[s]) > 1]
```

6.58. ábra Egyalapanyagos csoportok kiszűrése

Amennyiben ennek nincs visszatérő értéke, akkor nincs a keresésnek megfelelő csoport. Ezen esetben újra el kell végezni az előző műveleteket, ezúttal az egyalapanyagos csoportok kiírásával (6.59. ábra).

```
if not initial_suggestions:
    for q in queries:
        matches_single = [group for group, ingredients in ingredient_groups.items() if
        (q in ingredients[0].lower() or q in group.lower()) and len(ingredients) == 1]
        initial_suggestions.extend(matches_single)
```

6.59. ábra Egyalapanyagos csoportok szűrése

Végezetül készítettem még egy utolsó szűrést is, ami végig iterál az összes lehetséges ajánláson és csak akkor engedi át őket, ha abban nincsenek különleges karakterek, például vessző, plusz jel, illetve számok vagy olyan karakterláncok, mint az „és". (6.60. ábra)

```
filtered_suggestions = [s for s in initial_suggestions
                        if not any(x in s for x in [',', '+', '(', ')'])
                        and 'ízlés szerint' not in s
                        and 'szükség szerint' not in s
                        and 'ás' not in s
                        and 'és' not in s
                        and 'vagy' not in s
                        and not re.search( pattern: r'\d|[/\\%]', s)]
```

6.60. ábra Nem kívánt ajánlások szűrése

Itt ismét korábban bemutatott megoldásokat használtam, mint az *any* függvény (6.54. ábra), és a *regex* (6.28. ábra). A *regex*-ben ezúttal szűrtem a következőket:

- \d – decimális számokat
- per és százalékjeleket

amik között a „|” karakter egy *vagy* operátor.

Az ajánlásokhoz, szükség volt egy GET kérésre, amit az eddigiekkel ellentétben most JavaScript-ben hoztam létre, illetve meg kellett adni, hogy mit adjon vissza a függvény. A visszaadott érték eddig *render* volt, most azonban *JsonResponse* lesz „suggestions” névvel.

Ezután JavaScriptben létrehoztam a függvényt, majd első lépésként létrehoztam két változót. Egyet, ami a keresőbe írt szöveget tárolja, és egy másikat az ajánlások megjelenítéséhez. A második változóhoz HTML elemet is létrehoztam, ami egy egyszerű *div*.

Amikor magukat az ajánlásokat keressük, aszinkron módon kell dolgoznunk [25]. Tehát, a fő oldal nem fog várakozni az ajánlások befejezésére, kiválasztására, az oldal attól függetlenül teljes mértékben használható marad. Ehhez a Fetch API-t [26] alkalmaztam, ami lehetővé teszi a felhasználó által gépelt tartalom aszinkron elérését.

```
const fetchSuggestions = async (query) : Promise<...>  => {
    try {
        const response : Response  = await fetch( input: `/api/suggestions?query=${query}`);
        const data = await response.json();
        return data.suggestions;
```

6.61. ábra Ajánláslekérés Fetch API-jal

Ezt a *fetch* függvényben behívjuk: a *fetch* függvénnyel GET-et küldünk a létrehozott *JsonResponse*, a „suggestions”, felé, aminek az értéke *query* lesz. Ebből ezután kinyerjük a tényleges adatokat a *json* metódussal, majd megkapjuk a visszakapott ajánlásokat a *return*-ben (6.61. ábra). Az ajánlások megjelenítésére új ciklust készítettem (6.62. ábra).

```
suggestions.forEach(suggestion => {
    const suggestionElement : HTMLDivElement  = document.createElement( tagName: 'div');
    suggestionElement.textContent = suggestion;
    suggestionElement.classList.add('suggestion-item');
    suggestionContainer.appendChild(suggestionElement);
});
```

6.62. ábra Ajánlások megjelenítése

Ez az új *forEach* ciklus végignézni az összegyűlt ajánlásokat, létrehoz nekik egy új *div* HTML elemet, beilleszti őket az új elembe, ad nekik egy osztályt a CSS-ben való stilizáláshoz, majd megjeleníti az új *div*-et a létező, script elején beolvasott HTML elemhez.

Az ajánlások tényleges lehívásához szükség van még egy aszinkron függvényre. Ez a függvény fogja meghatározni a *query* paramétert, amire az első függvény hivatkozott, illetve ez fogja meghívni a többi másik függvényt is (6.63. ábra).

```
ingredientInput.addEventListener( type: 'input',  listener: async (event : Event ) : Promise<void>  => {
    const query : string  = event.target.value.trim().toLowerCase();
    if (query.length > 0) {
        const suggestions = await fetchSuggestions(query);
        displaySuggestions(suggestions);
```

6.63. ábra Eseményfigyelő az ajánlások megvalósításához

Ez az új függvény egy eseményfigyelő, ami az nézi, hogy a felhasználó gépel-e valamilyen input mezőbe. Egy ilyen mező van, a kereső. Ha igen, kisbetűssé alakítja az írtakat, illetve a *trim*-mel kiszűri a szóközöket, majd beteszi az eredményt a *query*-be. Ha a *query*-ben egynél több karakter szerepel, meghívja az első függvényt, bekéri a passzoló ajánlásokat, majd megjeleníti azokat a második függvénnyel.

Ehhez egy kattintásokat figyelő eseményfigyelőt készítettem, ami beteszi a rákattintott elemeket a keresőbe, figyelve arra, hogy ha a jelenleg a keresőben lévő tartalom vesszőt

tartalmaz, akkor ne írja felül, hisz vesszővel választjuk el a különböző alapanyagokat, amikor több alapján szeretnénk szűrni.

A csoportok miatt, ha például a vajra keresünk rá, akkor is visszakapunk olyan ételeket, amiben csak „hideg vaj” található alapanyagként. Eredményképp tehát ajánlásokat kapunk a 6.64. ábra mutatta formában.

6.64. ábra Ajánlások működése

További ajánlások

Tekintve, hogy az alap feladatom a receptajánlás, ezt a kiválasztott recept oldalán is megvalósítottam. Ehhez kiegészítettem ki a 6.2.4. fejezetben írt *view* függvényt és felsoroltam az oldal alján egyéb olyan recepteket, amelyekben elég egyező alapanyag van, illetve van közös kategóriájuk.

Ezt a kódban már gyakran használt Django szűrésekkel valósítottam meg, mint az „objects.filter”. A hasonló receptek szűréséért a 6.65. ábra kódja felel.

```
similar_recipes_titles = Ingredients.objects.filter(
    ingredient__in=ingredient_names
).values('recipe_title').annotate(
    count=Count('recipe_title')
).filter(count__gte=6).values_list('recipe_title', flat=True)
```

6.65. ábra Hasonló receptek találása alapanyag alapján

Itt az alapanyag nevek alapján szűrtem a táblát, amikhez hozzácsatoltam a receptcímet. Eredményképp a következőt kaptam: {'hasonló_recept1': egyező alapanyag 1}, {'hasonló_recept1': egyező alapanyag 2}, tehát minden hasonló recept annyiszor jelenik meg, ahány egyező alapanyagom van. Az *annotate* használatával specifikáltam egy olyan számolást, aminek az eredménye számban megadja, hogy melyik receptben hány egyező alapanyagunk van. Ezután kiszűrtem azokat a recepteket, amelyekben nincs legalább hat egyező recept a *count_gte*-vel, ahol a *gte* jelentése nagyobb vagy egyenlő. A szűrt receptcímeket listaként visszaadtam egy új listának.

Az új listát frissítettem, ezúttal a tényleges recept táblából vettem ki a címeket, és ügyeltem arra, hogy ne tegyem a listába azt a receptet, aminek az oldalán jelenleg tartózkodunk. Ehhez az *exclude* függvényt használtam.

Ismét készítettem egy új listát, ebbe beletettem a jelenlegi recept kategóriáit egy egyszerű *filter*-rel, majd elindítottam egy feltételes elágazást, ahol ismét frissítetten a szűrt recept listát. Itt ugyanazt a logikát használtam, mint a kezdeti szűrésnél, viszont elég, ha egyetlen egyezés is van, ezért kihagytam a köztes függvényeket a kezdetleges szűrés után egyből a *value_list*-re ugrással. Ennek oka az volt, hogy túl sok egyezés esetén nem szerettem volna telerakni az oldalt túl sok recepttel, ezért limitáltam a megjelenő recepteket 29-re, ehhez az „order_by [:29]"-et használtam.

A receptek részleteit mutató oldalak alján tehát a megfelelő hasonlóságú receptek megjelennek (6.66. ábra).

6.66. ábra Hasonló recept ajánlások

Navigációs gombok

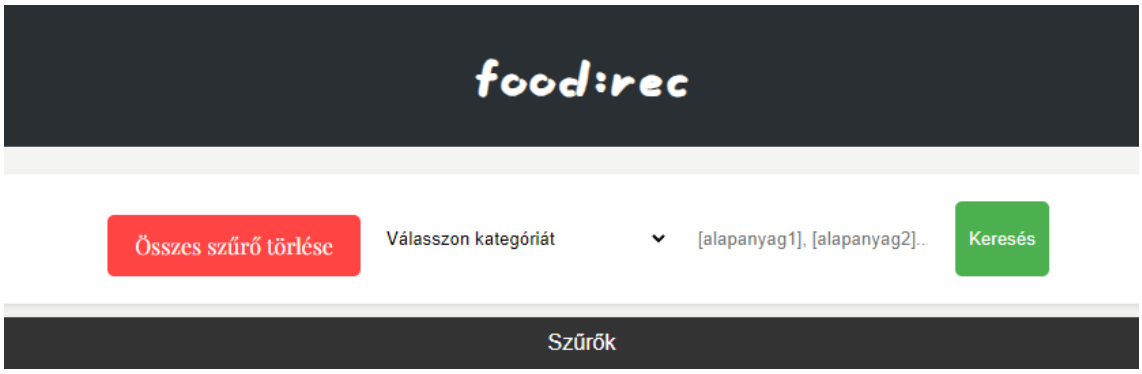
Az oldalról ebben a fázisban még feltűnően hiányoztak a navigációs gombok. Összesen két gombra van szükség: az egyik törli az eddigi szűréseket a főoldalon, a másik pedig visszavisz főoldalra a receptek oldaláról. Ehhez készítettem egy új hiperhivatkozást a HTML-ben, ami mindkét esetben a főoldalra visz, amihez a main nevezetű azonosítóval ellátott URL utat kellett megadni paraméterként (6.67. ábra).

```
{% if request.GET %}
<a href="{% url 'main' %}" class="reset-button">Összes szűrő törlése</a
{% endif %}
```

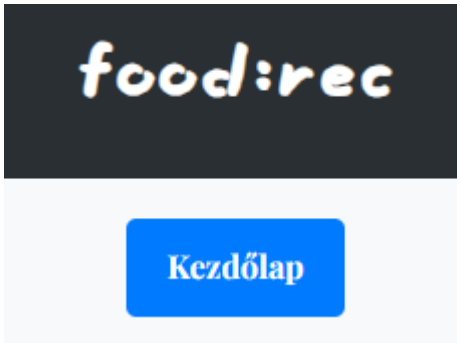
6.67. ábra Navigációs gomb elkészítése

A főoldalon való megjelenéshez hozzáadtam egy extra feltételt a gomb megjelenéséhez: legyen bármiféle aktív szűrésünk. Mivel bármilyen, az oldalon történt változást a *request.GET* fog közölni, ezért ehhez megvizsgáltam, hogy történt-e ilyen kommunikáció, szóval adunk-e éppen vissza bármilyen értéket.

A navigáció megkönnyítése végett így a főoldalon amint szűrést végzünk egy új gomb jelenik meg (6.68. ábra), illetve a receptrészletek oldalán pedig folyamatosan helyet foglal egy navigációs gomb, amivel visszatérhetünk a főoldalra (6.69. ábra).



6.68. ábra Gomb a szűrők törlésére



6.69. ábra Navigációs gomb a kezdőlapra visszatéréshez

7. ELEMZÉS

A végeredményt tekintve az elkészített oldal és kód követi a megfogalmazott specifikációt, teljesíti a feladatkiírásban definiált követelményeket. A kinézet modern, barátságos és könnyen átlátható, a különböző szűrők egymáshoz közel helyezkednek el, használatuk egyszerű és érthető. A végeredmény követi a specifikált weblap dizájnt. A létrehozott kódot a specifikáció szerint valósítottam meg, függvények és logika követi a tervet.

A recepteket szűréséhez több alapanyag is beírható, az ajánlások pedig segítenek a felhasználónak az alapanyagok könnyebb kiválasztása érdekében. Az oldal mindenképp használható, habár van hely a fejlesztéseknek.

A backend frissítések főképp a következők lehetnének:

Adatbázis javítása

A jelenlegi adatbázis átlátható, de integritása javítható lenne például azzal, hogy a receptcím minden táblába való elhelyezése helyett a recept ID-t használnám idegen kulcsként.

Emellett az adatbázis jelenlegi adatbevitele igazán amatőr. Az eredeti weboldal látszólag felhasználók által feltöltött receptekből épül fel, akik nem mindig megfelelően végezték el a feltöltést, illetve nem volt megfelelő adminisztráció a publikáláskor. Emiatt sok receptben gyakori hiba, hogy például az alapanyag mezőbe tették az alapanyag mennyiségét is. A receptek emellett nagyon „háziasak”. Az alapanyagok sokszor olyan mértékegységekben vannak megadva, mint az evőkanál. Ez a tényleges recept elkészítésénél barátságosabb, de csökkenti a szűrési lehetőségeket.

További szűrések

Az említett hibák változtatása után bevezethető lenne például egy kalória alapú szűrés is, azonban a jelenlegi alapok nem állnak készen ennek az implementálására. Készíthető lenne továbbá szűrés az elkészítési idő alapján.

Felhasználóbarátság

A továbbiakban beépíthetők lennének olyan felhasználóbarát elemek, mint a kedvenc ételek elmentése.

8. ÖSSZEFOGLALÓ

A szakdolgozatban egy receptajánló rendszert kellett megvalósítanom, amiben a felhasznált alapanyagok, illetve egyéb szűrők alapján képesek szűrni az általam web scraping technológiával begyűjtött receptadatbázist. Ehhez ki kellett választanom a megvalósítás nyelvét és keretrendszerét, majd kiválasztott eszközök felhasználásával weboldalt kellett készítenem, ami megfelelően ellátta a megfogalmazott feladatokat és felhasználóbarát környezetet nyújt.

Az elkészített oldal modern, letisztult érzetet áraszt, könnyen operálható. A receptek felsorolása átlátható. Habár a képek egybe vannak vonva a szöveggel, a szöveg nem elvesz a képekből, hanem fókuszt helyez a vibráns színekre, kiemeli őket.

A feladat elkészítéséhez a Python és a Django jó választásnak bizonyult. A Pycharm IDE könnyen használható környezetet nyújtott, amit magamtól is fel tudtam fedezni, illetve bármilyen szükséges csomaghoz könnyű telepítést biztosított.

A Python nyelv könnyen átlátható és olvasható volt a számomra, könnyebben olvashatónak éreztem a C# nyelvnél, ezért elégett vagyok a döntésemmel. Mind a Django-ban, mind a web scraping esetén hatalmas segítséget nyújtott a 4. fejezetben tárgyalt nagy mennyiségű segédanyag, különösen a hivatalos dokumentációk. Ezek hasznossága az Irodalomjegyzékben megadott tételekből is látható.

A legnagyobb gondot talán az alapanyag ajánlások elkészítése jelentette, ahol sok idő volt ötletet találni a felesleges ajánlások kiszűrésére, illetve a JavaScript és aszinkron programozás használata a kevés múltbéli tapasztalatom miatt a témákban.

Összességében a legtöbb használt technika ismerős, de mégis új volt számomra, ezért a megértésük viszonylag hosszú időt vett igénybe, de úgy érzem, hogy a feladatkiírást teljesítettem, még akkor is, ha az eredmény kiegészíthető maradt.

9. SUMMARY

In my thesis I had to implement a recipe recommendation system, in which a recipe database that I collected with web scraping could be filtered by ingredients and other filters. For this, I first had to choose the language and a framework for the implementation, then, using the selected tools, I had to create a website that would adequately perform the defined tasks and provide a user-friendly environment.

The finished page has a modern, clean feel to it, and is easy to operate. The list of recipes is easy on the eyes. Despite the recipe titles being displayed on top of the pictures, they don't take away from them, instead they end up accentuating them by making their color stand out even more.

Choosing Python and Django for the project proved to be a correct choice. The Pycharm IDE offered an easy-to-use environment to navigate, that I could explore myself. It also provided easy access to all the necessary packages.

The Python language proved easier to read and write than C#, so I ended up being happy with my choice. In the case of using Django and making scripts for web scraping, the plethora of community made resources mentioned in Chapter 4 proved very useful, especially the official documentations. The usefulness of these resources is reflected my bibliography.

The biggest obstacle was probably the auto suggestions when typing ingredients into the search bar, where it took a while to come up with the correct idea to filter out all the unnecessary suggestions, on top of having to use JavaScript and async programming, while I only had little experience with both.

In summary, while most of the used technology was familiar or close to what I used before, they were still new to me. Thanks to this, understanding them took a relatively significant amount of time, but despite that, I feel that the original goals of the project were accomplished, even if the finished product remains upgradable.

10.IRODALOMJEGYZÉK

[1] L. S. Vailshery, „statista.com,” Statista, 19 január 2024. [Online]. Available: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/. [Hozzáférés dátuma: 5 március 2024].

[2] Flask dokumentáció (Verzió 3.0.X), „Design Decisions in Flask,” [Online]. Available: https://flask.palletsprojects.com/en/3.0.x/design/#what-does-micro-mean. [Hozzáférés dátuma: 5. március 2024].

[3] T. S. Jalolov, *ADVANTAGES OF DJANGO FEMWORKER,* India: International Multidisciplinary Journal for Research & Development, 2023.

[4] Django Software Foundation, „Django Documentation,” Django Software Foundation, 2005. [Online]. Available: https://docs.djangoproject.com/en/5.0/. [Hozzáférés dátuma: 17. március 2024].

[5] Python Software Foundation, „urllib — URL handling modules,” [Online]. Available: https://docs.python.org/3/library/urllib.html. [Hozzáférés dátuma: 6. március 2024].

[6] Beautiful Soup dokumentáció (4.12.0. Verzió), „BeautifulSoup Documentation,” [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/#calling-a-tag-is-like-calling-find-all. [Hozzáférés dátuma: 5 március 2024].

[7] SQLite Consortium, „SQLite Home Page,” SQLite Consortium, [Online]. Available: https://www.sqlite.org/. [Hozzáférés dátuma: 4. március 2024].

[8] Django Software Foundation, „Models,” [Online]. Available: https://docs.djangoproject.com/en/5.0/topics/db/models/. [Hozzáférés dátuma: 11. március 2024].

[9] Django Software Foundation, „QuerySet API reference,” [Online]. Available: https://docs.djangoproject.com/en/5.0/ref/models/querysets/. [Hozzáférés dátuma: 5. március 2024].

[10] W3Schools, „Django Views,” [Online]. Available: https://www.w3schools.com/django/django_views.php. [Hozzáférés dátuma: 6. március 2024].

[11] Django Software Foundation, „The Django template language,” Django Software Foundation, [Online]. Available: https://docs.djangoproject.com/en/5.0/ref/templates/language/. [Hozzáférés dátuma: 16. március 2024].

[12] Python Software Foundation, „Simple HTML and XHTML parser,” [Online]. Available: https://docs.python.org/3/library/html.parser.html. [Hozzáférés dátuma: 8. március 2024].

[13] Django Software Foundation, „Django settings,” [Online]. Available: https://docs.djangoproject.com/en/5.0/topics/settings/. [Hozzáférés dátuma: 7. március 2024].

[14] Django Software Foundation, „django-admin and manage.py,” [Online]. Available: https://docs.djangoproject.com/en/5.0/ref/django-admin/#django-admin-inspectdb. [Hozzáférés dátuma: 10. Március 2024].

[15] Django Software Foundation, „Migrations,” [Online]. Available: https://docs.djangoproject.com/en/5.0/topics/migrations/. [Hozzáférés dátuma: 7. március 2024].

[16] Mozilla Corporation, „Django Tutorial Part 5: Creating our home page,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Home_page#overview. [Hozzáférés dátuma: 6. Március 2024].

[17] Mozilla Corporation, „Django Tutorial Part 6: Generic list and detail views,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Generic_views. [Hozzáférés dátuma: 6. március 2024].

[18] Django Software Foundation, „URL dispatcher,” [Online]. Available: https://docs.djangoproject.com/en/5.0/topics/http/urls/. [Hozzáférés dátuma: 7. Március 2024].

[19] C. Kolade, „With Open in Python – With Statement Syntax Example,” [Online]. Available: https://www.freecodecamp.org/news/with-open-in-python-with-statement-syntax-example/. [Hozzáférés dátuma: 21. március 2024].

[20] H. C. Chua, „Regular Expressions (Regex),” [Online]. Available: https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html. [Hozzáférés dátuma: 13. március 2024].

[21] W3Schools, „How To Create A Collapsible,” [Online]. Available: https://www.w3schools.com/howto/howto_js_collapsible.asp. [Hozzáférés dátuma: 2 Április 2024].

[22] Mozilla Corporation, „Window: localStorage property,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage. [Hozzáférés dátuma: 12. április 2024].

[23] W3Schools, „CSS Animations,” [Online]. Available: https://www.w3schools.com/css/css3_animations.asp. [Hozzáférés dátuma: 28 március 2024].

[24] W3Schools, „Python List extend() Method,” [Online]. Available: https://www.w3schools.com/python/ref_list_extend.asp. [Hozzáférés dátuma: 10 április 2024].

[25] S. András, „oktatas:web:javascript:javascript_async [szit],” [Online]. Available: https://szit.hu/doku.php?id=oktatas:web:javascript:javascript_async. [Hozzáférés dátuma: 10. Április 2024].

[26] Mozilla Corporation, „Using the Fetch API - Web APIs,” Mozilla Corporation , [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. [Hozzáférés dátuma: 9. április 2024].

[27] G.-M. Hector, D. U. Jeffrey és W. Jennifer, Adatbázisrendszerek megvalósítása, Budapest: Panem, 2001.

11.ÁBRAJEGYZÉK