ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

KANDÓ KÁLMÁN VILLAMOSMÉRNÖKI KAR
ELEKTRONIKAI ÉS KOMMUNIKÁCIÓS RENDSZEREK INTÉZET
MŰSZERTECHNIKAI ÉS AUTOMATIZÁLÁSI TANSZÉK

SZAKDOLGOZAT

OE-KVK
2023.

Zombori Patrik
T009537/FI12904/K

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Óbudai Egyetem
Kandó Kálmán Villamosmérnöki Kar
Műszertechnikai és Automatizálási Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: Zombori Patrik
Szakdolgozat száma: SZD2309040923437224[redacted]
Törzskönyvi száma: T009537/FI12904/K **Neptun kódja:** [redacted]

Szak: villamosmérnöki (magyar nyelven)
Specializáció: Műszer-automatika specializáció

A dolgozat címe: Intelligens bevásárlókocsi
A dolgozat címe angolul: Smart shopping cart
A feladat részletezése: Készítsen intelligens bevásárlókocsit, amely egy eladótérben elhelyezett áruk azonosítását követően a vásárlás adatait tárolja és megjeleníti, illetve felhasználói felületet biztosít a forgalom követéséhez!

Intézményi konzulens neve: Sándor Tamás

A kiadott téma elévülési határideje:2025. 12. 31.
Beadási határidő: 2023. 12. 15.

A szakdolgozat: Nem titkos.

Kiadva: Budapest, 2023. 10. 24. P.H.

Óbudai Egyetem Kandó Kálmán Villamosmérnöki Kar

.....
Intézetigazgató

A dolgozatot beadásra alkalmasnak találom:

.....
belső konzulens

.....
külső konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

KANDÓ KÁLMÁN VILLAMOSMÉRNÖKI KAR
ELEKTRONIKAI ÉS KOMMUNIKÁCIÓS RENDSZEREK INTÉZET
MŰSZERTECHNIKAI ÉS AUTOMATIZÁLÁSI TANSZÉK

HALLGATÓI NYILATKOZAT

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

Budapest, 2023.12.06.

.....
hallgató aláírása

Tartalom

1. Bevezetés

A szakdolgozatom témája egy intelligens bevásárlókocsi tervének elkészítése, amellyel már foglalkoztunk korábban Projektmunka 1, illetve Projektmunka 2 keretein belül. A teljes rendszer célja, hogy minden ember számára könnyebbé és kényelmesebbé tegye a hétévégi bevásárlásunkat legfőképpen egy élelmiszer áruházban. Ebben a projektben ketten vettünk részt az egyik szaktársammal, mert mind a kettőnk érdeklődését felkeltette ez a téma. A feladat megosztás szempontjából projekttársam az azonosítás és különböző azonosítási technológiáért volt felelős, még az én részem ebben a témában maga a szoftveres résznek megírása, majd esetleges implementálása. Néhány félévvel ezelőtt csak az ötlet fogalmazódott még csak meg, de ahogy haladtunk előre az időben egyre jobban kezdtünk elmélyülni a témában és feltérképezni azokat az elemeket, amire szüksége lenne egy felhasználónak a vásárlása során.

2. Motiváció

Az általunk választott téma egy intelligens bevásárlókocsi tervének elkészítése. Mivel minden ember valamilyen szinten jár vásárolni ezért úgy éreztük ez egy fontos ügy. Napjainkban a legtöbb embernek nem a legkedvencebb időtöltése a hétvégi nagybevásárlás vagy bármilyen amblokk bevásárlás, sok embert zavar a tömeg, az elvesztegetett idő mikor valamit terméket keresünk amblokk, mert nem vagyunk elég jártasok, vagy éppen most rendezték át az áruházat és bolyongunk, ezenkívül még rengeteg dolog miatt lehet bosszankodni még talán azt tudnám kiemelni mikor nem megtervezetten megyünk vásárolni nincs egy bevásárló listánk vagy rengeteg időt vesz el maga az az idő amit a kasszánál töltünk. Ezek között a felsorolt tények között rengeteg személyes tapasztalatom is van amire megoldást szeretnék kapni, hogy gördülékenyebben és effektíven történjen a bevásárlás. Nagyban motivált az is, hogy adatokat kell feldolgozni adatbázis segítségével, amelyet később pedig erre a célra megfelelő webes felületen kell megjeleníteni. Fontos szempont volt, hogy az azonosítási technikákat is megismerjük ugyanis ugyan is ez szinte ott van a mindennapjainkban, illetve az, hogy ezeket mégis hogyan miképpen válasszuk a mi specifikációnk alapján, nagyobb rálátást szerettünk volna elérni, valamint nekem személyesen volt a munkám során egy ilyen vagy ehhez hasonló projektem, amelyet nagyon élveztem. A projekt társammal készítettünk egy kérdőívet, ami nagyon fontos volt a projekt későbbi részében is, illetve az alapján indultunk el határoztuk meg, hogy mégis milyen problémát/problémákat okoznak az emberek életében ezek a vásárlások, illetve különböző vásárlási szokásoknak a felismerése is fontos volt, ami alapján mi is eltudunk indulni a projektünkkel kapcsolatban. Ezt a kérdőívet egy közösségi média platformon tettük fel (Facebook), ahol is rengeteg kitöltést kaptunk rövid idő alatt, ami számunkra egy nagyon pozitív visszajelzés volt ugyanis majd, hogy nem 300-an töltötték ki, itt már végleg láttuk azt, hogy az embereket miképpen is zavarja a vásárlás és miket kellene implementálnunk a projektünkben, ezt az ötletet a Témavezetőnk ajánlotta, ami szerintünk igen nagy sikert aratott a projektünk szempontjából. Illetve az utolsó motiváló szempontunk pedig az volt, hogy az ilyen fajta azonosítási technológia csak kevés üzleteben elérhető, és amennyiben elérhető akkor nem sokkal könnyíti meg a vásárlásunkat, vagy RFID-s azonosítási technológiát alkalmaz. Ezeket az azonosítási technológiákat másképpen gondoltuk alkalmazni első sorban a Projektmunka, illetve későbbiekben a Szakdolgozat tekintetében élelmiszer áruházakat/üzleteket vettünk górcső alá, ami minden embert érintő téma. Így úgy gondoljuk a téma mindenképpen megérhet egy befektetést valamilyen cégnek, nem feltétlenül a jelenlegi

állapotában, hanem mikor már konkrét fizikai megvalósítással rendelkezik. Ezt alátámasszák a kérdőív statisztikái.

3. Már létező megvalósítások

Az interneten tett kutatásaim során különböző intelligens bevásárlókocsi megoldásokba vizsgáltam meg, hogy megismerjem a piacon jelenlévő különféle implementációkat. Az egyik legismertebb és talán első ilyen projektet az Amazonhoz köthetjük, ahol a felhasználó telefonos bejelentkezést követően kezdi meg vásárlását. A termékek beolvasása és mérése után, elkerülve a kasszáknál kialakuló tömeget, a vásárló a terminálon keresztül is kifizetheti a kiválasztott árucikkeket. [1]

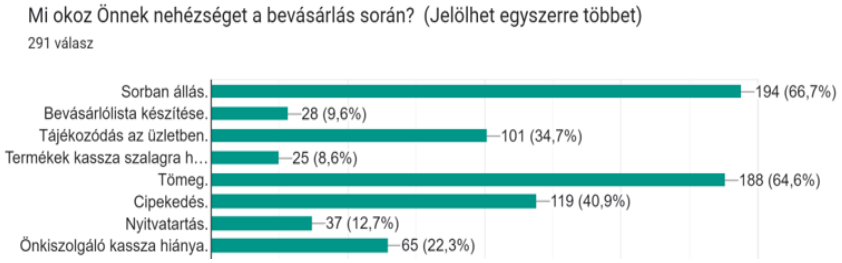
A CUST2MATE egy érintőképernyős változatot kínál, mely LFT (Legal for Trade)[2] súlymérő rendszerrel van felszerelve. Ezen kívül a kocsiban található vonalkód olvasó segítségével azonosítja a kiválasztott termékeket, melyeket a mérlegre helyezve megkönnyíti a fizetési folyamatot. Az intelligens bevásárlókocsik ezen rendszere hatékonyan gátolja a lopásokat, mivel a különböző kiszerelésű termékek cseréje így elkerülhető. Emellett a rendszer gyorsaságát az adja, hogy a termékeket nem kell újra kipakolni a kassza szalagra, mivel minden kocsi rendelkezik saját terminállal. Ez lehetővé teszi a fizetést az üzlet területén bárhol, elkerülve ezzel a tömeget és a sorban állást. Az interaktív felhasználói felület gördülékeny és könnyen kezelhető, továbbá a célzott marketing funkciója a felhasználó szokásait figyelembe véve optimális ajánlatokat és akciókat kínál.[3]

A New York-i székhelyű Caper Cart is érdemel figyelmet, mely már a harmadik verzióját hozta piacra az intelligens bevásárlókocsik közül. Az AI segítségével automatikusan felismeri a kosárba helyezett termékeket, és QR kód szkennelésnél a felhasználó könnyen készíthet bevásárlólistát az applikációban. Az integrált termék keresés funkció időt és energiát spórol a felhasználónak. [4]

Az Amazon Dash Cart egy okos bevásárlókocsi, amely a vásárlás gyorsítására és a vásárlói élmény javítására szolgál. Érzékelők és kamerák segítségével azonosítja a kosárban lévő termékeket, lehetővé téve a vásárlók számára a gyors és kényelmes vásárlást. Az automatikus fizetési rendszer az Amazon-fiókból vonja le a vásárlás összegét, míg a kijelzőn ellenőrizhető a kosár tartalma, és személyre szabott ajánlatok jelennek meg. Ezáltal elkerülhető a hagyományos kasszákhoz való fordulás és minimalizálható a várakozási idő. [5]

4. Kérdőív

Miután meghatároztuk a motivációt és alaposan áttanulmányoztuk az elérhető információkat, különböző szakirodalomból, szükségessé vált a következő lépés, a részletes kérdőív kidolgozása. Ezt a kérdőívet a Google Forms platformon hoztuk létre, odafigyelve arra, hogy minden kérdés precízen tükrözze az általunk elképzelt specifikációkat. A kérdőív célja az volt, hogy részletes visszajelzéseket gyűjtsünk a felhasználók preferenciáiról, különös figyelmet szentelve azokra az aspektusokra, amelyek a kitöltők számára kiemelkedően fontosok vagy kevésbé lényegesek. A kérdések megfogalmazásában hangsúlyt fektettünk arra, hogy feltérképezzük a különböző vásárlási szokásokat, és a kitöltők számára lehetőséget biztosítsunk arra, hogy olyan gondolatokkal, ötletekkel álljanak elő, amelyekre mi esetleg nem gondoltunk volna. Ezáltal a kitöltők bevonása a projektünkbe dinamikusabbá és sokszínűbbé vált. A kérdőív összeállításánál kiemelten fontos volt számunkra, hogy minél több és sokrétű korcsoportból származó válaszokat kapjunk, hogy az eredmények reprezentatívak legyenek. A kérdőív kitöltése időhatékony volt, mindössze 5-7 perc alatt teljesíthető, és a közösségi média erejét kihasználva arra törekedtünk, hogy minél szélesebb körben elérjük a kitöltőket. Egy héten belül, ami meglehetősen rövid idő, 286 kitöltés érkezett, így a kérdőívünk nemcsak sikeresnek mondható, hanem motiváló hatása is érzékelhető volt, hiszen láthatóan az embereket valósan érdekelte a téma.[6]



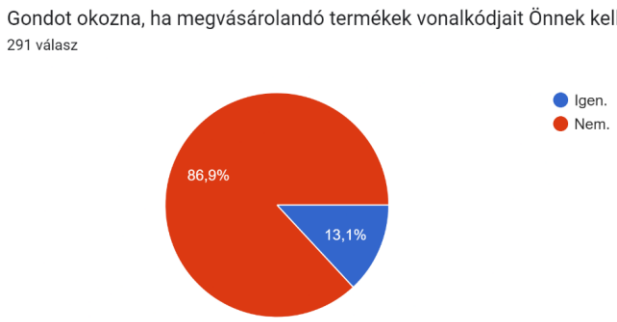
1. ábra Mi okoz problémát a vásárlás folyamán? (saját ábra)

Azon gondolkodtunk, hogy a sorban állás problémáját saját magunk oldjuk meg a vásárlás folyamán, ahol a termékeket saját vonalkód olvasónkkal becsipogjuk, hasonlóan ahhoz, ahogy egy nagy áruházban az önkiszolgáló kasszánál tennénk. Ennek előnye, hogy lényegében csak el kellene jutnunk a terminálunkhoz és kifizetni a bevásárló kocsink tartalmát, jelentősen megkönnyítve ezzel a sorban állást. Emellett az is megfontolásra került, hogy így kevesebb

ember tartózkodna az áruházban, legalábbis a kasszák területén, ami a vásárlók és az alkalmazottak biztonságát is szolgálná. Az ilyen önbevásárló megoldások lehetőséget teremtenek a hatékonyabb és gyorsabb tranzakciókra, minimalizálva a vásárlói feszültséget és növelve az áruházélmény kényelmét. A sorban állás elkerülése mellett a rendszerünknek az is célja, hogy optimalizálja az áruház belső területeit, csökkentve a zsúfoltságot és javítva az ügyfélszolgálati élményt.

2. ábra Önkiszolgáló kassza használata (saját ábra)

Ahogy a diagram is szemlélteti, meglehetősen sokan veszik igénybe az önkiszolgáló kassza lehetőségét, ami projektünk szempontjából igen kedvező jel. Ez adataink alapján arra utal, hogy a vásárlók jelentős része számára már rutinszerűvé vált az ilyen típusú (vonalkód olvasó) eszközök használata. A magas önkiszolgáló kassza-felhasználási arány azt sugallja, hogy a vásárlók kényelmesen és hatékonyan alkalmazzák ezeket az eszközöket a vásárlási folyamat során. Ez a megfigyelés lehetővé teszi számunkra, hogy a projekt szempontjából egyre biztosabbak legyünk abban, hogy az általunk gondolt azonosítási lehetőségek alapján helyesen döntöttünk.

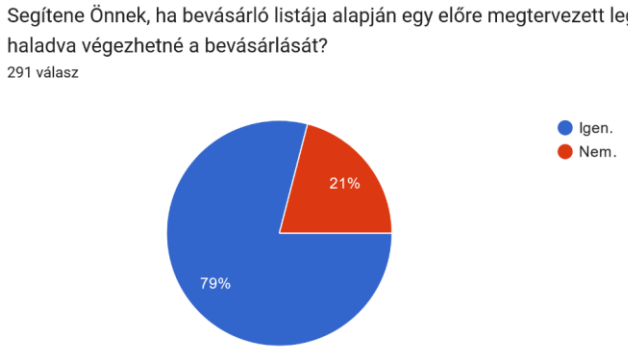


3. ábra Vonalkód beolvasásának felmérése (saját ábra)

Fontosnak tartottuk, hogy a válaszadók hajlandóságát is tekintetbe vegyük, mivel ennek függvényében kell döntenünk arról, hogy a projektünkben milyen azonosítási technológiát alkalmazzunk. Amennyiben a válaszadók nem kívánják használni a termékek vonalkódjainak beolvasását, más azonosítási módszerekre kell támaszkodnunk a probléma megoldása érdekében. Szerencsére azonban a fenti grafikon jól mutatja, hogy a válaszadók túlnyomó többségének nem jelent problémát a gyorsabb és kényelmesebb fizetésért való választás, ami azt sugallja, hogy a felhasználók körében kifejezetten pozitív a fogadtatás az ilyen technológiákkal szemben. Ennek fényében terveinket a vonalkódolvasásra alapozva tudjuk továbbfejleszteni, szem előtt tartva a felhasználói igényeket és kényelmet.

4. ábra Nyomon követési funkció felmérése (saját ábra)

A visszajelzések alapján világossá vált, hogy kiemelkedő igény mutatkozik ezen funkció iránt (kosár nyomon követése). A keresőblokk felépítése és használata ugyanis számos olyan szituációt céloz meg, amikor könnyen előfordulhat, hogy felelőtlenül vásárlunk, és csak a pénztárnál szembesülünk a végösszeggel. A beépített ellenőrzés segítségével könnyen monitorozhatjuk bevásárlólistánkat, és pontosan követhetjük, hány terméket helyeztünk a kosarunkba. Ennek eredményeképpen jelentősen csökkenne az esélye annak, hogy valami a boltok polcain maradjon, hozzájárulva a tudatosabb vásárlási szokások kialakításához. Ezen túlmenően, kiemelendő, hogy az ilyen jellegű funkció segít mélyebben megérteni az esetleges vásárlási szokásokat, és így tovább finom hangolhatjuk az alkalmazásunkat a felhasználók igényeihez.



5. ábra Legoptimálisabb útvonal funkció felmérése (saját ábra)

A többség pozitívan értékelte ezt a funkciót, mivel lehetővé teszi számunkra, hogy még részletesebben modellezzük és megértsük a vásárlási szokásokat. Ezáltal hatékonyabban tervezhetjük útvonalunkat, és az áruház is alkalmazhatja ezeket az adatokat a struktúra optimalizálásához. Az ilyen típusú információk segítségével az áruház képes átgondolni a belső elrendezést, oly módon, hogy mind a vásárlók, mind az üzlet profitáljon a változtatásokból. Ez a kölcsönös előnyös helyzet tovább erősíti az alkalmazás vonzerőjéét és a vásárlói élményt, miközben optimalizálja az áruház belső folyamatait.

5. Specifikáció

A specifikáció kialakítását kérdőív segítségével történt. A kérdőív számos fontos funkciót ölel fel, köztük a kosár kijelzését, mely dinamikus keresőfelülettel és információs gombbal rendelkezik. A rendszerben egy felhasználói azonosítási lehetőséget is implementáltam, mely tartalmaz egy regisztrációs felületet. A kosár kijelzése olyan elem, amely a felhasználó számára lehetővé teszi a kiválasztott termékek áttekintését és kezelését. A dinamikus keresőfelület segítségével a felhasználó könnyedén megtalálhatja a keresett termékeket, míg az információs gomb további részleteket nyújt a termékekkel kapcsolatban. A felhasználói azonosítás és regisztráció lehetővé tesz egyfajta „személyes fiók" létrehozását majd később pedig be tud jelentkezni ebbe a fiókba. A legrövidebb útvonal tervezése azt jelenti, hogy a felhasználók a lehető legrövidebb idő alatt és lehetőleg kevesebb lépéssel érhetik el a céljukat a rendszeren belül az alapján mit vásárol. Ez növeli az vásárlási élményt és az elégedettséget az áruházzal szemben. Végül, az adatbázis és a weboldal a rendszer alapvető részei. Az adatbázis tárolja és kezeli a rendszerben szükséges adatokat, míg a weboldal a felhasználók interakcióját teszi lehetővé a rendszerrel. Mindkét elem kulcsfontosságú a rendszer megfelelő működéséhez és a felhasználók kiszolgálásához.

6. Logikai rendszerterv

6. ábra Logikai rendszerterv (saját ábra)

Az ábra jól szemlélteti a felhasználási menetet először is regisztrálnunk kell, majd amennyiben az sikerült akkor eltárolásra kerülnek az adataink, valamint, ha nem akkor kapunk egy hibaüzenetet mi miatt nem sikerült a regisztrálási kísérlet. Utána pedig a bejelentkezés történik ugyan ilyen metódussal. A blokkok között sajnos nem tudunk lépkedni. Az elkészült blokkok tehát önálló részegységeket képeznek, amelyek nem tudnak egymással kommunikálni. A

blokkok ezen kívül megfelelően működnek tehát az Blokk 1, amely a kosár tartalmának kijelzéséért szolgál a keresési felület, aktuális összeg, illetve termék információs funkciójával. A Blokk 3, amely a legoptimálisabb útvonal megtervezéséért felelős. Majd végül pedig a 4. Blokk, amely egy adatbázissal összekötött weboldalas felület, ahol termékeket lehet feltölteni, törölni, valamint rendeléseket lehet leadni. A Blokkoknál legjobb lenne, ha tudnának egymás között kommunikálni, tehát a bejelentkezés után a kosár tartalmának kijelzésekor a egy listát hozunk létre egyfajta bevásárlási listát ahol csak bepakoljuk ezeket a termékeket majd megterveztetjük a legoptimálisabb útvonal segítségével. Lehetőség lenne az internetes megrendelésre is a weboldalas felületen.

7. Megvalósítás röviden

A megvalósításhoz a kérdőívnek megfelelő specifikációkat terveztünk megvalósítani, szétosztva a feladatokat a projekttársammal egymás között. A feladatok között az adatbázis a weboldal, a kosár tartalomkezelése, valamint a felhasználó regisztrációja és azonosítása, illetve a térképes útvonaltervezés kialakítása mind az én felelősségi körömbe tartoztak. Először is az általam vállaszott viszonylag könnyebb feladatokkal kezdtem el foglalkozni, legkönnyebbnek a kosár tartalmának való ellenőrzését megjelenítését pakolására gondoltam. Ennek a feladatnak az egyetemen tanult Informatika 2 órákon szerzett tudást használtam fel, úgy gondoltam, hogy erre a célra használt fizikai eszköz megvétele nélkül (kijelző) szoftveresen tudom majd ezt megjeleníteni a fejlesztői környezet használatával. Fejlesztői környezet tekintetében legfőképpen Pycharmot használtam, ugyanis az egyetemen is ezzel a környezettel ismerkedtünk meg. A kosár aktuális tartalmát egy listában tárolom el, ahol Tkinter importálásával jelenítem meg az általam elképzelt alakban, illetve azt is, hogy milyen adatokat szeretnék közölni, illetve kiíratni. Itt megtalálhatók ABC sorrendbe a termékek, amelyet kedvünk szerint adhatunk a kosarunkba, illetve eltávolíthatjuk egyesével, valamint a kosár tartalmának teljes törlése is elérhető. Fontosnak tartottam, hogy egy egységár oszlopot is hozzak létre, amiben lényegében a felhasználó több információt kap arról mit is vesz milyen egységárban. Mindenképpen még fontos kiemelni, ahogy a felhasználó belerak valamit a „kosarában" ezen a felületen rögtön megjelenik a termék, amit behelyezett és annak az összege is, így nem tudunk úgymond olyan felelőtlenül vásárolni, vagy gondolkodtunk még azon, hogy könnyebb lenne azoknak a felhasználóknak fizetni a kasszánál, akik még készpénzzel ugyanis előre tudják készíteni a vásárlás összegét távozás előtt. A másik ilyen blokk a felhasználó azonosítása, illetve regisztrálása. Mivel viszonylag egyszerűen lehetett elkészíteni és a célnak tökéletesen megfelelő volt emiatt újból a Pycharm fejlesztői környezetet használtam python programozási nyelvben készítettem el ezt a blokkrészt megjelenítésre ugyan úgy a szoftver fejlesztő környezet által adott lehetőséget használtam ki tehát a grafikus felület megjelenítésért a Tkinter volt az egész projekt mozgatógyűrűje. Ezt a blokkot először is egy regisztrációs felülettel kezdtem el, ahol is a felhasználónak meg kell adni az általunk meghatározott bemeneti adatokat, mint Felhasználónév, Vezeték, Keresztnév, e-mailcím, Telefonszám jelszó, valamint annak az újbóli ismétlése majd pedig utoljára a ilyen regisztrálásokkor megszokható ÁSZF-et (Adatvédelmi és Általános Szerződési Feltételek) kell elfogadni. Ezek után meg kellett határozni ennek a regisztráció sikerének előfeltételét, amely pl az ÁSZF Feltételek elfogadása, jelszavak egyezése, valamint az általunk megadott plusz feltételek, mint például a jelszó karaktereinek

[PZ1] megjegyzést írt: Ez így kell esetleg a specifikációk után?

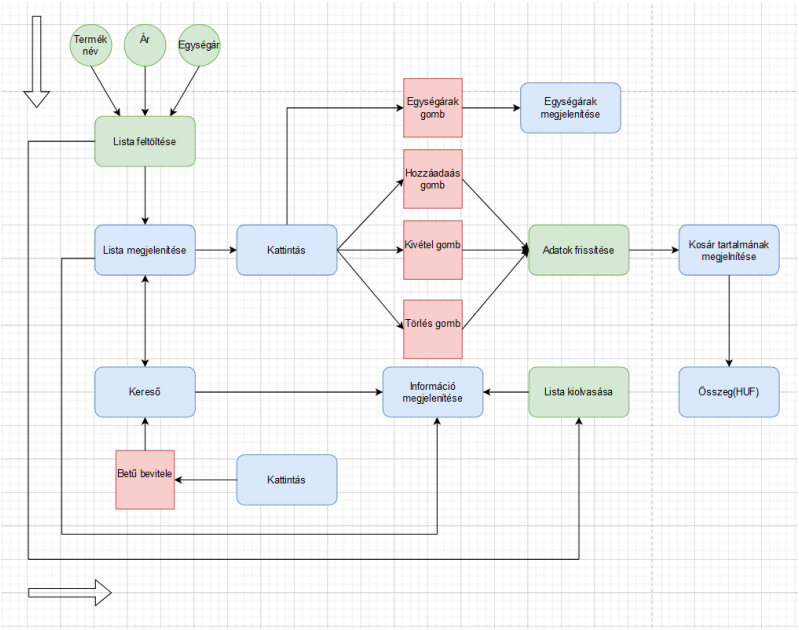
hosszúsága, illetve kötelező legalább egy nagybetű, illetve egy szám használata, amennyiben sikeres a regisztráció akkor pedig egy vissza igazoló ablak ugrik fel, amely megerősíti a felhasználót a sikeres regisztráció sikerével. Mivel fontos a felhasználónak az is, hogy azonosítani kell magát a belépés során, el kell tárolnunk a korábban regisztrációval létrehozott profilokat. Ezt egy szöveges tömbben tárolom el, amelyet a bejelentkező felület folyamatosan olvas, abban az esetben, ha ebben a szövegestömbben megtalálható a beírt felhasználónév, illetve jelszó akkor pedig sikeres a belépésünk. A harmadik blokk már egy igen nehéz blokknak nyilvánul, ebben a pontokat összekötő legegyszerűbb útvonalat választom ki olyan feltétellel, hogy az oszlopokat szimbolizáló téglalapokon keresztül ne menjen át a pontokat összekötő volna, itt ennek a megvalósításához is az eddigiekben említett fejlesztői környezetet használtam ugyan úgy. A feladatot egy kereső algoritmussal oldottam meg és ugyan úgy terveztem rá egy grafikus felületet, ezért a pontokat kézzel helyezem le, tehát folyamatosan teszteltem, hogy jól működjön ezzel az egész blokkal véleményem szerint jól sikerült reprezentálni az általunk kívánni problémát megoldással karöltve. Negyedik és egyben utolsó ilyen blokkunk a maga az adatbázisunk, amely egy weboldallal van összekapcsolódva, lényegében ez a legnehezebb legnagyobb, illetve legkomplexebb blokkunk. Az alapját az egyetemen megismert MySQL adatbázis adja meg. Ebben a MySQL adatbázisban tároljuk azokat az adatokat, amelyet meg akarunk majd később jeleníteni a weboldalunkon, tehát úgymond ebből táplálkozik. Különböző táblákat hoztam itt létre gondolva azokra az adatokra, amelyeket szeretnék majd megjeleníteni a weboldalon. A MySQL programon belül is van lehetőség ezeknek a tábláknak a feltöltésére ezt kipróbáltam, de jobban tetszett az az ötlet, hogy a Pycharm fejlesztői környezettel kötöm össze és ott használom ezt a feltöltést. Amely, mint később kiderült elengedhetetlen ugyanis a szervert ebből a Pycharm környezetből indítom el. Ebben a környezetben is mivel, ha csak egy file-t (Pycharm.py) file-t használtam volna akkor hosszú lett volna egyben a kód és valószínűleg átláthatatlan, így emiatt több fájlt készítettem az átláthatóság miatt, amit a korábban létrehozott MySQL tábláknak neveit figyelembe véve tettem, ugyanis így egyértelmű mit adok át a weboldalnak. Miután ez sikerült a szervert kellett elindítanom, amelyet később megtudom nyitni a web-en. A weboldalon hozzáadtam az összes eddig meghatározott funkciót a 4. blokkból ezután pedig teszteltem ezeket a funkciókat, mint pl: a felhasználó által leadott rendelésnek a kiíratása.

8. Szoftveres Blokkok leírása

Alapvetően a kérdőív alapján határoztuk meg azt, hogy miket is szeretnénk implementálni a szoftverünkbe milyen megvalósítások iránt van érdeklődés. A projektet átgondolva a blokkokat nehézségi sorrendbe helyeztem egymás után, úgy mintha egy adott megrendelés lenne egy külsős vevőtől vagy megrendelőtől. A projekt fejlesztése során kulcsfontosságú tényező a szisztematikus és hatékony előrehaladás, amely a felhasználói igényekre és elvárásokra összpontosít. Az elsődleges lépés a kérdőív eredményeinek részletes elemzése, melyek meghatározzák a projekt irányát és alapját képezik a további tervezésnek. Az egyes fejlesztési blokkok kialakítása előtt fontos tisztázni, hogy milyen információkat kell tudnunk azok megvalósításához. Ez a lépés pontosítja a terveket és biztosítja a fejlesztés fókuszált és hatékony haladását. A blokkok priorizálása alapvető fontosságú a projekt sikere szempontjából. A felhasználói igények és a projekt céljai alapján rendezzük el a blokkokat úgy, hogy a legfontosabb és legértékesebb funkciók kerüljenek előtérbe. Ez lehetővé teszi, hogy a rendelkezésre álló erőforrásokat a leghasznosabb módon használjuk fel. A tesztelés és hibafeltárás folyamatosan jelen van a fejlesztés során. Az elkészült blokkokat alaposan teszteljük, és az esetleges hibákat azonosítva azokat azonnal kijavítjuk. Emellett a tesztelés során felmerülő új igényekre és változtatásokra rugalmasan reagálunk, és szükség esetén plusz funkcionalitásokat adunk hozzá a projekt teljesítményének javítása érdekében. A projekt fejlesztése így egy folyamatos és rugalmas folyamat, amely az ügyfél igényeire összpontosít, miközben biztosítja a hatékony és minőségi szoftverfejlesztést. A dokumentáció készítése további támogatást nyújtanak a projektben belüli átláthatóság és hatékony együttműködés, illetve a jövőbeli újabb fejlesztéseket

8.1. 1. Blokk – Kosár kijelzés

A projektem kezdetén gondosan terveztem az alkalmazás mindenen részét, fókuszálva, azaz górcső alá véve a felhasználói élményt az átláthatóságot és az információmé jelenítését. Az első blokkban kialakítottam egy dinamikosan működő, könnyen bővíthető és elvehető terméklistát, amelyet görgetéssel is elláttam. Ez alatt a lista alatt található egy kereső felület. A kereső felületben csak betűket tudunk bevinni és az eddigi feltöltött listánk nevére tudunk szűrni a karakterek (betűk) bevitelével. A szűrés maga valós időben történik ugyanis minél közelebb járunk az adott termék beírásához gondolok itt arra, mint karakteresen tehát, ha 3 betűt leütök akkor ez a lista frissül és már csak azok a termékek látszódnak, amelyek megfelelnek ennek a bevitelnek. Fontosnak tartottam egy információs menüt is, amely akkor aktiválódik, ha a listába belekattintunk egy termékre, és arról kapjuk meg az általunk adott listában feltöltött adatot vissza, jelen esetben az egységárat a termék elhelyezkedését, illetve a termék árát. Található ezeken kívül egy „Kosár" ablak, amelyben a kosarunknak a tartalma jelenik meg valós időben frissülve. Ezt az ablakot az alatta lévő gombokkal tudjuk manipulálni. Különböző funkciók találhatóak itt, mint például a kosárba rakás, amely a listából kiválasztott terméket a kosarunkba teszi, valamint található kivétel funkció is, amely pedig a kosárba rákattintok terméket helyezi kívülre a kosarunkból. Található még egy teljes kosár törlése funkció is, illetve egy egységár gomb is, amely a termékek egység árát rakja ki bővebben. Végül pedig a kosár tartalma alatt helyezkedik el az összeg, amely minden egyes kosárba behelyezett termék után vagy kivétele után frissül és egyértelmű visszajelzést adva a felhasználó számára.



7. ábra 1. Blokk folyamatábrája (saját ábra)

Zöld háttér színnel rendelkező elemek az adatokkal foglalkozó részegységek ez az első lépés ezeknek a feltöltése pontosabban a lista feltöltése. Kék háttér színnel rendelkező részelemek a különböző interakciókat megjelenítő felületek, mint például a listának a megjelenítése, a keresőmező, valamint információ megjelenítés. Piros alapon található meg a különböző folyamatoknak az elindítása, mint a betű bevitele a kereső felületben, vagy a gomboknak a kezelésese.

```
import tkinter as tk
from tkinter import messagebox
```

8. ábra 1. Blokk implementálás (saját ábra)

Ezen sorok a **tkinter** könyvtár és annak **messagebox** moduljának importálását végzik. Ezután a **messagebox** modul segítségével könnyedén jelenítek meg olyan párbeszédpaneleket, amelyek interakcióra ösztönzik a felhasználót vagy egyszerűen tájékoztatást nyújtanak. A **tkinter** pedig a különböző grafikus felületekért felelős.

```
class UnitPricesWindow(tk.Toplevel):
    def __init__(self, products):
        super().__init__()
        self.title("Egységárak")
        self.geometry("300x400")

        self.unit_prices_listbox = tk.Listbox(self, selectmode=tk.SINGLE, height=20, width=30)
        for product, info in products.items():
            self.unit_prices_listbox.insert(tk.END, f"{product} - {info['price']} HUF/{info['unit']}")
        self.unit_prices_listbox.pack(pady=10)
```

9. ábra 1. Blokk termék lista ablak (saját ábra)

Ez a kódrészlet egy Tkinter alkalmazásban egy **Toplevel** ablakot definiál, amely az egységárakat jeleníti meg. Az ablak címe "Egységárak", mérete pedig 300x400 pixel. A **Listbox** widgetet használva a termékekhez tartozó egységárak és mértékegységek listáját jeleníti meg a felhasználónak. [7][8]

```
class ShoppingApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Shopping App")

        self.products = {
            "Tojás": {"price": 200, "unit": "db", "row": 1, "shelf": 1},
            "Liszt": {"price": 280, "unit": "kg", "row": 1, "shelf": 2},
            "Tej": {"price": 150, "unit": "l", "row": 2, "shelf": 1},
            "Pelenka": {"price": 10000, "unit": "csomag", "row": 2, "shelf": 2},
            "Cukor": {"price": 250, "unit": "kg", "row": 3, "shelf": 1},
            "Kenyér": {"price": 200, "unit": "db", "row": 3, "shelf": 2},
            "Vaj": {"price": 350, "unit": "dkg", "row": 4, "shelf": 1},
```

10. ábra 1. Blokk termékek listája (saját ábra)

Ebben a kódrészletben a **ShoppingApp** osztály **__init__** metódusában kerül sor a termékek inicializálására. A termékek adatait egy szótárban tárolom, ahol minden termék nevéhez egy másik szótár rendelődik, amely az árat mértékegységet és oszlopot, valamint polcot tartalmazza. Ez a struktúra segít rendszerezni és könnyen kezelni a termékekhez kapcsolódó információkat. Ezt követően egy listába töltöm ezeket az adatokat, így könnyen iterálhatom és hozzáférhetek a termékek adataihoz a későbbi feldolgozások során.

```
# Header
header_label = tk.Label(self.root, text="Üdvözöljük a Bevásárló alkalmazásban", font=("Helvetica", 16), pady=10)
header_label.pack()
```

11. ábra 1. Blokk header lehelyezése (saját ábra)

Ez a kódrészlet egy üdvözlő címkét hoz létre az alkalmazás fő ablakában. A címke szövege "**Üdvözöljük a Bevásárló alkalmazásban**", a betűtípusa "Helvetica", mérete 16, és 10 pixel magas **padding**-gel rendelkezik. Ezután a **pack** metódus segítségével az elrendezési rendszer automatikusan elhelyezi az ablakban, az alapértelmezett felső részen, így a felhasználó azonnal láthatja az üdvözlő szöveget az általam esztétikusan elhelyezett pozícióban.

```
    # Main Frame
    main_frame = tk.Frame(self.root)
    main_frame.pack(pady=10)
```

12. ábra 1. Blokk main_frame elhelyezése (saját ábra)

A **tk.Frame** a Tkinter modul egy keretosztályát reprezentálja, amely lehetővé teszi a widgetek csoportosítását és rendezését az ablakban. Az **pady=10** paraméter beállítja a felső és alsó külső **padding** értékét 10 pixelre. Ennek eredményeként az elrendezési rendszer az ablak felső részén helyezi el a tartalmi keretet, ami 10 pixel magas térközzel van elválasztva a felette elhelyezkedő widgetektől.

```
# Product List
product_frame = tk.Frame(main_frame)
product_frame.grid(row=0, column=0, padx=10, sticky="w")
```

13. ábra 1. Blokk product_frame elhelyezése (saját ábra)

Létrehozok egy új keretet (**Frame** objektumot), amely a fő kerethez (**main_frame**) tartozik. A **grid** metódus segítségével elhelyezi a keretet az ablak rácsozatában az első sorban, az első oszlopban, 10 pixeles belső térközzel és balra igazítva.

```
product_label = tk.Label(product_frame, text="Termékek", font=("Helvetica", 12))
```

14. ábra 1. Blokk product_label (saját ábra)

Létrehozok egy címkét ("Termékek") a termékek keretében. A címke betűtípusa "Helvetica", mérete pedig 12. A **grid** metódussal elhelyezi a címkét.

```
self.product_listbox = tk.Listbox(product_frame, selectmode=tk.MULTIPLE, height=10, width=40)
for product, info in self.products.items():

    self.product_listbox.insert(tk.END, f"{product} - {info['unit']}")
self.product_listbox.grid(row=1, column=0, padx=10)
```

15. ábra 1. Blokk listabox termékek megjelenítése (saját ábra)

Megvalósítok egy listboxot, amely a termékek megjelenítésére szolgál a termékek keretében. A listbox lehetővé teszi, hogy több terméket is kiválasszon egyszerre. A magassága 10, ami azt jelenti, hogy egyszerre tíz termék látható a listában, a szélessége pedig 40 karakter, ami elegendő helyet biztosít a termékek megjelenítéséhez. A listába való elemek hozzáadása során egy **for** ciklus segítségével végigmegy a termékek szótáron. Minden terméket a listboxhoz ad hozzá a következő formátumban: **"{termék neve} - {mértékegység}"**. Ez biztosítja, hogy a listboxban minden termék megjelenjen a nevével és a hozzátartozó mértékegységgel.

```
# Keresés
search_frame = tk.Frame(self.root)
search_frame.pack(pady=15)

cart_label = tk.Label(search_frame, text="Keresés", font=("Helvetica", 12))
cart_label.grid(row=0, column=0, padx=0, sticky="w")
```

16. ábra 1. Blokk search_frame elhelyezése (saját ábra)

Létrehozok egy új keretet, **search_frame** néven, az alkalmazás főablakában (**self.root**). Ezt a keretet a **pack** módszerrel helyezem el az ablakban, 15 pixeles felső és alsó térközzel. Ezután létrehozok egy címkét a **search_frame** kereten belül, **cart_label** néven. A címke szövege "Keresés", betűtípusa "Helvetica", mérete pedig 12 pont.

```
self.search_var = tk.StringVar()
self.search_var.trace_add("write", self.search_changed)
self.search_entry = tk.Entry(product_frame, width=30, textvariable=self.search_var)
self.search_entry.grid(row=8, column=10, padx=10, sticky="w")
```

17. ábra 1. Blokk search_frame funkció (saját ábra)

Először egy szöveg variable-t hoz létre, **self.search_var** néven, majd hozzárendel egy eseményfigyelőt (**trace_add**), amely akkor aktiválódik, amikor a változó értéke módosul.

Ebben az esetben a **self.search_changed** metódust hívja meg, amely egy keresést indít. Ezután létrehozok egy beviteli mezőt, **self.search_entry** néven, amelynek a szélessége 30 karakter, és a szöveg variable a **self.search_var**. A beviteli mezőt a **product_frame** kereten belül helyezi el.

```
self.search_entry.bind("<KeyRelease>", self.search_entry_key_release)
```

18. ábra 1. Blokk keresési folyamat aktiválása (saját ábra)

Ez a kódsor egy eseményfigyelőt (event listener) köt hozzá a **self.search_entry** beviteli mezőhöz. Amikor a felhasználó lenyom egy billentyűt, az eseményt "KeyRelease" váltja ki. Ennek hatására a program a **self.search_entry_key_release** nevű metódust hívja meg, amely valószínűleg felelős a keresési eredmények frissítéséért vagy módosításáért az alkalmazásban.

```
product_info_frame = tk.Frame(main_frame)
product_info_frame.grid(row=0, column=1, padx=10, sticky="w")

product_info_label = tk.Label(product_info_frame, text="Termék információ", font=("Helvetica", 12))
product_info_label.grid(row=0, column=0, padx=10, sticky="w")

self.product_info_label = tk.Label(product_info_frame, text="", font=("Helvetica", 10), justify="left",
                                   wraplength=250)

self.product_info_label.grid(row=1, column=0, padx=10)

self.product_listbox.bind("<<ListboxSelect>>", self.show_product_info)
```

19. ábra 1. Blokk termék információk (saját ábra)

Ebben a kódrészletben először is létrejön egy keret (**product_info_frame**), amely a termékinformációkat fogja tartalmazni a fő kereten belül. Ebben a keretben található egy címke (**product_info_label**), amelyen szerepel a "Termék információ" szöveg. A **wraplength** paraméter beállítása 250 pixelre biztosítja, hogy a szöveg tördelése ne lépje túl ezt a szélességet. **self.product_listbox.bind("<<ListboxSelect>>", self.show_product_info)** rész pedig összeköti a kiválasztott elem eseményét a megfelelő metódussal, így a termékinformációk automatikusan frissülnek a kiválasztott termék alapján.

```
cart_frame = tk.Frame(self.root)
cart_frame.pack(pady=10)

cart_label = tk.Label(cart_frame, text="Kosár", font=("Helvetica", 12))
cart_label.grid(row=0, column=0, padx=10, sticky="w")

self.cart_listbox = tk.Listbox(cart_frame, height=10, width=40)
self.cart_listbox.grid(row=1, column=0, padx=10)
```

20. ábra 1. Blokk kosár felirat elhelyezése (saját ábra)

Ebben a függvényben először egy új keretet (**cart_frame**) hozok létre, mely a kosárhoz kapcsolódó elemeket fogja tartalmazni. A **cart_frame**-et az ablakban (**self.root**) a **pack** módszerrel pozícionálják, 10 pixeles térközzel az alatta lévő widgetektől. Ezután készítnek egy címkét (**cart_label**), mely a "Kosár" szöveget tartalmazza, majd ezt a címkét a **cart_frame** kereten belül a **grid** módszerrel helyezem el az ablakon belüli elrendezéshez. Végül létrehozok egy listboxot (**self.cart_listbox**), mely a kosárban lévő elemeket jeleníti meg. A listbox magassága 10, szélessége pedig 40 karakter.

```
button_frame = tk.Frame(self.root)
button_frame.pack(pady=10)

add_to_cart_button = tk.Button(button_frame, text="Kosárba", command=self.add_to_cart, width=15)
add_to_cart_button.grid(row=0, column=0, padx=5)

remove_from_cart_button = tk.Button(button_frame, text="Kosárból kivesz", command=self.remove_from_cart,
                                    width=15)
remove_from_cart_button.grid(row=0, column=1, padx=5)

clear_cart_button = tk.Button(button_frame, text="Kosár törlése", command=self.clear_cart, width=15)
clear_cart_button.grid(row=0, column=2, padx=5)

# Unit Prices Button
unit_prices_button = tk.Button(button_frame, text="Egységárak", command=self.show_unit_prices, width=15)
unit_prices_button.grid(row=0, column=3, padx=5)
```

21. ábra 1. Blokk gombok elhelyezése és funkciók hozzáadása (saját ábra)

Ebben a kódrészletben először egy új keretet (**button_frame**) hozok létre, mely gombokhoz kapcsolódó elemeket tartalmaz. A keret négy különböző funkciót implementál, mindegyik gomb szélessége pedig 15 karakter. A gombokat a **grid** módszerrel pozícionálják az ablakon. A "Kosárba" gomb (**add_to_cart_button**) létrehozása: A gomb szövege "Kosárba", a parancs (**command**) pedig a **self.add_to_cart** metódust hajtja végre. A "Kosárból kivesz" gomb (**remove_from_cart_button**) létrehozása: A gomb szövege "Kosárból kivesz", a parancs pedig a **self.remove_from_cart** metódust hajtja végre. A "Kosár törlése" gomb (**clear_cart_button**) létrehozása: A gomb szövege "Kosár törlése", a parancs pedig a **self.clear_cart** metódust hajtja végre. Az "Egységárak" gomb (**unit_prices_button**) létrehozása: A gomb szövege "Egységárak", a parancs (**command**) pedig a **self.show_unit_prices** metódust hajtja végre. Ez a gomb felel a "Unit Prices" (egységárak) megjelenítéséért, és a hozzárendelt parancs a **self.show_unit_prices**.

```
total_frame = tk.Frame(self.root)
total_frame.pack(pady=10)

total_label = tk.Label(total_frame, text="Összesen (HUF):", font=("Helvetica", 12))
total_label.grid(row=0, column=0, padx=10, sticky="w")

total_value = tk.Label(total_frame, textvariable=self.total_price_huf, font=("Helvetica", 12))
total_value.grid(row=0, column=1, padx=10)
```

22. ábra 1. Blokk teljes összeg elhelyezése (saját ábra)

Ebben a részben először egy "Összesen (HUF):" feliratú címkét (**total_label**) hozok létre, amelyet a **total_frame** kereten belül a **grid** módszerrel pozícionálok az ablakon belüli elrendezéshez. A címke balra van igazítva. Ezenkívül egy másik címkét (**total_value**) hozom létre, amely a **self.total_price_huf** változó értékét tartalmazza. Ez a változó a bevásárlókosárban lévő termékek összértékét jelenti forintban. A **total_value** címke a **total_frame** kereten belül a második oszlopban található, és a **grid** módszerrel kerül pozícionálásra. Ez a blokk felelős az összesített összeg megjelenítéséért, ahol a "Összesen (HUF):" feliratú címke és az aktuális összeg található.

```
def add_to_cart(self):
    selected_indices = self.product_listbox.curselection()
    for index in selected_indices:
        product_info = self.product_listbox.get(index)
        product, unit = product_info.split(" - ")
        self.selected_products.append((product, unit))

    self.update_cart()
```

23. ábra 1. Blokk kosárhoz adás funkció (saját ábra)

Ebben a függvényben először lekérem a **self.product_listbox** listboxból kiválasztott elemek indexeit, majd minden egyes kiválasztott elemhez hozzárendelem a megfelelő terméket és mértékegységet és azokhoz tartozó információkat. Ezeket az adatpárokat hozzáadja a **self.selected_products** listához. Ezután meghívja az **update_cart** metódust, amely frissíti a bevásárlókosár tartalmát az újonnan hozzáadott termékekkel.

```
def remove_from_cart(self):
    selected_indices = self.cart_listbox.curselection()
    for index in selected_indices:
        product_info = self.cart_listbox.get(index)
        product, unit = product_info.split(" - ")
        self.selected_products.remove((product, unit))

    self.update_cart()
```

24. ábra 1. Blokk kosárból kivételi funkció (saját ábra)

Ebben a funkcióban először lekérem a **self.cart_listbox** listboxból kiválasztott elemek indexeit. Majd minden kiválasztott elemhez hozzárendelem a megfelelő termék és mértékegység információkat, amelyeket a **product_info** változóban tárol. Ezeket az adatpárokat eltávolítja a **self.selected_products** listából. Végül meghívja az **update_cart** metódust, amely frissíti a bevásárlókosár tartalmát a kiválasztott elemek eltávolításával.

```
def update_cart(self):
    self.cart_listbox.delete(0, tk.END)

    for product, unit in self.selected_products:
        self.cart_listbox.insert(tk.END, f"{product} - {unit}")

    total_price = sum(self.products[product]["price"] for product, _ in self.selected_products)
    self.total_price_huf.set(f"{total_price:.2f}")
```

25. ábra 1. Blokk kosár frissítése funkció (saját ábra)

Ebben a függvényben először törlöm a **self.cart_listbox** listbox összes elemét. Ezután végigiterál a **self.selected_products** listán, és minden termék-mértékegység párost hozzáad a **self.cart_listbox** listboxhoz. Ezután kiszámolja a bevásárlókosárban lévő összegzett árat, amelyet a **total_price** változóban tárol. A **self.products** szótárból lekéri a termékekhez tartozó árakat, majd a **self.total_price_huf** változó értékét beállítja erre az összegre, formázva azt két tizedesjegyig.

```
def clear_cart(self):
    self.selected_products = []
    self.total_price_huf.set("0.00")
    self.cart_listbox.delete(0, tk.END)
    messagebox.showinfo("Kosár törölve", "A kosár tartalma törölve lett.")
```

26. ábra 1. Blokk kosár törlése funkció (saját ábra)

Ebben a definícióban először a **self.selected_products** listát üres listává állítom, a bevásárlókosár teljes tartalma törlődik. Ezután a **self.total_price_huf** változót "0.00"-ra állítva az összesített ár nullázódik. A **self.cart_listbox** listbox összes elemét törölve a grafikus felületen a kosárban szereplő termékek eltűnnek. Végül, a **messagebox.showinfo** segítségével egy információs ablak jelenik meg, értesítve a felhasználót arról, hogy a kosár tartalma sikeresen törölve lett. A cím ("Kosár törölve") és az üzenet ("A kosár tartalma törölve lett.") szövegei segítik a felhasználót az elvégzett művelet megértésében.

```
def show_product_info(self, event):
    selected_indices = self.product_listbox.curselection()
    if selected_indices:
        selected_index = selected_indices[0]
        selected_product = self.product_listbox.get(selected_index)
        product, unit = selected_product.split(" - ")
        info_text = f"Termék: {product}\nEgységár: {self.products[product]['price']}" \
                    f" HUF/{unit}\nSor: {self.products[product]['row']}\nPolc: {self.products[product]['shelf']}"
        self.product_info_label.config(text=info_text)
        self.selected_product.set(f"{product} - {self.products[product]['price']} HUF/{unit}")
```

27. ábra 1. Blokk információk kiírásának funkció (saját ábra)

Ebben a részben először lekérem a **self.product_listbox** listboxban kijelölt elemek indexeit. Amennyiben van kijelölt elem, kiválasztja az elsőt (**selected_index**), majd ennek az indexnek megfelelően lekéri a kijelölt termék adatait. Az adatokat felhasználva létrehoz egy információs szöveget (**info_text**), amely tartalmazza a termék nevét (**product**), az egységárat HUF-ban (**self.products[product]['price']**), és az egységmértéket (**unit**).

```
def show_unit_prices(self):
    unit_prices_window = UnitPricesWindow(self.products)
    unit_prices_window.transient(self.root)
    unit_prices_window.grab_set()
    self.root.wait_window(unit_prices_window)
```

28. ábra 1. Blokk adatok átadása (saját ábra)

Ebben a függvényben egy új **UnitPricesWindow** ablakot hozzok létre, amely a szükséges adatokat a **self.products** szótárból kapja. Az ablakot átmenetivé (**transient**) teszi a fő ablakhoz (**self.root**), így mindig a fő ablak felett jelenik meg. A **grab_set()** metódussal az új ablak kapja meg a fókuszt, és a **self.root.wait_window(unit_prices_window)** sorral a fő ablak addig vár, amíg az **unit_prices_window** bezáródik. Ezáltal biztosítva van, hogy a felhasználó csak az új ablakban végzett műveletek után térhet vissza a fő ablakhoz.

```
def populate_product_listbox(self):
    # Terméklista feltöltése
    self.product_listbox.delete(0, tk.END)
    for product, info in self.products.items():
        self.product_listbox.insert(tk.END, f"{product} - {info['unit']}")
```

29. ábra 1. Blokk termékek feltöltése funkció (saját ábra)

Ebben a kódrészben **self.product_listbox** listabox feltöltődik a termékek neveivel és mértékegységeivel. Először törli a listabox jelenlegi tartalmát, majd egy for ciklus segítségével

végigiterál a **self.products** szótáron. Minden termékhez hozzáad egy sort a listaboxhoz, amely tartalmazza a termék nevét és mértékegységét egy kötőjellel összekapcsolva.

```
def search_changed(self, *args):
    # Keresés funkció
    search_text = self.search_var.get().lower()
    self.populate_product_listbox()  # Mindig frissítjük az eredeti listával

    # Keresés az összes termék listájában
    for i in range(self.product_listbox.size() - 1, -1, -1):  # Visszaszámlálás a törlés miatt
        product_info = self.product_listbox.get(i)
        product, unit = product_info.split(" - ")
        if search_text not in product.lower():
            self.product_listbox.delete(i)
```

30. ábra 1. Blokk kereső ablak megváltozása funkció (saját ábra)

Ebben a függvényben azokat a termékeket szűröm le a listából, amelyek nevében nincs jelen a felhasználó által megadott keresőszöveg. A keresőszöveget kisbetűssé alakítja (**search_text**), majd az eredeti, teljes terméklistát frissíti a **self.populate_product_listbox()** hívásával. Ezt követően végigmegy az összes terméken, és azokat távolítja el a listából, amelyek neve nem tartalmazza a keresőszöveget. Ezáltal dinamikusan frissíti és szűri a terméklistát az aktuális keresőkifejezés alapján, ami segíti a felhasználót az áttekinthető és célzott kiválasztásban.

```
def search_entry_key_release(self, event):
    # Keresőmező tartalmának változásakor fut le
    self.search_changed()
```

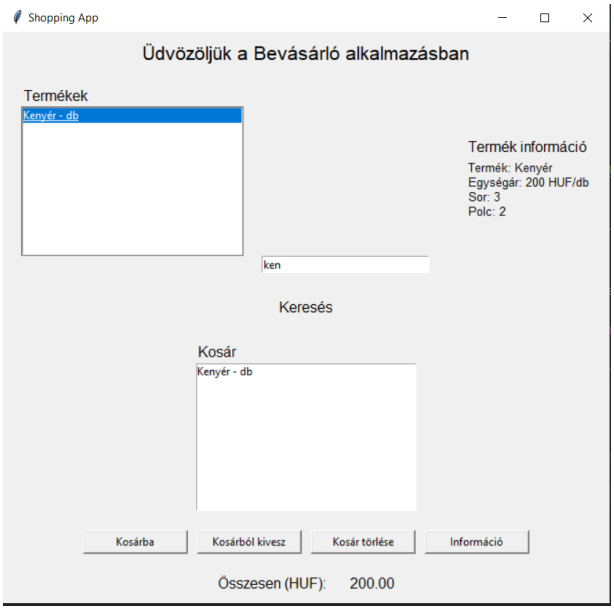
31. ábra 1. Blokk kereső ablak megváltozása 2 funkció (saját ábra)

A **search_entry_key_release** függvény akkor aktiválódik, amikor a felhasználó elengedi a billentyűt a keresőmezőben. Ezután egyszerűen meghívja a **self.search_changed()** függvényt, amely frissíti és szűri a terméklistát a keresőmező aktuális tartalma alapján.

```
if __name__ == "__main__":
    root = tk.Tk()
    app = ShoppingApp(root)
    root.mainloop()
```

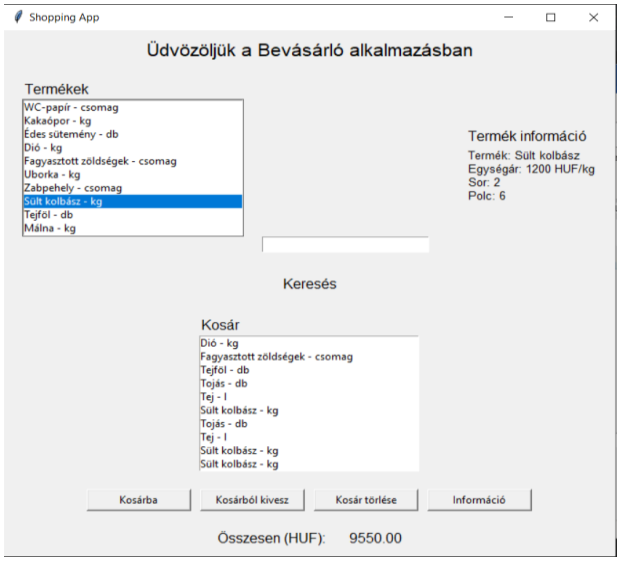
32. ábra 1. Blokk alkalmazás futtatás (saját ábra)

Végül az **if __name__ == "__main__":** rész az indító rész, amely meghatározza, hogy a scriptet közvetlenül futtatjuk-e (nem importálva más modulokból). Ebben az esetben egy új Tkinter főablakot (root) hoz létre, majd létrehoz egy **ShoppingApp** objektumot, ami inicializálja az alkalmazást. Ezután elindítja a Tkinter eseménykezelő hurkot (loop-ot) a **root.mainloop()** hívással, ami lehetővé teszi az alkalmazás futását és az események kezelését.



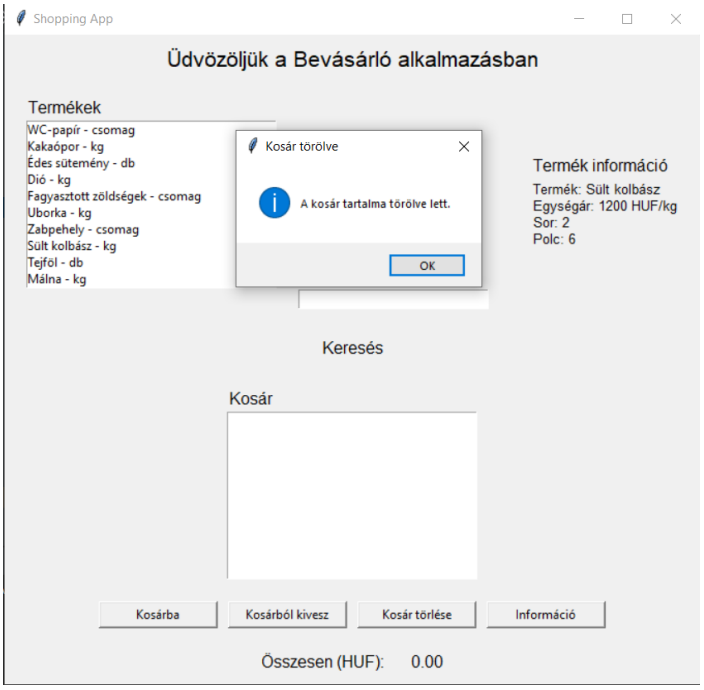
33. ábra 1. Blokk keresési funkció működés közben (saját ábra)

Ez a keresőfelület lehetőséget nyújt a működésének vizsgálatára, amely dinamikusan frissül az alapján, hogy milyen karaktert írunk be.



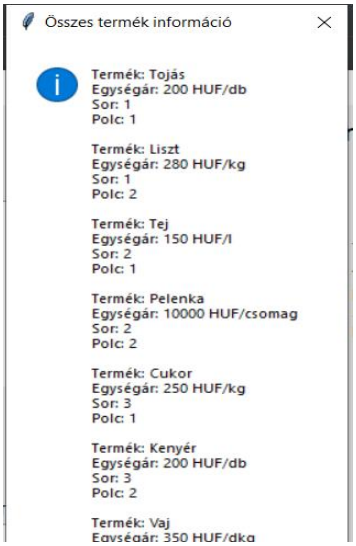
34. ábra 1. Blokk kosár tartalmának feltöltése (saját ábra)

Ezen a grafikus funkció segítségével részletesen követem és vizualizáltam a kosár tartalmának változását és kiírtatását



35. ábra 1. Blokk kosár tartalmának törlése (saját ábra)

Kosár tartalmának törlés a Kosár törlése gomb lenyomása után. A felhasználó visszajelzést kap a folyamatról.

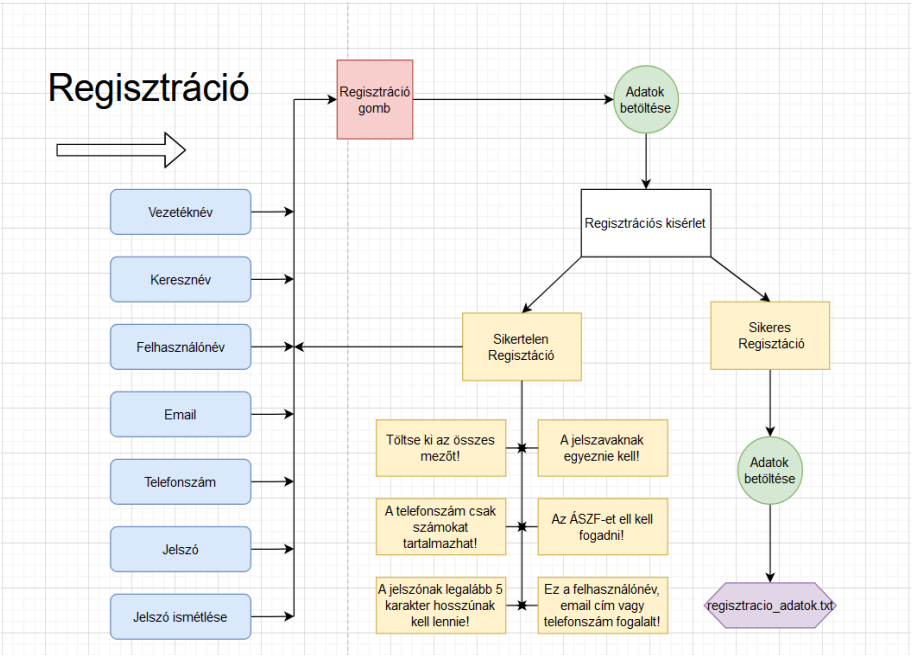


36. ábra 1. Blokk információs gomb működése (saját ábra)

Összes termék információ gomb működése, amely kilistázza az összes terméket és az azokhoz hozzárendelt információt.

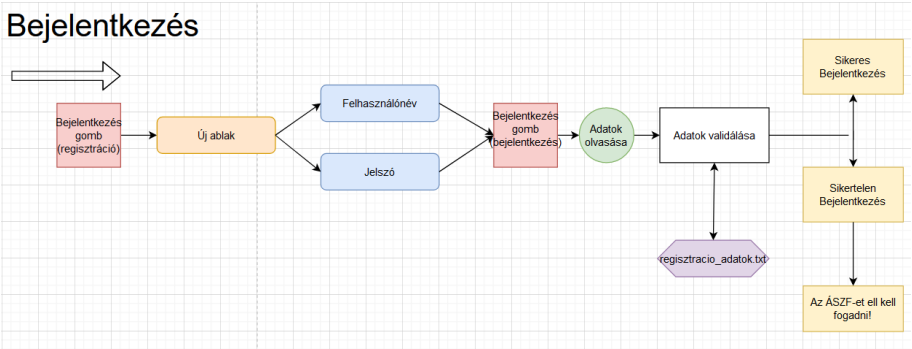
8.2. 2. Blokk – Regisztráció és bejelentkezés

A második blokkba tervezett regisztrációs folyamat során az elsődleges cél a felhasználóbarát és átlátható grafikus felület kialakítása, amely egyszerűvé teszi a regisztrációt. A szoftvernek számos szempontot kell figyelembe vennie, hogy biztosítsa a sikeres regisztrációt és megfeleljen a felhasználók elvárásainak. A jelszó helyességének, a kötelező mezők kitöltésének, valamint más, látszólag triviálisnak tűnő, de mégis kritikus részleteknek is megfelelően kell működniük, hogy kizárjuk a hibák lehetőségét a regisztrációs folyamatban. A jelszó validitása kiemelten fontos, és a rendszernek gondoskodnia kell arról, hogy a felhasználó könnyen követhesse az előírt kritériumokat. A grafikus felület feladata, hogy érthető módon mutassa be ezeket a követelményeket, és a felhasználót szükség esetén segítse a helyes adatbevitelben. A regisztrációs folyamat során a felhasználó kap visszajelzést, legyen az a regisztráció sikeres vagy sikertelen. A rendszer pontosan megjelöli, ha valamelyik adat nem felel meg a követelményeknek, és részletes tájékoztatást nyújt a hibás adatról. Ezen információk segítenek a felhasználónak a szükséges korrekciók elvégzésében. A sikeres regisztrációt követően a rendszer az összegyűjtött adatokat strukturáltan egy szöveges fájlba menti. A fájl nem csupán tároló, hanem egy könnyen értelmezhető dokumentum is, amely segíti az adatok kezelését és esetleges további felhasználását. A bejelentkezési folyamat során a rendszer visszaolvassa az előzőleg mentett adatokat egy folyamatosan frissülő szöveges fájlból. Ha a felhasználó helyesen adja meg a felhasználónevét és jelszavát, a rendszer sikeres bejelentkezésről tájékoztatja. A visszajelzés részletes és informatív, különös figyelmet fordítva az esetleges hibás adatok pontos megjelölésére. Így a felhasználó pontosan értheti, mi okozta a bejelentkezési problémát, ha ilyen bekövetkezik. Ennek révén a felhasználók könnyedén kezelhetik és javíthatják az esetleges hibákat a rendszerrel való hatékony interakció érdekében.



37. ábra 2. Blokk regisztrációs folyamatábra (saját ábra)

A folyamatábrák szemléltetik a program logikájának működését: a kékkel jelölt mezők a felhasználó által bevitt információkat, a piros hátterű rész a folyamat elindítását, a zöld hátterű rész az adatok bekérését, a sárga hátterű részek a felugró ablakokat, végül a lila hátterű rész pedig az adatok tényleges mentését jelöli.



38. ábra 2. Blokk bejelentkezési folyamatábra (saját ábra)

A piros mezők továbbra is funkcionális gombok, melyek különböző folyamatokat indítanak el, a kék hátterű mezők pedig a felhasználó által kívánt beviteli részeket jelölik. A zöld háttérrel rendelkező mezők az adatokat tartalmazzák, míg a sárga háttérrel ellátott mezők különböző megjelenítéseket jelképeznek, például felugró ablakokat a felhasználó felé.

```
import tkinter as tk
from tkinter import messagebox
import os
from datetime import datetime
```

39. ábra 2. Blokk importálások (saját ábra)

Az alkalmazás az alábbi modulokat importálja: **tkinter** a grafikus felület kezeléséhez, **messagebox** a felugró üzenetablakok kezeléséhez, **os** a fájlrendszerrel való interakcióhoz, és **datetime** az időbélyegzéshez. Ezek a modulok biztosítják az alkalmazásnak a felhasználói felület létrehozását, üzenetek kezelését, fájlrendszer-interakciót és időbélyegzést a regisztráció és bejelentkezés folyamataiban.

```
def is_valid_phone_number(phone_number):
    # Ellenőrzi, hogy a telefonszám csak számokat tartalmaz-e
    return phone_number.isdigit()
```

40. ábra 2. Blokk telefonszám ellenőrzése

Az is_valid_phone_number függvény a telefonszám validitását vizsgálja, biztosítva, hogy kizárólag számokat tartalmazzon. Ez a funkció elengedhetetlen a regisztráció során, ahol a megadott telefonszám formátumának helyessége fontos az adatok megfelelő kezelése és tárolása érdekében.

```
def show_success_message():
    messagebox.showinfo("Sikeres regisztráció", "A regisztráció sikeresen megtörtént!")

def show_error_message(message):
    messagebox.showerror("Hiba", message)
```

41. ábra 2. Blokk regisztrációs üzenet (saját ábra)

A **show_success_message** és **show_error_message** függvények olyan felugró ablakokat hoznak létre, amelyek fontosak a felhasználói élmény és visszajelzés szempontjából. Az előbbi azt a pozitív üzenetet jeleníti meg, hogy a regisztráció sikeres volt. Utóbbi pedig értesíti a

felhasználót, ha valamilyen hiba történt a regisztráció folyamata során, például hiányzó vagy helytelenül megadott adatok esetén.

```
# Ellenőrzés, hogy az adott e-mail cím, felhasználónév vagy telefonszám már szerepel-e a regisztrációs fájlban
if check_existing_data(felhasznalonev, email, telefonszam):
    show_error_message("Ez a felhasználónév, e-mail cím vagy telefonszám már használatban van!")
    return

# Regisztrációs adatok feldolgozása
print("Vezetéknév:", vezeteknev)
print("Keresztnév:", keresztnev)
print("Felhasználónév:", felhasznalonev)
print("E-mail:", email)
print("Telefonszám:", telefonszam)
print("Jelszó:", jelszo)
print("Elfogadva:", elfogadva)
```

42. ábra 2. Blokk már létező adatok vizsgálása (saját ábra)

A **check_existing_data** függvény kulcsfontosságú szerepet tölt be a regisztrációs folyamatban, mivel az ellenőrzi, hogy a felhasználónév, e-mail cím vagy telefonszám, amelyet egy új felhasználó megadott, már szerepel-e a rendszerben. Ennek a funkciónak a célja a redundancia elkerülése és a felhasználói egyediség biztosítása.

```
def save_to_file(vezeteknev, keresztnev, felhasznalonev, email, telefonszam, jelszo, elfogadva):
    # Asztal elérési útvonala
    desktop_path = os.path.join(os.path.join(os.path.expanduser('~')), 'Desktop')

    # Fájl elérési útvonala az asztalon
    file_path = os.path.join(desktop_path, "regisztracio_adatok.txt")

    # Időpont és dátum formázása
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

43. ábra 2. Blokk fájlba kimentés (saját ábra)

A **save_to_file** függvény felelős az új regisztrációs adatok rögzítéséért és tárolásáért a szöveges fájlban. Ez az eljárás hozzájárul az alkalmazás állandóan frissülő felhasználói adatbázisának karbantartásához. A funkció az új regisztrációs adatokat időbélyegzővel ellátva, strukturált formában fűzi hozzá a szöveges fájlhoz. Ez magában foglalja a felhasználó nevét (vezeték- és keresztnév), felhasználónevet, e-mail címet, telefonszámot, jelszót és az Általános Szerződési Feltételek elfogadásának állapotát. Az időbélyegző hozzáadása lehetővé teszi az adatok rögzítési időpontjának követését és visszakeresését.

```python
def get_last_id():
    # Asztal elérési útvonala
    desktop_path = os.path.join(os.path.join(os.path.expanduser('~')), 'Desktop')

    # Fájl elérési útvonala az asztalon
    file_path = os.path.join(desktop_path, "regisztracio_adatok.txt")

    # Ellenőrizze, hogy a fájl létezik-e
    if not os.path.exists(file_path):
        return 0  # Ha a fájl nem létezik, akkor az id 0

    # Olvassa be a fájlt és találja meg az utolsó id-t
    with open(file_path, "r") as file:
        lines = file.readlines()

    for line in reversed(lines):
        if line.startswith("ID: "):
            return int(line.split(":")[1].strip())  # Az utolsó id

    return 0  # Ha nem találtunk id-t a fájlban
```

44. ábra 2. Blokk regisztráció azonosítás kiolvasása (saját ábra)

A **get_last_id** függvénynek kulcsszerepe van abban, hogy visszaadja a regisztrációkhoz tartozó egyedi azonosítók közül a legutolsót a szöveges fájlban. Ennek a funkciónak a használata különösen fontos, amikor új regisztráció történik, mivel az egyedi azonosító garantálja, hogy minden regisztráció egyediségét és nyomon követhetőségét biztosítva kezelhető legyen.

```python
def submit_registration():
    # Funkció a regisztráció gomb lenyomásakor
    vezeteknev = vezeteknev_entry.get()
    keresztnev = keresztnev_entry.get()
    felhasznalonev = felhasznalonev_entry.get()
    email = email_entry.get()
    telefonszam = telefonszam_entry.get()
    jelszo = jelszo_entry.get()
    jelszo_ismet = jelszo_ismet_entry.get()
    elfogadva = elfogadva_var.get()
```

45. ábra 2. Blokk regisztrációs kísérlet gombbal (saját ábra)

A **submit_registration** függvény a regisztráció gomb lenyomásakor fut le. Ellenőrzi a mezők kitöltöttségét, a telefonszám formátumát, a jelszó hosszát és egyezőségét, valamint az ÁSZF elfogadását. Sikeres regisztráció esetén az adatokat elmenti a fájlba.

```
# Főablak létrehozása
root = tk.Tk()
root.title("Regisztráció")

# Szélesség és magasság beállítása
root.geometry("500x400")
```

46. ábra 2. Blokk regisztrációs ablak létrehozása (saját ábra)

Ebben a sorban egy **Tk()** objektum létrehozása történik, ami az ablakot reprezentálja. Ezt az objektumot a **root** változó tárolja. Ez a sor beállítja a főablak címét a "Regisztráció" szövegre. Ebben a sorban a **geometry** metódus segítségével állítjuk be a főablak méretét. A "500x400" érték azt jelenti, hogy az ablak szélessége 500 pixel, magassága pedig 400 pixel lesz

```
# Regisztráció gomb
regisztracio_gomb = tk.Button(root, text="Regisztráció", command=submit_registration, font=("Arial", 12))
regisztracio_gomb.grid(row=8, column=0, columnspan=2)
```

47. ábra 2. Blokk regisztrációs gomb elhelyezése (saját ábra)

Létrehozunk egy regisztrációs gombot, amely a **submit_registration** függvényt hajtja végre a lenyomásakor. Elhelyezését a **.grid** metódussal teszem az általam megadott (sor oszlophoz)

```
bejelentkezes_gomb = tk.Button(root, text="Bejelentkezés", command=open_login_window, font=("Arial", 12))
bejelentkezes_gomb.grid(row=9, column=0, columnspan=2)
```

48. ábra 2. Blokk bejelentkezés gomb elhelyezése (saját ábra)

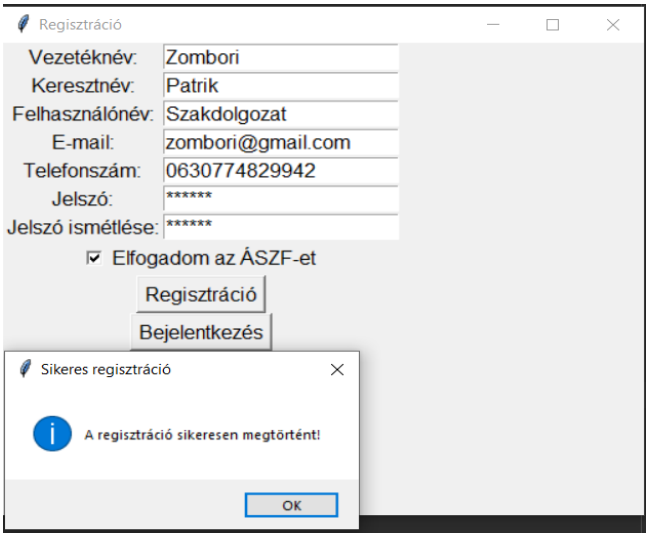
Létrehozunk egy bejelentkezési gombot is, amely egy új ablakot nyit meg a bejelentkezéshez. Elhelyezését a **.grid** metódussal teszem.

```
root.mainloop()
```

49. ábra 2. Blokk mainloop (saját ábra)

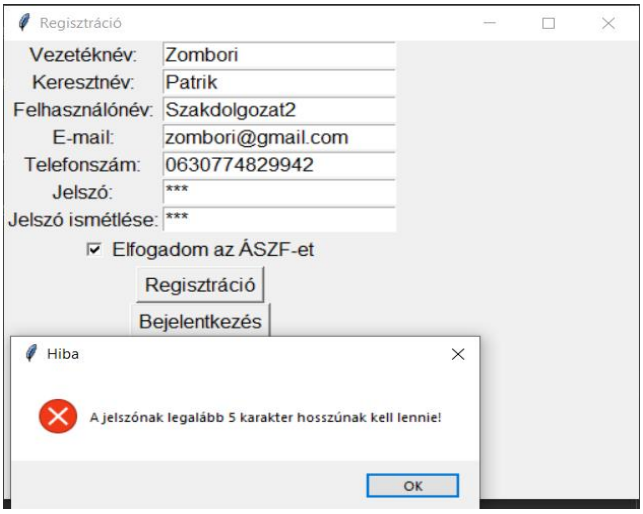
Indítjuk a Tkinter főciklusát a **mainloop** függvénnyel, hogy az alkalmazás folyamatosan figyelje a felhasználói interakciókat.

A szoftver működése az alábbi képeken látható:



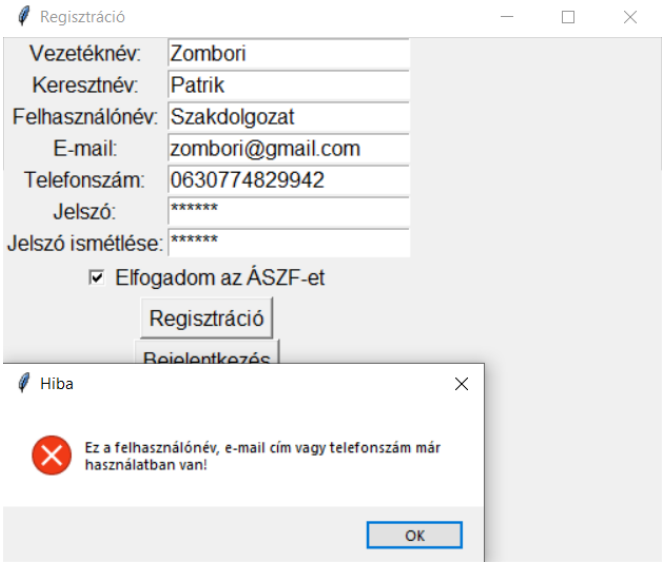
50. ábra 2. Blokk sikeres regisztráció (saját ábra)

A felhasználó megadja a feltételeknek megfelelő beviteli adatokat, tehát minden if-águnk a megfelelő értéket kapja, így nem jelenik meg hibaüzenet. A regisztráció sikeres, amit egy felugró ablak is megerősít.



51. ábra 2. Blokk sikertelen regisztráció (saját ábra)

Az ellenőrzési funkciók jelenlegi tesztelése során, ahogyan a felugró ablak is jelzi, a jelszó nem elég hosszú. Ez azt mutatja, hogy az általunk megadott feltétel, mint az "else" ág, helyesen működik, és a felhasználó egy felugró ablakon keresztül tájékoztatást kap a hibás részről.

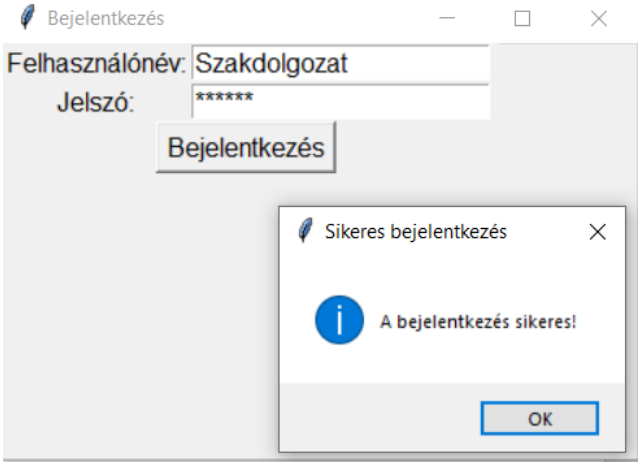


52. ábra 2. Blokk sikertelen regisztráció2 (saját ábra)

Regisztrációs ellenőrzés azonos e-mail cím telefonszám vagy felhasználónév alapján amennyiben egyezést talál a txt-ben akkor nem engedi a regisztrációs folyamatot ezt ugyan úgy mint az eddigiekben egy felugró ablakkal szemléltetem a felhasználó számára.

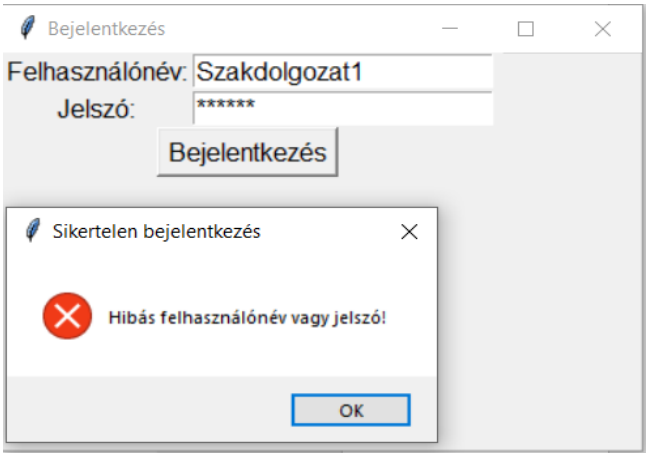
53. ábra 2. Blokk adatok fájlba írása (saját ábra)

Sikeres regisztráció utáni fájlba írás megtörténik, amelyet megnyithatunk és az átláthatóság kedvéért ID-val Dátummal és idővel valamin mínuszjelekkel választottam el, hogy ezek a funkciók jól láthatóak legyenek. Emellett pedig tudjuk ellenőrizni azt, hogy tényleg valóban elmentésre kerültek az regisztrációs adatok



54. ábra 2. Blokk sikeres bejelentkezés (saját ábra)

Fájlból való olvasásnak a sikerességének vizsgálata valóban létezett ilyen felhasználó a txt alapján és helyesen írta be az adatait. Ezt a felhasználó felé pedig egy felugró ablakkal közli a szoftver.

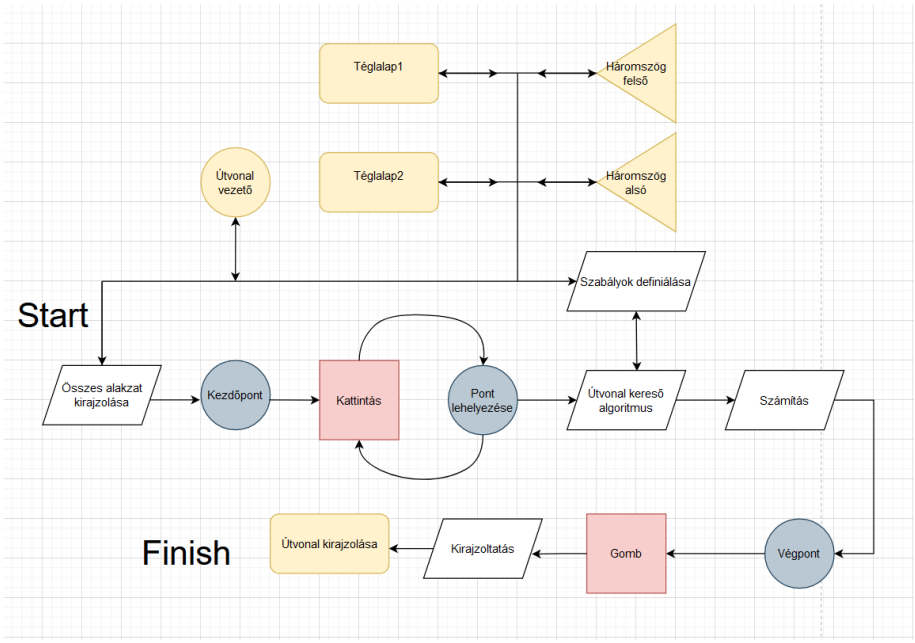


55. ábra 2. Blokk sikertelen bejelentkezés (saját ábra)

Fájlból való olvasásnak sikertelenségét láthatjuk, ami alapján nem megfelelőek azok az adatok, amelyeket bevittünk. A szoftver egy felugró ablakban közli ezt velünk.

8.3. 3. Blokk – Legoptimálisabb útvonal

A harmadik blokkban tett erőfeszítéseim középpontjában az optimális vásárlási útvonal megtervezése állt, amit egy letisztult grafikus felülettel igyekeztem könnyen érthetővé tenni. Ebben a szakaszban alkalmaztam különféle alakzatokat, például két téglalapot, melyek az áruházi sorokat szimbolizálják. Ezekhez a sorokhoz kapcsolódnak körök, amelyek egy-egy termék elhelyezkedését ábrázolják a térképen. Emellett két háromszöget is elhelyeztem, amelyek a bejáratot és a kasszát jelképezik, egyértelműen definiálva ezzel a kezdő- és végpontot. A felhasználó a térképen fekete pontokat helyezhet el, amelyek az általa kiválasztott termékeket szimbolizálják. Az alkalmazás kiszámolja az optimális útvonalat, figyelembe véve, hogy a felhasználó által elhelyezett pontok ne menjenek át a sorokat jelképező téglalapokon, illetve ezeket a pontokat ne lehessen a téglalapok területén belül elhelyezni. Ezáltal a vásárlás minden esetben a bejáratnál veszi kezdetét, a kasszánál pedig fejeződik be, a köztes útvonal pedig a felhasználó által választott termékektől függ. Ez a megoldás nem csupán egy útvonaltervező funkciót biztosít, hanem egy intuitív és látványos felhasználói felülettel segíti a vásárlót az áruházban való hatékony tájékozódásban és a kiválasztott termékek könnyű megtalálásában.



56. ábra 3. Blokk folyamatábra (saját ábra)

Sárga alapon látszódik az alapvető elemeknek a megrajzolása, mint a téglalapoké, illetve a háromszögeké emellett pedig a piros útvonal vezető köröknek a kirajzolása, valamint a kezdő és végponté is egyaránt, amely kezdőpont esetén a bejárat még végpont szempontjából pedig a kassza. Ezen útvonal megtervezéshez a felhasználó kattintással tud lehelyezni fekete pontokat ezután az összekötés gomb használatával, kirajzolódik a legoptimálisabb útvonal, figyelembe véve a szabályokat, amiket megadtunk például: nem helyezhető le a téglalapba, illetve a háromszögbe a feketepont.

```
import tkinter as tk
import heapq
import math
```

57. ábra 3. Blokk importlálás (saját ábra)

A **tkinter** egy grafikus felhasználói felület (GUI) készítésére szolgáló Python modul. A **heapq** egy olyan modul, amely különböző heap (kupac) adatszerkezeteket tartalmaz, például prioritási sorokat. A **math** modul matematikai függvényeket és konstansokat biztosít.

```
# Canvas létrehozása
self.canvas = tk.Canvas(root, width=600, height=600, bg="white")
self.canvas.pack()
```

58. ábra 3. Blokk Canvas elhelyezése (saját ábra)

Ez a rész a kód felelős egy új **Canvas** widget létrehozásáért a Tkinter GUI-ban. A **width** és **height** paraméterek meghatározzák a rajzterület méretét pixelben. **bg** paraméter az alapértelmezett háttérszínt állítja be. A **pack** metódus automatikusan elrendezési beállításokat végez.

```
# Háromszögek adatai
self.triangle_size = 60
self.gap_rect = 100

# Téglalapok adatai
self.rect_width = 100
self.rect_height = 150
self.gap = 100
```

59. ábra 3. Blokk háromszögek és téglalapok adatai (saját ábra)

self.triangle_size = 60: Ez a változó meghatározza a háromszögek méretét. Az értéke 60 pixel, ami azt jelenti, hogy a háromszögek oldala 60 pixel hosszú lesz. **self.gap_rect = 100**: Ez a változó a téglalapok közötti részt határozza meg. Az értéke 100 pixel. **self.rect_width = 100 a téglalap** szélességet adja meg még **self.rect_height = 150** a téglalap magasságát tartalmazza.

```
# Első téglalap rajzolása
self.rect1 = self.canvas.create_rectangle(
    150, 150, 150 + self.rect_width, 150 + self.rect_height, fill="blue")
# Második téglalap rajzolása a megfelelő távolságra
self.rect2 = self.canvas.create_rectangle(
    150 + self.rect_width + self.gap_rect, 150, 150 + self.rect_width * 2 + self.gap_rect,
    150 + self.rect_height, fill="red")
# Bal felső sarokban lévő háromszög rajzolása
self.triangle_left = self.canvas.create_polygon(
    50, 50, 50, 50 + self.triangle_size, 50 + self.triangle_size, 50 + self.triangle_size,
    fill="green", outline="black")
# Jobb alsó sarokban lévő háromszög rajzolása
self.triangle_right = self.canvas.create_polygon(
    600 - 50, 600 - 50, 600 - 50, 600 - 50 - self.triangle_size,
    600 - 50 - self.triangle_size, 600 - 50 - self.triangle_size,
    fill="orange", outline="black")
```

60. ábra 3. Blokk alakzatok elhelyezése (saját ábra)

Az ábrán megfigyelhető, hogy ezeket az alakzatokat a **canvesen** belül helyezem el **create_rectangle** metódussal egy téglalapot hozok létre még a **create_polygon** metódussal pedig egy sokszögeket lehet rajzoltatni ezt én a háromszög kirajzolására használtam. A **fill**-t metódussal adom meg a kitöltési színét az adott alakzatnak.

```
# Szöveg a bal felső háromszögbe
self.bejartat_text = self.canvas.create_text(
    100 + self.triangle_size / 2, 50 + self.triangle_size / 2, text="bejártat", fill="black")
# Szöveg a jobb alsó háromszögbe
self.kassza_text = self.canvas.create_text(
    600 - 100 - self.triangle_size / 2, 600 - 50 - self.triangle_size / 2, text="kassza", fill="black")

# Button to összekötés
self.connect_button = tk.Button(root, text="összekötés", command=self.connect_points)
self.connect_button.pack()
```

61. ábra 3. Blokk szövegek, illetve gomb elhelyezése (saját ábra)

Az ábrán látható módon létrehozok egy bejárat és kassza szöveget, amelyet nem messze a két háromszögtől helyezek le, ezek fogják szimbolizálni a kezdő, illetve végpontot értelemszerűen a bejárat a kezdőpontunk még a kassza pedig a végpontunk. Elhelyezésre került egy gomb összekötés névvel ellátva, amely a **connect_points** metódus hívja meg.

```
for i in range(4):
    x1 = 50 + i * 30
    y3 = 160
    y4 = 180
    y5 = 200
    y6 = 220
    y7 = 240
    y8 = 260
    y9 = 280

    self.canvas.create_oval(x1, y3, x1 + 5, y3 + 5, fill="red")
    self.red_points_coords.append((x1, y3))
    self.canvas.create_oval(x1, y4, x1 + 5, y4 + 5, fill="red")
    self.red_points_coords.append((x1, y4))
    self.canvas.create_oval(x1, y5, x1 + 5, y5 + 5, fill="red")
    self.red_points_coords.append((x1, y5))
    self.canvas.create_oval(x1, y6, x1 + 5, y6 + 5, fill="red")
    self.red_points_coords.append((x1, y6))
    self.canvas.create_oval(x1, y7, x1 + 5, y7 + 5, fill="red")
    self.red_points_coords.append((x1, y7))
    self.canvas.create_oval(x1, y8, x1 + 5, y8 + 5, fill="red")
    self.red_points_coords.append((x1, y8))
    self.canvas.create_oval(x1, y9, x1 + 5, y9 + 5, fill="red")
    self.red_points_coords.append((x1, y9))
```

62. ábra 3. Blokk vezető piros körök elhelyezése (saját ábra)

Ezen az ábrán látható a piros körök létrehozása a **create_oval** metódussal amelyet a ciklus alapján x(1) y(3,4..9)-ig elhelyezz pozíciókban végig fut egy **for** cikluson keresztül és lehelyezi ezeket a piros köröket, valamint **red_points_coords** metódus pedig a koordinátákat tárolja.

```python
def place_black_point(self, event):
    # Get coordinates of the mouse click
    x, y = event.x, event.y

    # Check if the click is far enough from existing points
    is_far_enough = all(
        ((x - px) ** 2 + (y - py) ** 2 > 25) for px, py, _, _ in
        (self.canvas.coords(point) for point in self.black_points)
    )
    # Check if the point is inside the rectangles and far enough from existing points
    if not (
            (150 <= x <= 150 + self.rect_width and 150 <= y <= 150 + self.rect_height) or
            (50 <= x <= 50 + self.triangle_size and 50 <= y <= 50 + self.triangle_size) or
            (600 - 50 - self.triangle_size <= x <= 600 - 50 and 600 - 50 - self.triangle_size <= y <= 600 - 50)
    ) and is_far_enough:
        # Create a black point and add it to the list
        black_point = self.canvas.create_oval(x - 2.5, y - 2.5, x + 2.5, y + 2.5, fill="black")
        self.black_points.append(black_point)
        # Címke a fekete pont sorszámával
        point_label = f"#{self.point_counter}"
        text = self.canvas.create_text(x + 10, y - 10, text=point_label, fill="black")
        self.black_points_text.append(text)
        # Hozzárendeli a fekete pont sorszámát a listához
        self.black_points_numbers.append(self.point_counter)
        # Növeli a fekete pontok számlálóját
        self.point_counter += 1
```

63. ábra 3. Blokk fekete pont elhelyezése (saját ábra)

Ez a függvény a **place_black_point** nevet viseli, és az egérkattintás eseményére reagál. Az **event** argumentumból kinyeri, és a **x** és **y** változókba menti. Ezek után ellenőrzöm azt, hogy a lehelyezett fekete pontok ne legyenek benne az alakzatokban (if not rész). Amennyiben megfelelnek **black_point** metódussal létrehozza a feketepontokat, és **point_labellel** #1 majd pedig **point_counterrel** minden lehelyezett fekete pontot 1-gyel növeli az átláthatóság kedvéért.

```python
def connect_points(self):
    if self.black_points and hasattr(self, 'bejartat_text'):# Connect the bejartat t
        x1, y1 = self.canvas.coords(self.bejartat_text)[:2]
        x2, y2 = self.canvas.coords(self.black_points[0])[:2]
        self.connect_along_red_points(x1, y1, x2, y2)
    # Connect black points along the path
    for i in range(len(self.black_points) - 1):
        x1, y1, _, _ = self.canvas.coords(self.black_points[i])
        x2, y2, _, _ = self.canvas.coords(self.black_points[i + 1])
        # Connect to each red point along the path
        for red_point_coords in self.red_points_coords:
            x3, y3 = red_point_coords
            if self.is_line_near_point(x1, y1, x2, y2, x3, y3):# Check if the line p
                # Connect with a straight line to the red point
                line = self.canvas.create_line(x1, y1, x3, y3, x2, y2, fill="black")
                self.lines.append(line)
                x1, y1 = x3, y3
        if i < len(self.black_points) - 2:
            x1, y1, _, _ = self.canvas.coords(self.black_points[i + 1])
            x2, y2, _, _ = self.canvas.coords(self.black_points[i + 2])
            self.connect_along_red_points(x1, y1, x2, y2)
    if self.black_points and hasattr(self, 'kassza_text'):# Connect the last black p
        x1, y1, _, _ = self.canvas.coords(self.black_points[-1])
        x2, y2 = self.canvas.coords(self.kassza_text)[:2]
        line = self.canvas.create_line(x1, y1, x2, y2, fill="black")
        self.lines.append(line)
```

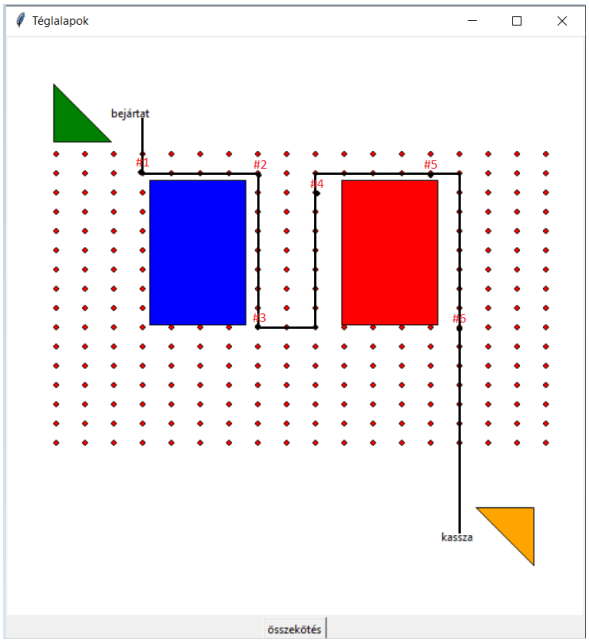
64. ábra 3. Blokk pontok összekötése (saját ábra)

Az első ponthoz a bejárat szövegből húzza a vonalat az első fekete ponthoz. **self.connect_along_red_points** függvényt hívja meg, amely összeköti két pontot a vörös körök közötti útvonalon, közben a lehelyezett fekte pontok koordinátáit figyelembe veszi, majd legvégül pedig a kassza szöveggel köti össze.

```python
def dijkstra_with_red_points(self, start, end):

    distances = {vertex: float('infinity') for vertex in self.graph}
    distances[start] = 0
    priority_queue = [(0, start)]
    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in self.graph[current_vertex].items():
            distance = distances[current_vertex] + weight
            # Ha a szomszéd egy piros pont, mérlegeljük a távolságát is
            if neighbor.startswith('red_point_'):
                distance += self.calculate_red_points_distance(current_vertex, neighbor)

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances[end]
```

65. ábra 3. Blokk dijkstra algoritmus (saját ábra)

def dijkstra_with_red_points Dijkstra algoritmust valósít meg a legrövidebb útvonal megtalálásához, figyelembe véve a piros köröket. A **distances** inicializálja a távolságokat, minden csúcs távolsága végtelen, kivéve a kezdő csúcsot, ami 0. **current_distance, current_vertex = heapq.heappop(priority_queue)** kiválasztja a prioritási sor legkisebb súlyú csúcsát. **if current_distance > distances[current_vertex]: continue** - Ellenőrzi, hogy a megtett út rövidebb-e a jelenlegi távolságnál. Ha igen, akkor folytatja a következő iterációt. **if neighbor.startswith('red_point_'):** - Ellenőrzi, hogy a szomszéd egy piros pont-e. Ha van akkor hozzá adja piros körök távolságát. **distance = distances[current_vertex] + weight** - Kiszámolja a szomszédhoz vezető legrövidebb út távolságát. **if distance < distances[neighbor]:** - Ellenőrzi, hogy a frissített távolság kisebb-e, mint a meglévő, ha igen akkor frissíti a távolságokat[9]



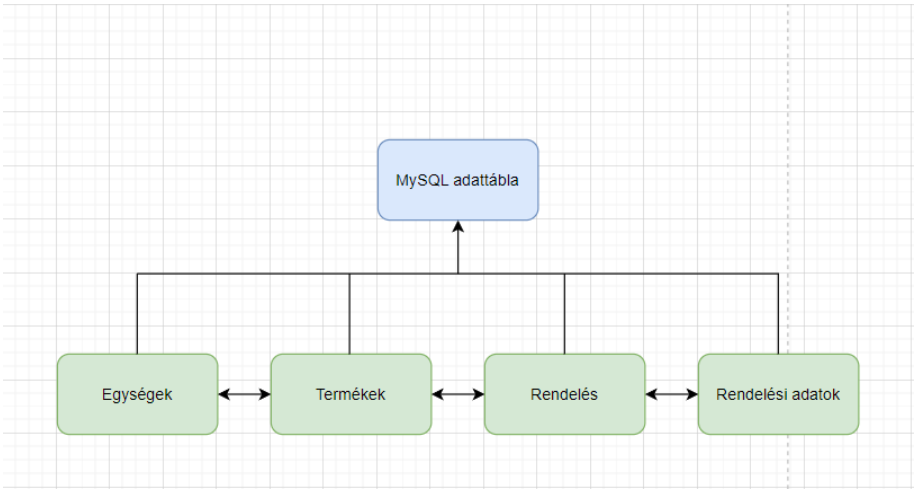
66. ábra 3. Blokk grafikus felület működés közben

Az ábrán látható a „legrövidebb” útvonal kirajzoltatása a piros körök mentén. Látszik, hogy az általunk meghatározott szabályok jól működnek tehát, a fekete pontokat lehelyezzük, amelyet, folyamatosan növeli a következő lehelyezett fekete pont értékét, a kezdő, illetve végpontot is helyesen megtalálja, valamint ki kerüli a két téglalapot, ugyanis ott nincsenek piros körök az algoritmus pedig csak az alapján tud menni.

8.4. 4. Blokk – Adatbázis és weboldal

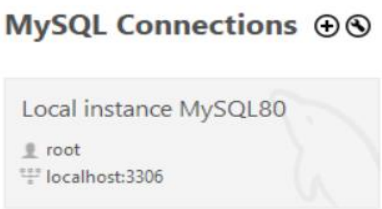
Az utolsó blokként egy adatbázissal összekötött weboldalt terveztem meg amelynél fontos szempont volt, hogy informatív legyen és ténylegesen valóban kommunikáció történjen az adatbázis, illetve a weboldal között. Az adatbázisnak egy MySQL adatbázist választottam az elő tanulmányaim miatt, illetve emellett ingyenesen beszerezhető program, valamint számos projekt található az interneten. A MySQL adatbázisomat összekötöttem egy Pythonban írt résszel, ami lényegében ugyan azt tudja mit az adatbázis, de kellett egy keret rendszer, ahol ezeket az adatokat fel tudom majd tölteni, illetve továbbítani tudom a weboldal felé. Weboldal tekintetében pedig Html keret rendszerű weboldalt választottam, amely api root-on keresztül éri el a megfelelő portokat A tervezett projekt utolsó blokkja az adatbázissal összekötött weboldal kialakítása, amelynek két fő pillére van: az adatbázis kezelése és a weboldal felülete. Az adatbázis aspektusában a MySQL-t választottam, ennek hátterében az előző tanulmányaimból származó ismeretek és az interneten elérhető számos projekt szolgált. Az ingyenesen elérhető programok segítségével könnyen implementálható és skálázható, míg a Pythonban írt backend rész szolgál az adatbázis és a weboldal közötti hatékony kommunikációval. A Pythonban írt rész lényegében ugyanazokat a funkcionalitásokat nyújtja, mint az adatbázis, és ezen keresztül érhetőek el, frissíthetőek vagy tölthetőek fel az adatok. Azonban az adatok kezeléséhez és átviteléhez szükség volt egy keretrendszerre, amely segíti a rendszer strukturált működését. Ezen a ponton HTML keretrendszerű weboldalt választottam, amely API root-on keresztül éri el a megfelelő portokat. Ez lehetővé teszi az adatok szabad áramlását a backend és a frontend között, lehetőséget adva a weboldalnak az adatok megjelenítésére és kezelésére. A biztonság terén különös figyelmet fordítok arra, hogy megfelelően védjem az adatokat és a felhasználói információkat. Az SQL sérülése elkerülése érdekében precízen kidolgozott paraméterezett lekérdezéseket alkalmazok, és az adatátviteli folyamatokat is olyan módon biztosítom, hogy azok védettek legyenek külső fenyegetésektől. Emellett a rendszert alaposan tesztelem, beleértve az adatbázis lekérdezéseit, a Python backend funkcionalitását és a webalkalmazás felhasználói felületét. Ezt azért teszem, hogy biztos lehessek abban, hogy minden egyes részlet megfelelően működik, és a különböző komponensek egymással zökkenőmentesen együttműködnek.

8.4.1. MySQL



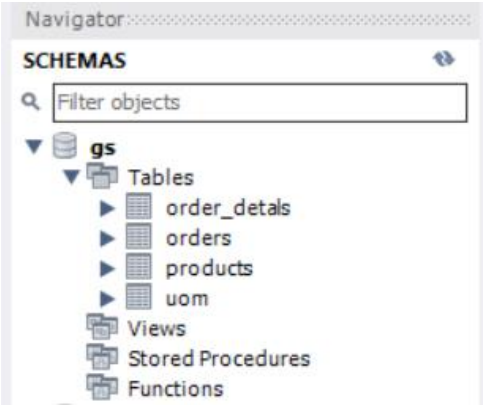
67. ábra 4. Blokk MySQL folyamatábra (saját ábra)

Az MqSQL folyamat ábráját írja le, amely során létrehozom az adattáblákat, amelyeket összekötök egymással, hogy tudják a különböző ellenőrzési funkciókat, mint például a helyes adat bevitel termék feltöltés esetén (mértékegység).



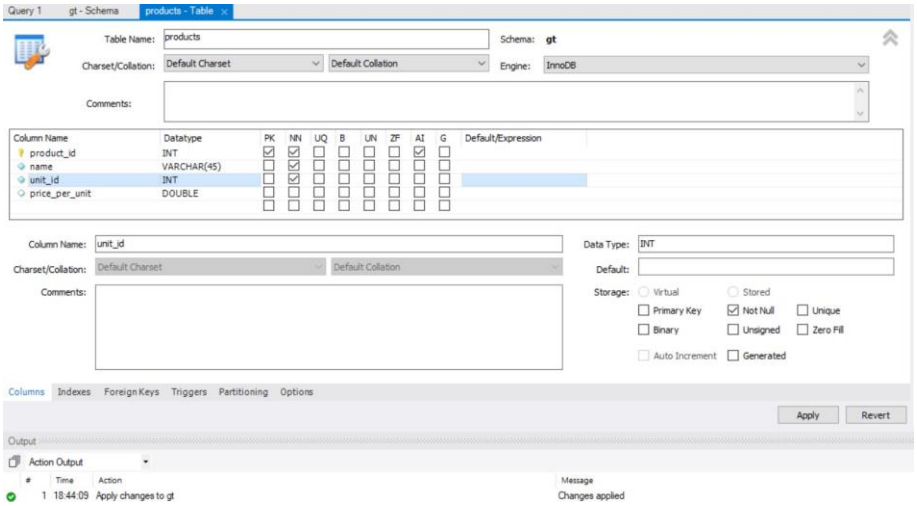
68. ábra 4. Blokk MySQL adatbázis létrehozása (saját ábra)

Létrehozom a projektemet a workbenchemet, és gondoskodom arról, hogy megfelelően beállítsam a szükséges paramétereket.



69. ábra 4. Blokk MySQL adattáblák létrehozása (saját ábra)

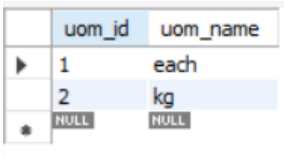
Az adatbázis létrehozása során alaposan átgondolom, hogy milyen adatokat kívánok tárolni, és mérlegelten hozok létre különböző táblákat az adatok strukturált és hatékony kezelése érdekében. Láthatóan 4 táblát hoztam létre.



70. ábra 4. Blokk MySQL products adattábla (saját ábra)

A products adattábla feltöltése során részletesen definiálom azokat az információkat, amelyeket tárolni kívánok. A product_id az egyedi azonosító, ami minden terméket egyértelműen azonosít, a name a termék nevét tükrözi, míg a unit_id az adott termékhez kapcsolódó egység egyedi azonosítóját tartalmazza. A folyamat során gondosan kiválasztom azokat az oszlopokat,

amelyek értékei nem lehetnek nullák. Ez azt jelenti, hogy minden termék esetében kötelező megadni az egységenkénti árat, a termék nevét, valamint a product_id-t. Ezzel biztosítom, hogy a tábla minden sorában megbízható és teljes körű információk legyenek elérhetők. A datatype oszlopban részletesen meghatározom, hogy az INT típus alatt csak egész számokat fogadok el, míg a VARCHAR típus alatt maximum 45 karakteres karakterláncokat várok. Ez a strukturált adatok kezelését segíti, és elősegíti az adatok integritásának és megbízhatóságának fenntartását a táblában.



uom_id	uom_name
1	each
2	kg
NULL	NULL

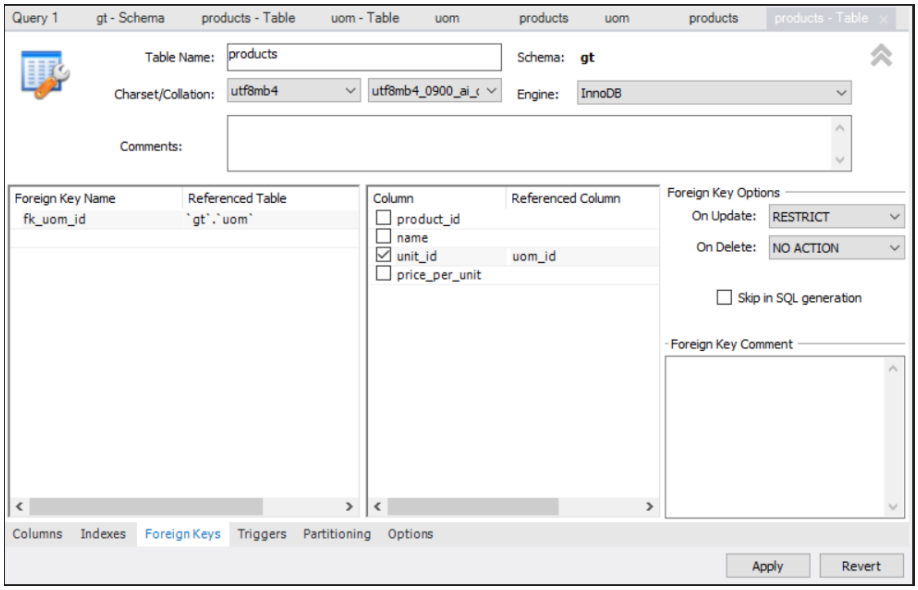
71. ábra 4. Blokk MySQL uom tábla (saját ábra)

Az uom táblát készítem el, amelyben a különböző mértékegységek kerülnek tárolásra. Az id oszlop automatikus növekedést kapott, így minden újonnan hozzáadott mértékegység esetén az id értéke automatikusan eggyel növekszik. Ez biztosítja, hogy minden mértékegység egyedi azonosítóval rendelkezzen a táblában, megkönnyítve ezzel az adatok kezelését és az azokhoz való hozzáférést.

```
1    INSERT INTO `gt`.`uom` (`uom_id`, `uom_name`) VALUES ('1', 'each');
2    INSERT INTO `gt`.`uom` (`uom_id`, `uom_name`) VALUES ('2', 'kg');
```

72. ábra 4. Blokk MySQL uom tábla feltöltése (saját ábra)

Feltöltöm a létrehozott uom tábla adatait.



73. ábra 4. Blokk MySQL products adattábla Foreign Keys létrehozása (saját ábra)

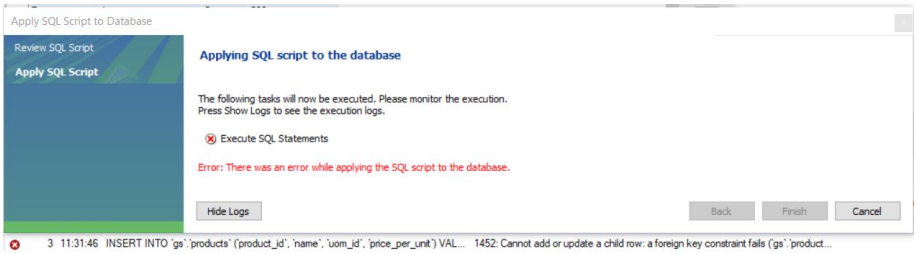
Különböző Foreign Keys-eket hozok létre, amelyek segítségével optimalizálom a product tábla feltöltését. Ezen kulcsok segítségével a product tábla a uom táblára hivatkozik, és csak az általam előre meghatározott id-ket fogadja el az uom oszlopban. Például, a korábban feltöltött 1-es érték az "each" mértékegységet, míg a 2-es érték a "kg" mértékegységet jelenti. Ezáltal egy plusz funkcionalitást is bevezetek, amely segít ellenőrizni, hogy az adatbázis csak a várt értékeket fogadja el. Implementáltam egy ellenőrzést, amely figyeli, hogy csak az előre definiált 1-es vagy 2-es értéket lehessen megadni a mértékegység oszlopban (uom). Tehát, ha valaki véletlenül más értéket próbál megadni, például a 4-et, az adatbázis hibaüzenetet dob, így biztosítva a koherens és pontos adatokat a táblában.

74. ábra 4. Blokk MySQL Foreign Keys működése 1 (saját ábra)

Az ábrák hatékonyan szemléltetik azt a "szakdolgozat" terméket, amelyet be szeretnék vinni a rendszerbe. Ebben az esetben a 4-es számot adom meg az uom oszlopban.

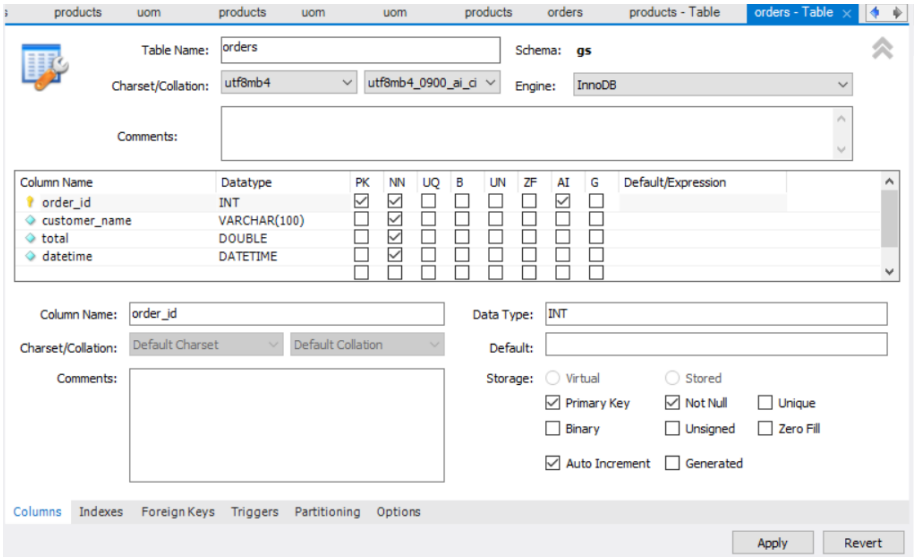
75. ábra 4. Blokk MySQL Foreign Keys működése 2 (saját ábra)

Megpróbálom feltölteni 4-es számmal ellátott uom oszlopot.



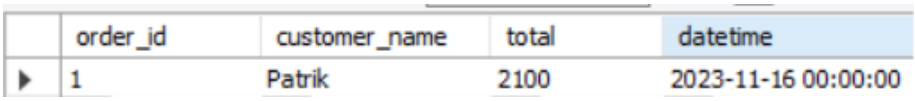
76. ábra 4. Blokk MySQL Foreign Keys működése 3 (saját ábra)

A rendszer azonban a feltöltés után hibaüzenetet generál, amely szemlélteti a Foreign Keys működését. Ez a mechanizmus biztosítja, hogy csak az előre meghatározott, érvényes mértékegység-azonosítókat lehessen használni, ezzel hozzájárulva az adatbázis integritásához és az adatok konzisztenciájához.



77. ábra 4. Blokk MySQL orders adatábla létrehozása (saját ábra)

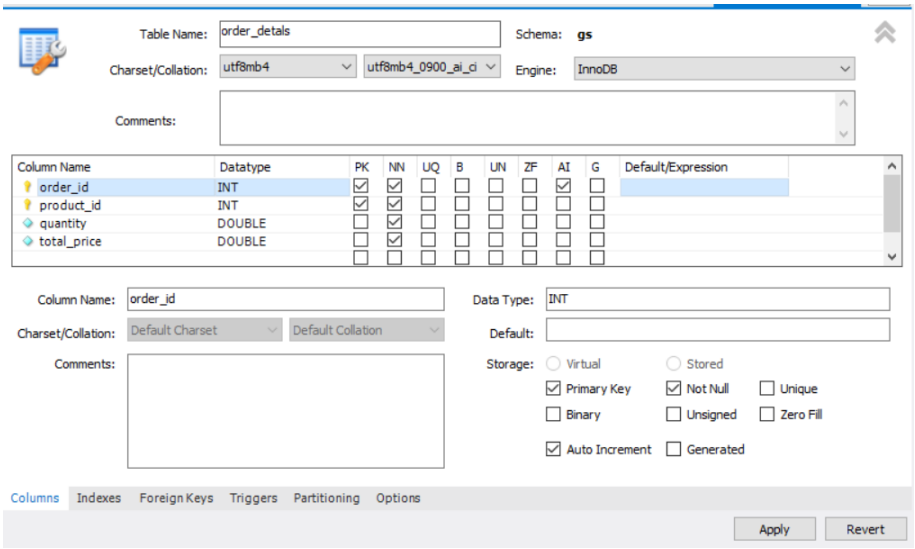
Az ábrán bemutatott folyamat során egy új adattábla, az "orders" létrehozására kerül sor, mely a megrendelési adatok tárolására szolgál. Az "orders" nevét azért választottam, mert ebben a táblában rögzítem a megrendelésekhez kapcsolódó információkat. Ebben a táblában található egy id, amely sosem lehet nulla, és automatikusan növeli az értékét minden új rendelés hozzáadásakor. Ez az azonosító egyedien azonosítja a rendeléseket, és segít a rendezésben és az azonosításban. Ezenkívül fontos adatokat is tárolok, például a vásárló nevét, amit "customer_name"-ként neveztem el. A "total" oszlop a rendelés teljes összegét tartalmazza, míg a "datetime" az adott rendelés időpontját rögzíti. Minden oszlopnak kötelező kitöltendőnek lennie, és ezt az "NOT NULL" kikötéssel biztosítom. Tehát minden mezőnek értéket kell kapnia a tábla létrehozásakor vagy adatainak feltöltésekor, különben a folyamat nem sikerülhet. Ez a szabályozás garantálja az adatok integritását és pontos tárolását a rendszerben.



order_id	customer_name	total	datetime
1	Patrik	2100	2023-11-16 00:00:00

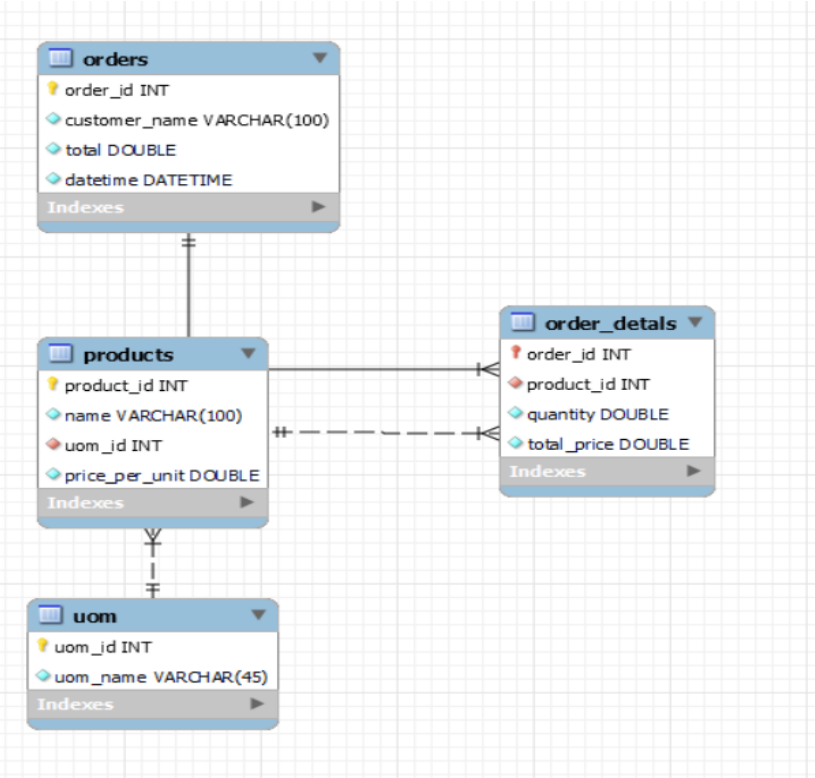
78. ábra 4. Blokk MySQL orders tábla megjelenítése (saját ábra)

Helyes feltöltés után következőképpen így nézz ki az adott táblánk. Ezzel párhuzamosan létre hozom az order_details adattáblát mint ahogy a neve is mutatja ebben az adott rendelési adatokat tárolom.



79. ábra 4. Blokk MySQL order_detals létrehozása (saját ábra)

Az adatok tárolásához szükséges oszlopnevek a következők: először is, egy "order_id", ami egy rendelés egyedi azonosítója, továbbá egy "product_id", amely a termék táblában található azonosító alapján kap értéket. Ez a mező egy Foreign Key, amely biztosítja a helyes azonosítást a kapcsolódó termékekhez. Emellett van egy "quantity" oszlop, ami a rendelt termék mennyiségét jelöli, és végül egy "total_price" oszlop, amely a rendelés teljes összegét tartalmazza. Minden oszlopnak kötelező értékkel kell rendelkeznie, azaz a "NOT NULL" kikötéssel biztosítom, hogy minden szükséges információ rendelkezésre álljon a táblában. Ezen adatok rögzítése segíti a rendszer teljes körű és pontos nyomon követését a megrendelésekről, miközben a Foreign Key segítségével ellenőrizhető, hogy minden egyes rendelés a megfelelő termékhez van rendelve.



80. ábra 4. Blokk MySQL saját programon belüli adatstruktúra kapcsolat (saját ábra)

Az adatbázisunk sikeresen létrejött, és a MySQL lehetőséget kínál egy vizuális adattábla blokkvázlat elkészítésére. Ez a grafikus ábrázolás lehetővé teszi számunkra, hogy könnyedén áttekinthessük az adattábláink közötti kapcsolatokat és kapcsolódásokat.

8.4.2. Python

Ebben a részben létrehoztam Pycharm fejlesztői környezetben python programozási nyelvben egy adatbázis csatlakozást, feltöltést tehát a backend oldalát még pluszban ennek az egész 4-es számú blokknak.

```
import mysql.connector

__cnx = None

def get_sql_connection():
    global __cnx
    if __cnx is None:
        __cnx = mysql.connector.connect(user='root', password='0e889a689',
                                        host='127.0.0.1',
                                        database='gs')
        return
```

81. ábra 4. Blokk Python sql_connection.py szerver kapcsolat létrehozása (saját ábra)

Ezt a python kód részletet az adatbázisom csatlakozásához készítettem, amelyet el kell indítani ahhoz, hogy a python illetve a MySQL adatbázis csatlakozzon egymáshoz. A __cnx egy globális változót definiálok, amelyet későbbiekben használom az adatbázis kapcsolat tárolásához. Továbbá definiált get_sql_connection függvény, amely lekéri a kapcsolatot, ha az már létrejött akkor, akkor nem hoz létre új kapcsolatot. __cnx változónak a None érték megadásával bizonyosodunk meg az alaphelyzetről tehát, hogy nincs kapcsolat. Ahogy a képen is látható itt az adatbázisunknak a nevét (workbench), jelszavát, illetve host-ját kell megadni, valamint a létrehozott adatbázis nevét ezek paraméterek helyes bevitelével már megfelelően kapcsolódik az adatbázisunk.[10]

```
def get_uoms(connection):
```

82. ábra 4. Blokk Python uom_dao.py szerver csatlakozás (saját ábra)

A függvénynek egy adatbázis kapcsolatot kell kapnia paraméterként. Ezt a kapcsolatot a **get_sql_connection** függvény segítségével hozzom létre. Az **adatbazis_kapcsolat** változóban tárolja a kapcsolatot.

```
    cursor = connection.cursor()
```

83. ábra 4. Blokk Python uom_dao.py cursor (saját ábra)

Létrehoz egy ún. "kurzort", amely az adatbázisműveletek végrehajtására szolgál. A kurzor egyfajta pozícionálható mutató az adatbázisban, amelyet a program használ az eredmények feldolgozására.

```
    query = ("select * from uom")
    cursor.execute(query)
```

84. ábra 4. Blokk Python uom_dao.py SQL lekérdezés 1 (saját ábra)

A függvény definiál egy SQL lekérdezést, amely a "uom" táblából kiválasztja az összes oszlopot (*). A kurzor segítségével végrehajtja ezt a lekérdezést az adatbázisban.

```
    response = []
    for (uom_id, uom_name) in cursor:
        response.append({
            'uom_id': uom_id,
            'uom_name': uom_name
        })
    return response
```

85. ábra 4. Blokk Python uom_dao.py SQL lekérdezés 2 (saját ábra)

Az eredményeket egy üres listába (**response**) tárolja. A kurzor által visszaadott eredményeket egy **for** ciklus segítségével iterálja végig. Minden rekordot egy szótár formájában tárol el a listában, ahol a **uom_id** és **uom_name** kulcsok az adott rekordot jellemzik.

```
def get_all_products(connection):
    cursor = connection.cursor()
    query = ("select products.product_id, products.name, products.uom_id, products.price_per_unit,"
             " uom.uom_name from products inner join uom on products.uom_id=uom.uom_id")
    cursor.execute(query)
    response = []
```

86. ábra 4. Blokk Python products.py táblák csatlakoztatása (saját ábra)

Az SQL lekérdezés a **products** és **uom** táblák közötti belső csatlakozást (**INNER JOIN**) végez. A lekérdezés kiválasztja a **products** tábla **product_id**, **name**, **uom_id**, **price_per_unit** oszlopait, valamint a **uom** tábla **uom_name** oszlopát. A két tábla közötti kapcsolatot az **uom_id** mezők egyenlőségével határozza meg.

```
response = []
for (product_id, name, uom_id, price_per_unit, uom_name) in cursor:
    response.append({
        'product_id': product_id,
        'name': name,
        'uom_id': uom_id,
        'price_per_unit': price_per_unit,
        'uom_name': uom_name
    })
return response
```

87. ábra 4. Blokk Python products.py eredmény tárolása (saját ábra)

Az eredményeket egy üres listába (**response**) tárolja. A kurzor által visszaadott eredményeket egy **for** ciklus segítségével iterálja végig. Minden rekordot egy szótár formájában tárol el a listában, ahol a kulcsok a kiválasztott oszlopnevek.

```
def insert_new_product(connection, product):
    cursor = connection.cursor()
    query = ("INSERT INTO products "
             "(name, uom_id, price_per_unit)"
             "VALUES (%s, %s, %s)")
```

88. ábra 4. Blokk Python products.py új termék hozzáadása 1 (saját ábra)

Definiálja az SQL lekérdezést az új termék beszúrására. A **%s** helyőrzőkkel jelzi, hogy a később megadott értékeket kell beszúrni a megfelelő helyekre.

```
data = (product['product_name'], product['uom_id'], product['price_per_unit'])
```

89. ábra 4. Blokk Python products.py új termék hozzáadása 2 (saját ábra)

Elkészíti a beszúrandó adatokat a **product** szótárból.

```
    cursor.execute(query, data)
    connection.commit()

    return cursor.lastrowid
```

90. ábra 4. Blokk Python products.py új termék hozzáadása 3 (saját ábra)

A kurzor segítségével végrehajtja a SQL lekérdezést, és a **%s** helyőrzőket a **data** változóban található értékekkel helyettesíti. **connection.commit()** A tranzakciót menti, azaz véglegesíti az adatbázisban végrehajtott műveleteket. **last_inserted_id = cursor.lastrowid** Lekéri az újonnan beszúrt rekord azonosítóját az adatbázisból. **cursor.close()**Bezárja az adatbázis kurzort az erőforrások hatékony kezelése érdekében. **return last_inserted_id** Visszaadja az újonnan beszúrt termék azonosítóját, amit a függvény hívója felhasználhat.

```
def delete_product(connection, product_id):
    cursor = connection.cursor()
```

91. ábra 4. Blokk Python products.py termékek törlése 1 (saját ábra)

Inicializál egy adatbázis kurzort, amely lehetővé teszi a SQL műveletek végrehajtását.

```
query = ("DELETE FROM products where product_id=" + str(product_id))
```

92. ábra 4. Blokk Python products.py termékek törlése 2 (saját ábra)

Meghatározza az SQL lekérdezést, amelynek segítségével törli a terméket a megadott azonosító alapján az adatbázisból.

```
cursor.execute(query)
```

93. ábra 4. Blokk Python products.py termékek törlése 3 (saját ábra)

A tranzakciót menti, azaz véglegesíti az adatbázisban végrehajtott műveleteket.

```
return cursor.lastrowid
```

94. ábra 4. Blokk Python products.py termékek törlése 4 (saját ábra)

Az utolsó sor azonosítóját kérdezi le. Fontos megjegyezni, hogy a törlési művelet esetén ez az érték nem releváns, mivel a törölt sor már nincs jelen az adatbázisban. Bezárja az adatbázis kurzort az erőforrások hatékony kezelése érdekében. Visszaadja az utolsó sor azonosítóját. Fontos megjegyezni, hogy a törlési művelet esetén ez az érték nem értelmezett, és csak a konzisztencia miatt maradt benne.

```python
def insert_new_product(connection, product):
    cursor = connection.cursor()
    query = ("INSERT INTO products "
             "(name, uom_id, price_per_unit)"
             "VALUES (%s, %s, %s)")
    data = (product['product_name'], product['uom_id'], product['price_per_unit'])

    cursor.execute(query, data)
    connection.commit()

    return cursor.lastrowid

if __name__ == '__main__':
    connection = get_sql_connection()
     #print(get_all_products(connection))
    print(insert_new_product(connection, {
        'product_name': 'cabbage',
        'uom_id': '1',
        'price_per_unit': 10
    }))
```

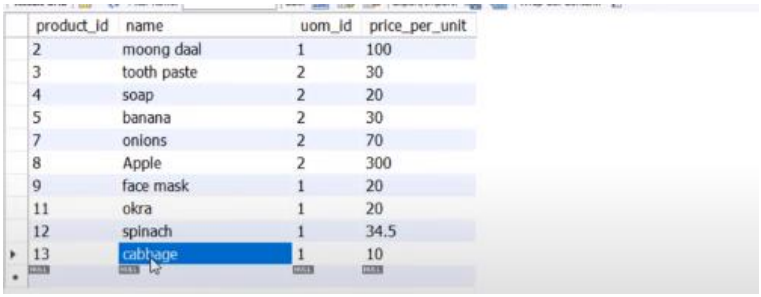
95. ábra 4. Blokk Python products.py új termék felvétel az adatbázisba (saját ábra)

Ebben a részletben először egy adatbázis kapcsolatot hoz létre a **get_sql_connection** függvény segítségével. Ezt követően az **insert_new_product** függvény felhasználásával új terméket szúr be az adatbázisba, melynek neve 'cabbage', egységkezelője '1', és egységára 10. Az újonnan beszúrt termék azonosítóját kiírja a képernyőre. Kiemelendő, hogy a kód a **if __name__ == '__main__':** blokkban található, tehát csak akkor hajtódik végre, ha a fájlt közvetlenül futtatják, nem pedig más helyről importálják. Ezenkívül, ha a korábban részletezett függvénydefiníciókat beillesztettük, a megfelelő helyeken módosítottuk a print sorokat, lehetőség nyílik a különböző paraméterek megadásával adatok feltöltésére vagy más függvények esetében azok törlésére.

```
13

Process finished with exit code 0
```

96. ábra 4. Blokk Python products.py utolsó oszlop kiíratása (saját ábra)

A kód lefutásával kiírja a terminál hányadik oszlopba helyezte be ezt az adott általunk bevinni kívánt termékek.



product_id	name	uom_id	price_per_unit
2	moong daal	1	100
3	tooth paste	2	30
4	soap	2	20
5	banana	2	30
7	onions	2	70
8	Apple	2	300
9	face mask	1	20
11	okra	1	20
12	spinach	1	34.5
13	cabbage	1	10

97. ábra 4. Blokk Python products.py feltöltés ellenőrzése MySQL-ben (saját ábra)

Az ábrán láthassuk is, hogy megfelelően lefutott is bele is helyezte.

```
from datetime import datetime
from sql_connection import get_sql_connection
```

98. ábra 4. Blokk Python orders.py importálás (saját ábra)

from datetime import datetime: Importálja a **datetime** modulból csak a **datetime** osztályt, amely lehetővé teszi a dátum és idő kezelését. **from sql_connection import get_sql_connection**: Importálja a **get_sql_connection** függvényt a **sql_connection** modulból, amely segítségével kapcsolatot hoz létre az adatbázissal.

```
def insert_order(connection, order):
    cursor = connection.cursor()
```

99. ábra 4. Blokk Python orders.py curzor létrehozása (saját ábra)

cursor = connection.cursor(): Létrehoz egy kurzort az adatbáziskapcsolatban.

```
order_query = ("INSERT INTO orders "
               "(customer_name, total, datetime)"
               "VALUES (%s, %s, %s)")
order_data = (order['customer_name'], order['grand_total'], datetime.now())
cursor.execute(order_query, order_data)
order_id = cursor.lastrowid
order_details_query = ("INSERT INTO order_details "
                       "(order_id, product_id, quantity, total_price)"
                       "VALUES (%s, %s, %s, %s)")
order_details_data = []
```

100. ábra 4. Blokk Python orders.py SQL lekérdezése (saját ábra)

Az **order_query** változó létrehoz egy SQL lekérdezést az új rendelés adatainak beszúrására az **orders** táblába, ideértve az ügyfél nevét, a teljes összeget és az aktuális időpontot. A **order_data** változó előkészíti az új rendelés adatait a SQL lekérdezéshez, például az ügyfél nevét, a teljes összeget és az aktuális időpontot. A **cursor.execute(order_query, order_data)** kódsor végrehajtja az SQL lekérdezést az **orders** táblába történő új rendelés adatainak beszúrására. Az **order_id** változóban tárolódik az újonnan beszúrt rendelés azonosítója. Az **order_details_query** változó létrehoz egy SQL lekérdezést az új rendelés részleteinek beszúrására az **order_details** táblába, beleértve a rendelés azonosítóját, a termék azonosítóját, a mennyiséget és a teljes árat. Az **order_details_data** változó egy üres lista, amely előkészíti a rendelés részleteinek adatokat.

```
for order_detail_record in order['order_details']:
    order_details_data.append([
        order_id,
        int(order_detail_record['product_id']),
        float(order_detail_record['quantity']),
        float(order_detail_record['total_price'])
    ])
cursor.executemany(order_details_query, order_details_data)

connection.commit()

return order_id
```

101. ábra 4. Blokk Python orders.py rendelési lista létrehozása (saját ábra)

for order_detail_record in order['order_details']:: A rendelés részletein végig iterálva előkészíti azokat a beszúrás céljából. **order_details_data.append ([...])**: Ezután az egyes rendelés részleteit tartalmazó listát hozzáadja az **order_details_data** listához. **cursor.executemany (order_details_query, order_details_data)**: Miután összegyűjtötte a rendelés részleteit, végrehajtja az SQL lekérdezést, amely az összes rendelés részletét beilleszti az **order_details** táblába. **connection.commit()**: A tranzakció megerősítése céljából a mentésre kerültek az adatbázisban végrehajtott műveletek. **return order_id**: Végül visszaadja az újonnan beszúrt rendelés azonosítóját.

```
cursor = connection.cursor()
query = "SELECT * from order_details where order_id = %s"
query = "SELECT order_details.order_id, order_details.quantity, order_details.total_price, "\
        "products.name, products.price_per_unit FROM order_details LEFT JOIN products on " \
        "order_details.product_id = products.product_id where order_details.order_id = %s"
data = (order_id, )
cursor.execute(query, data)
records = []
```

102. ábra 4. Blokk Python orders.py order_details és products táblák közötti kapcsolat (saját ábra)

query = "SELECT * from order_details where order_id = %s": Egy alap lekérdezést állít be az **order_details** táblából, amely csak az adott rendeléshez tartozó részleteket kéri le. **query = "SELECT order_details.order_id, ..."**: Később módosítja a lekérdezést, hogy az összes szükséges információt lekérje az **order_details** és **products** táblák közötti összekapcsolással. **data = (order_id,)**: Az adatokat egy tuple-be helyezi, amelyet a lekérdezés végrehajtásakor használ. **cursor.execute(query, data)**: Végrehajtja a lekérdezést az előkészített adatokkal. **records = []**: Létrehoz egy üres listát a lekért rekordok tárolására.

```
for (order_id, quantity, total_price, product_name, price_per_unit) in cursor:
    records.append({
        'order_id': order_id,
        'quantity': quantity,
        'total_price': total_price,
        'product_name': product_name,
        'price_per_unit': price_per_unit
    })
cursor.close()
return records
```

103. ábra 4. Blokk Python orders.py eredmények hozzáadása (saját ábra)

for (order_id, quantity, total_price, product_name, price_per_unit) in cursor:: Végigiterál a kurzor által visszaadott eredményeken, minden rekordot hozzáad a **records** listához. **cursor.close()**: A kurzor bezárásával gondoskodik az erőforrások hatékonyabb kezeléséről. **return records**: A lekért rekordokat tartalmazó listával tér vissza, melyek az adott rendelés részleteit reprezentálják.

```python
if __name__ == '__main__':
    connection = get_sql_connection()
    #print(get_all_orders(connection))
    #print(get_order_details(connection,4))
    print(insert_order(connection,
        {
         'customer_name': 'Patrik',
         'total': '2100',
         'datetime': datetime.now(),
         'order_details': [
             {
                 'product_id': 3,
                 'quantity': 3,
                 'total_price': 2100
             },

         ]
     }))
```

104. ábra 4. Blokk Python orders.py rendelési adatok feltöltése MySQL-be (saját ábra)

Majd legvégül látható miképpen is kerül be az adatbázisunkba az adat, tehát be kell írni a kívánt adatokat.

```python
from flask import Flask, request, jsonify
from sql_connection import get_sql_connection
import json
import products_dao
import orders_dao
import uom_dao

app = Flask(__name__)

connection = get_sql_connection()
```

105. ábra 4. Blokk Python server.py importálások (saját ábra)

Az alap **Flask** modulok, mint a **Flask**, **request** és **jsonify**, importáló kódrészlet jelenti a webalkalmazás magját. A **Flask** objektum szolgál a webalkalmazás fő keretrendszerének, a request modul kezeli a HTTP kéréseket, míg a **jsonify** segít az adatok **JSON** formátumba alakításában. Az adatbáziskapcsolat létrehozásáért felelős **get_sql_connection** függvény

importálása a **sql_connection** modulból. Az adatok JSON formátumban történő munkavégzést lehetővé tevő json modul importálása. A termékekkel kapcsolatos adatbázis-műveletek implementációját tartalmazó **products_dao** modul importálása. A rendelésekkel kapcsolatos adatbázis-műveletek implementációját tartalmazó **orders_dao** modul importálása. Az egységkezelőkkel kapcsolatos adatbázis-műveletek implementációját tartalmazó **uom_dao** modul importálása. Az alkalmazás létrehozása a Flask objektum segítségével. Az **name** az aktuális modul nevével egyezik meg, és szolgál a Python fájl nevének azonosítására.

```
@app.route('/getUOM', methods=['GET'])
def get_uom():
    response = uom_dao.get_uoms(connection)
    response = jsonify(response)
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response
```

106. ábra 4. Blokk Python server.py getUOM app.route (saját ábra)

A Flask alkalmazásban az '/getUOM' útvonalat a **get_uom** nevű függvény kezeli, kizárólag HTTP GET kérésekre reagálva. Ezen végpont függvénye először a **uom_dao** modul **get_uoms** függvényét hívja meg, amely az adatbázisból lekéri az egységkezelőket, majd a választ JSON formátumba alakítja a **jsonify** segítségével. A kliens által küldött kérést bármilyen eredeti forrásról történő elérést lehetővé téve, az 'Access-Control-Allow-Origin' fejléccel rendeli hozzá a válaszhoz. A függvény végül ezt a formázott JSON választ adja vissza, amely az egységkezelők adatait tartalmazza.

Az összes többi HTTP kérésnek megfelelően, a '/getProducts', '/insertProduct', '/getAllOrders', '/insertOrder' és 'deleteProduct' útvonalakat is hozzáadtam az alkalmazáshoz.

```
@app.route('/insertProduct', methods=['POST'])
def insert_product():
    request_payload = json.loads(request.form['data'])
    product_id = products_dao.insert_new_product(connection, request_payload)
    response = jsonify({
        'product_id': product_id
    })
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response
```

107. ábra 4. Blokk Python server.py insertProduct app.route (saját ábra)

@app.route('/insertProduct', methods=['POST']): A '/insertProduct' útvonalat definiálja, és csak HTTP POST kérésekre reagál. **def insert_product():**: Létrehozza az 'insert_product' nevű

funkciót, amely az '/insertProduct' útvonalon érkező kéréseket kezeli. **request_payload = json.loads(request.form['data'])**: A beérkező JSON payload feldolgozása. A kérésből kinyeri a 'data' kulcsú JSON objektumot, majd a **json.loads** segítségével átalakítja Python adatszerkezetté. **product_id = products_dao.insert_new_product(connection, request_payload)**: Az új termék beszúrása az adatbázisba a **products_dao** modul **insert_new_product** függvényével. A függvény visszatérési értéke az új termék azonosítója. **response = jsonify({'product_id': product_id})**: A visszatérési érték JSON formátumba alakítása, tartalmazva az újonnan beszúrt termék azonosítóját. **response.headers.add('Access-Control-Allow-Origin', '*')**: Hozzáadja a válasz fejlécéhez az 'Access-Control-Allow-Origin' bejegyzést, engedélyezve ezzel a klienseknek a kéréseket bármely eredeti forrásról. **return response**: A funkció visszatér a formázott JSON válasszal, amely az újonnan beszúrt termék azonosítóját tartalmazza.

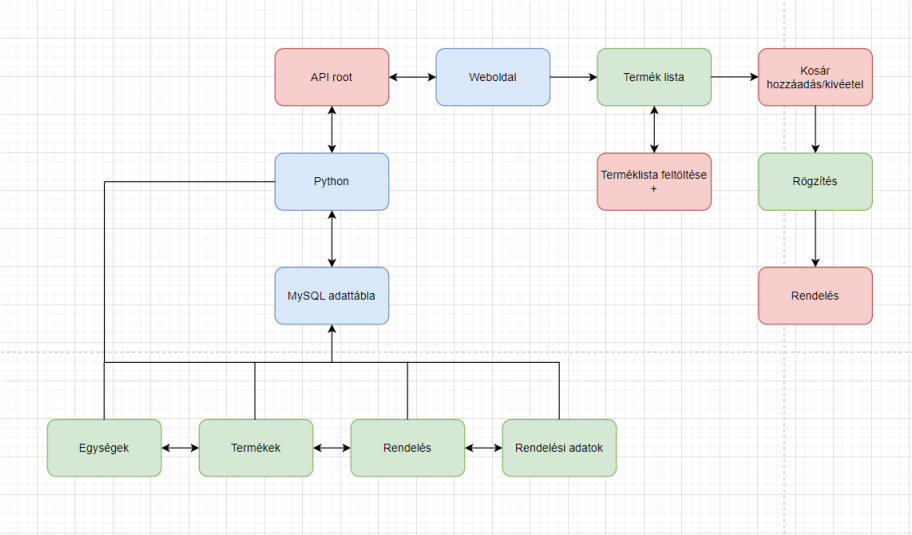
```
if __name__ == "__main__":
    print("Starting Python Flask Server For Grocery Store Management System")
    app.run(port=5000)
```

108. ábra Blokk Python server.py port kiíratása (saját ábra)

Ez a kódrészlet a Flask alkalmazás indításáért felel. A **if __name__ == "__main__":** rész ellenőrzi, hogy a Python fájl közvetlenül fut-e, vagy csak importálják-e más modulokból. Ha közvetlen futásról van szó, akkor az alábbi műveletek történnek: **print("Starting Python Flask Server For Grocery Store Management System")**: Kiírja a konzolra a szöveget, amely jelzi, hogy a Flask szerver elindult. **app.run(port=5000)**: Elindítja a Flask alkalmazást a 5000-es porton. Ezen a porton lesz elérhető az alkalmazás, és a kéréseket fogadja a kiszolgálás során.

8.4.3. Frontend

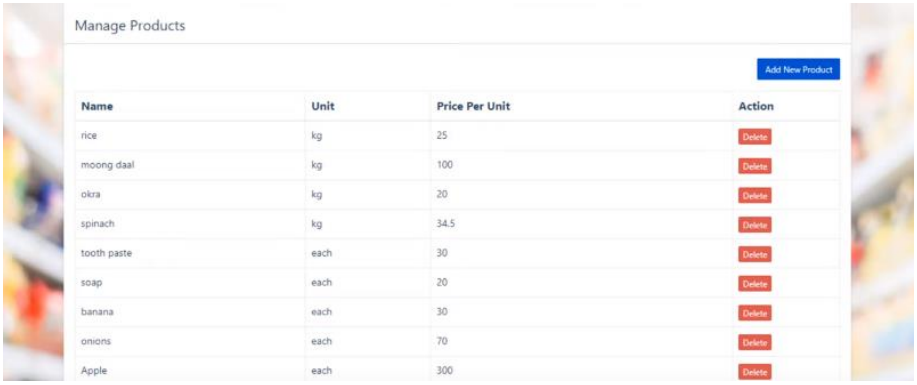
A weboldal frontendjén azt a célt tűztem ki, hogy a felhasználó számára számos olyan funkciót hozzak létre, amelyekről már korábban beszéltem. Konkrétan azon dolgoztam, hogy a MySQL adatbázisba feltöltött adatokat áttekinthető adattáblák formájában jelenítsem meg. Ezen célnak megfelelően HTML, CSS és JSON fájlokat hoztam létre, és ezeket implementáltam a kódban. Az HTML és CSS fájlok segítségével strukturáltam és formáztam a weboldal felületét, hogy esztétikus és felhasználóbarát legyen. A JSON fájlok használatával könnyen kezelhető és átvihető formában tároltam az adatokat, amelyeket a frontend számára megjelenítek. Az adatok eléréséhez és kezeléséhez egyedi API root-ot is kialakítottam. Ez az API root egy kapcsolódási pontként szolgál, ahol a frontend egyszerűen lekérheti az adatokat a szerverről. Ez a megoldás lehetővé teszi, hogy az adatok gyorsan és hatékonyan érkezzenek meg a frontend felé, optimalizálva ezzel a böngészési élményt.[11]



109. ábra 4. Blokk Frontend folyamatábra (saját ábra)

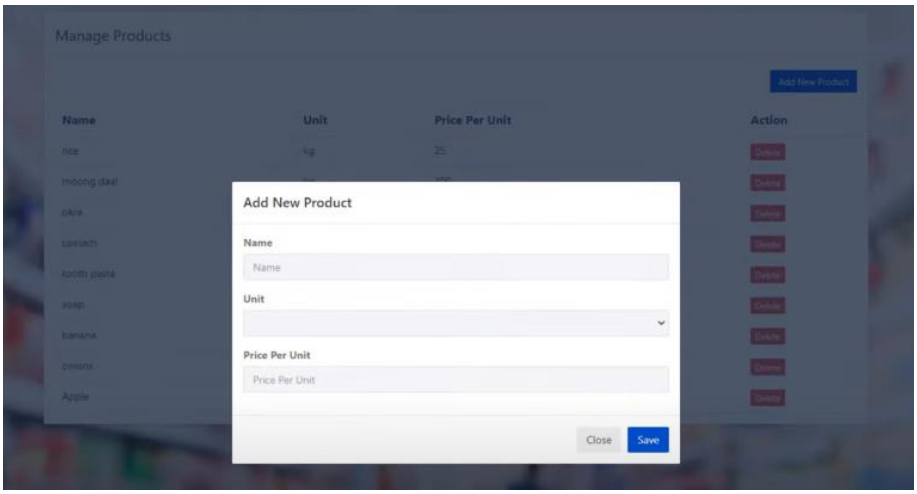
A folyamatábrán jól látható a korábban használt MySQL rész össze van kötve a korábbiakban kifejtett Python résszel, hogy elinduljon az API root, amely segítségével például az egyik

(getUOM) ilyen weboldalt nyitja meg. A felhasználó által megnyitott webfelületen pedig tudja kihasználni az implementált funkciókat.



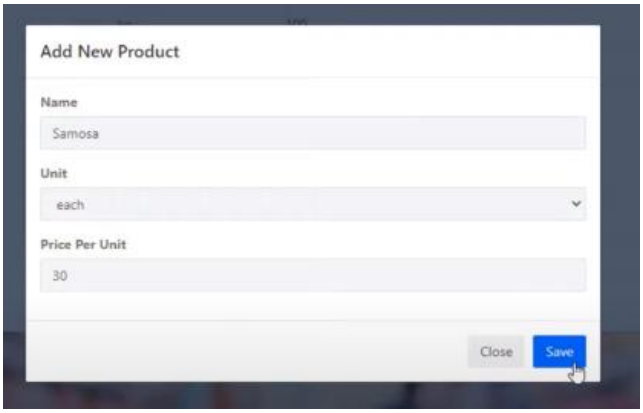
110. ábra 4. Blokk Frontend Manage Products oldal 1 (saját ábra)

A termékfelület megjelenése a következőképpen alakul, ahol lehetőségünk van termékek törlésére vagy új termékek létrehozására. A felület az eddigi oszlopokat tartalmazza, beleértve a termék nevét, mértékegységét és árát. Ezen a helyen könnyen áttekinthetjük és kezelhetjük a termékeket a megszokott információkkal és funkciókkal.



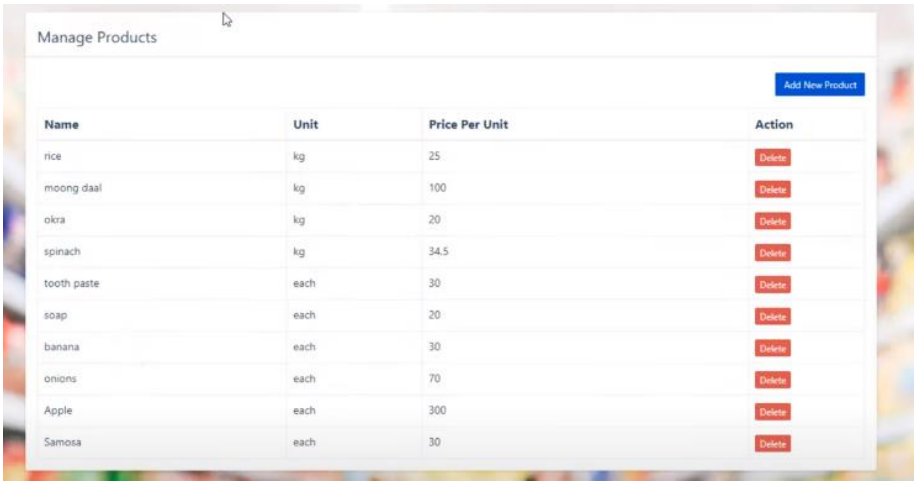
111. ábra 4. Blokk Frontend Add New Product 1 (saját ábra)

Az "Add New Product" funkciót láthatjuk, amely lehetővé teszi számunkra, hogy egyszerűen létrehozzunk egy új terméket. Miután sikeresen elmentettük, a termék az "Manage Products" oldalon jelenik meg.



112. ábra 4. Blokk Frontend Add New Product 2 (saját ábra)

Létre szeretnénk hozni egy új terméket Samosa néven, amelyet sikeresen mentünk el.



113. ábra 4. Blokk Frontend Manage Products oldal 2 (saját ábra)

Az ábrán pedig jól látható, hogy elmentődött az általunk létrehozott Samosa nevű termék.

9. Teszt ESP32-vel

A projekt kipróbálása mindenképpen vonzott engem, különösen azon részével kapcsolatban, hogy a rendszer mikrokontrolleres környezetben hogyan működik. Konkrétan a weboldal futtatására gondolok, és emiatt döntöttem úgy, hogy az ESP32 mikrokontrollert választom. Korábban már volt tapasztalatom az ESP32 programozásával, mivel részt vettem különböző otthoni projektekben, így a választásom egy olyan eszközre esett, amellyel már tisztában voltam. Így nem volt szükség erősebb mikrokontroller beszerzésére.

114. ábra ESP32 mikrokontroller [12]

Termék leírása:[12]

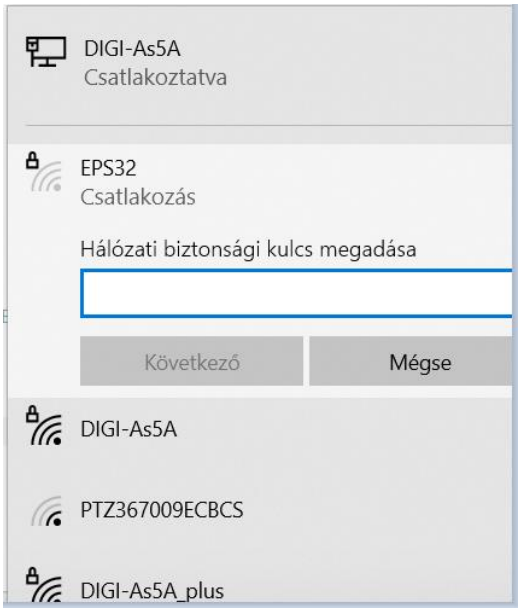
Mikrokontroller: ESP32 (Espressif)
Usb chip: CP2102
Arhitektúra: 32 bit RISC
Maximum tápfeszültség: 3.3V – 5V
Működési feszültség: 3.3V
Memória: 520 Kbyte SRAM
Flash memória: 4MB
Órajel: 160 – 240 Mhz
Wi-Fi: 802.11 b/g/n
Bluetooth: 4.2 BR/EDR és BLE

sketch_may3a.ino

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <WebServer.h>

// Replace with your network credentials
const char* ssid = "DIGI-As5A"; //Patrik Iphone
const char* password = "hHQQ7jqp";//Esp32jelszo
// Replace with your AWS S3 bucket URL and HTML file path
const char* aws_s3_url = "https://vizuri.hu/";//projekt.zp.ta/mindegy.html,
// Create an instance of the web server
WebServer server(80);
void handleRoot() {
  // Send an HTTP GET request to the AWS S3 bucket for the HTML file
  HTTPClient http;
  http.begin(aws_s3_url);
  int httpCode = http.GET();
  if (httpCode == HTTP_CODE_OK)
  {
    // Read the response from the AWS S3 bucket and send it to the client
    String html = http.getString();
    server.send(200, "text/html", html);
  } else
  {
    // Send an error message to the client if the AWS S3 request fails
    server.send(500, "text/plain", "Failed to download HTML file");
  }
  http.end();
}
void setup()
{
  // Start serial communication for debugging purposes
  Serial.begin(115200);
  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
```

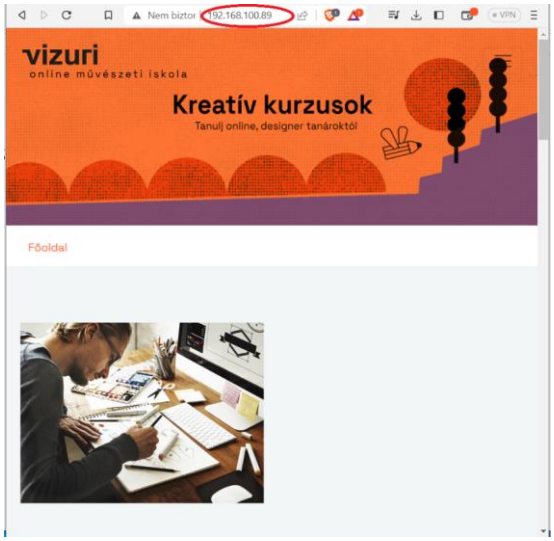
115. ábra ESP32 weboldal kódja (saját ábra)

A projekt kódját az Arduino IDE környezetben írtam meg, amely egy előre kiválasztott projektet futtat, konkrétan a https://vizuri.hu/ weboldalt. Ahhoz, hogy elérjem saját hálózatomat, különböző beállításokat kellett konfigurálnom. Először is meg kellett adnom az ssid részben a saját WiFi hálózatom nevét, amelyhez csatlakozni szeretnék. Ezután meg kellett adnom a jelszót is, hogy sikeresen felcsatlakozhassak a hálózatra. Ezt követően a terminálban kapok egy IP-címet, amit a böngészőbe másolva eljutok a kívánt felületre, vagyis a https://vizuri.hu/ oldalra.



116. ábra ESP32 Hotspotként (saját ábra)

Kihívást jelentett, hogy a WiFi modul rendkívül gyenge és lassú volt. Előzőleg azt terveztem, hogy hotspotként is szeretném használni, hogy kommunikáljon a számítógépemmel, de sajnos nem volt alkalmas erre a feladatra. Ennek következtében felmerült a probléma, hogy a modul teljesítőképessége nem volt elegendő a tervezett hotspot funkció megvalósításához.



117. ábra ESP32 weboldal (saját ábra)

A terminálból visszaadott IP cím beillesztése látható, amelyet mi adtunk meg, és ennek következtében átirányítás történik a kívánt vizuális interfészű weboldalra. Összeségében örülök, hogy sikerült kipróbálni egy mikrokontrolleren miképpen működik ez az egész rendszer, viszont a blokkban komplexebb rendszerre már nem volt alkalmas ennek a weboldalnak a futtatásakor is rengeteg hőt termelt a mikrokontroller, illetve néha nagyon lassan működött ezért gondolom azt, hogy a későbbiekben egy erősebb mikrokontrollert kell erre a célra majd használni.

10. További lehetséges fejlesztések

A projekt további fejlesztéseinek terén még számos izgalmas lehetőség kínálkozik. Az elsődleges szempont a szoftveres rész további optimalizálása, amely magában foglalhatja a grafikus felületek további szebbé tételét, például a termékekről szóló információk beillesztését képekkel. Ezeknek a logikai blokkoknak az logikai összekötését kommunikáljanak egymással a blokkok, illetve a webes felület mindenképpen kapcsolódjon össze a kijelzési funkcióval ugyanis ekkor tudomást szerezhetnénk az áruház készleteiről. Ezenkívül az áruház térképének összekötése egy bevásárló lista funkcióval, amely, megtervezést út megtervezést kínál az áruházon belül ezáltal a legrövidebb úton haladhatunk. Az egyéni felhasználói adatok kezelése is kiemelten fontos terület lehet. A regisztráció és bejelentkezés fejlesztése lehetővé tenné olyan opciók hozzáadását, mint a Google-bejelentkezés, valamint a különböző e-mailes visszaigazoló üzenetek és jelszóelfelejtési lehetőségek. Az adatok tárolásának áthelyezése egy adatbázis szerverre nemcsak biztonságosabbá tenné a rendszert, hanem lehetőséget adna az egyedi felhasználói tárhely kialakítására is, amely fontos a vásárlás közbeni adatoknak a tárolására a felhasználóról, így könnyebben megismerjük a vásárlási szokásainkat és ezek alapján egy vásárlási listát is létrehozhatunk A projekt további skálázhatósága szempontjából érdemes megfontolni a szoftverek multiplatform jellegének erősítését, például mobilalkalmazások kifejlesztésével. Így a felhasználók többféle eszközön, például okostelefonon vagy tableten is könnyen hozzáférhetnének az alkalmazáshoz, tovább növelve ezzel a projekt elérhetőségét és használhatóságát. Emellett pedig komplexen megtervezni a feldolgozó hardvert.

11. Összefoglalás

A szakdolgozatom írása során rádöbbentem arra, hogy a komplex rendszerek tervezése és kivitelezése valójában egy mélyreható műszaki kihívásokkal teli utazás. Az igényeim és mások elvárásainak megértése és összehangolása nem csak egy egyszerű folyamat, hanem egy folyamatos alkalmazkodást igénylő dinamikus folyamat, ahol a szükségletek és az elvárások folyamatosan változnak és egymáshoz igazodnak. Az először saját igényekre, majd mások elvárásaira való fókuszálás nem csupán egy hierarchikus megközelítés, hanem egy iteratív folyamat, amely során az egyszerűnek tűnő igények mögött gyakran bonyolult rendszerek és kapcsolatok rejlenek. A közös metszet megtalálása és a prioritások kijelölése a felmérés, elemzés és hatásvizsgálat eredményeképpen vált lehetővé. A projekt döntéseinek súlya rendkívül nagy, mivel ezek meghatározzák az időtervet, a szoftverek tervezését, a funkciók kidolgozását és azok optimalizálását mind a felhasználói, mind a fejlesztői oldalról. Ezek a döntések nem csupán a technikai részt érintik, hanem szorosan kapcsolódnak a projekt céljaihoz, a felhasználói élmény tervezéséhez és az egész rendszer hosszú távú fenntarthatóságához. A szakdolgozatom fő témája a szoftvertervezés volt, és a grafikus megjelenítésre való hangsúly fektetése azt jelentette, hogy nemcsak az eszközök és technológiák, hanem az esztétika és a felhasználói interakció mélyebb megértése is nélkülözhetetlen volt. Az algoritmusok beépítése, az adatbázis kezelése és a weboldal tervezése olyan rétegek voltak, amelyek a szoftver egészének kifinomultságát növelték. A fejlődés, amelyen keresztül mentem a szoftverek fejlesztése során, nem csak technikai készségeimet növelte, hanem a problémamegoldó képességeimet is fokozta. A blokkvázlatokon alapuló kezdeti elképzelések és tervek néha csak a problémák felszínét karcolták meg, és a fejlesztés során újabb és újabb szempontok, korábban nem gondolt kérdések, illetve problémák merültek fel. A projekt pozitív hatása abban is megnyilvánult, hogy a backend és frontend oldal részletes megismerése által teljes körű képet kaptam a szoftverek működéséről mind a háttérben, mind a felhasználói felületen. A problémák és azok megoldása tovább erősítette a problémamegoldó képességemet és kreativitásomat a fejlesztés során. Végül, a szoftvertervezés és a programozás terén való elmélyülésem nem csak technikai szempontból gazdagított, hanem olyan készségeket is fejlesztett ki bennem, amelyek alkalmassá tesznek a folyamatos fejlődésre és az új kihívások sikeres kezelésére, és rengeteg különböző hasznos fórummal ismerkedtem meg. Az egész folyamat egyfajta szellemi utazás volt, ahol az önként vállalt kihívások és azok megoldása kiszélesítette a látókörömet és elmélyítette a szakmai tudásomat.

12. Summary

Writing my thesis made me realise that designing and implementing complex systems is actually a journey full of profound technical challenges. Understanding and aligning my needs and the expectations of others is not just a simple process, but a dynamic process requiring constant adaptation, where needs and expectations are constantly changing and aligning. Focusing first on one's own needs and then on the expectations of others is not just a hierarchical approach, but an iterative process in which often complex systems and relationships lie behind seemingly simple needs. Finding a common intersection and setting priorities is the result of survey, analysis and impact assessment. Project decisions carry a great deal of weight, as they determine the timeline, software design, feature development and optimisation from both the user and developer side. These decisions are not only technical, but are closely linked to the project's objectives, the design of the user experience and the long-term sustainability of the whole system. The main focus of my thesis was software design, and the emphasis on graphic design meant that a deeper understanding of not only the tools and technologies, but also the aesthetics and user interaction was essential. The incorporation of algorithms, database management and website design were layers that added to the sophistication of the software as a whole. The progression I went through in developing software not only increased my technical skills, but also enhanced my problem solving abilities. Initial ideas and designs based on block diagrams sometimes only scratched the surface of problems, and as development progressed, new aspects and previously unthought of issues and problems arose. The positive impact of the project was also reflected in the fact that by getting to know the backend and frontend in detail, I was able to get a complete picture of how the software works, both in the backend and in the user interface. The problems and their solutions further strengthened my problem solving skills and creativity during the development. Finally, my immersion in software design and programming has not only enriched me from a technical point of view, but has also developed skills that enable me to continuously improve and successfully tackle new challenges, and has exposed me to a wide variety of useful forums. The whole process has been an intellectual journey, where the challenges I have volunteered to take on and solve have broadened my horizons and deepened my professional knowledge.

13. Irodalom jegyzék

[1] https://www.forbes.com/sites/pamdanziger/2023/08/03/smart-carts-may-be-thedisruptive-technology-grocery-stores-need-now/?sh=6ce0945c4588 (2022.10.05)

[2] https://scaletronicglobal.com/what-does-legal-for-trade-mean/ (2023.10.05)

[3] https://www.cust2mate.com/shopper/ (2023.10.10)

[4] https://www.caper.ai/ (2023.10.10)

[5] https://mashable.com/article/amazon-dash-carts-shopping?europe=true (2023.10.10)

[6] https://docs.google.com/forms/d/1cZxhgx2aXbCQX1cdka9dN_SaIdL2S1lWP5LisD8sUA/edit#responses (2022.10.30)

[7] https://www.codeconquest.com/blog/code-documentation-tools-and-techniques/ (2023.10.13)

[8] https://realpython.com/python-gui-tkinter/ (2023.10.20)

[9] https://hu.wikipedia.org/wiki/Dijkstra-algoritmus (2023.11.07)

[10] https://dev.mysql.com/doc/connector-python/en/connector-python-example-cursor-transaction.html (2023.11.23)

[11] https://github.com/codebasics/python_projects_grocery_webapp (2023.11.23)

[12] https://www.microcontroller.hu/termek/esp32-wifi-bt-mikrokontroller-38-pin/ (2023.11.25)

14. Ábrajegyzék