A képen szöveg látható

Automatikusan generált leírás

**Borok értékelésére szolgáló online alkalmazás**

**Szakdolgozat**

**Készítette:**

**Név:** Juhász Gábor

**Neptunkód:** BN9G72

**Szak:** Mérnök Informatikus BSc

Korszerű web technológiák szakirány

**Témavezető:** Tompa Tamás

Tartalom

[1. Bevezető 3](#_Toc87559466)

[2. Technológiák áttekintése 6](#_Toc87559467)

[2.1. Motiváció 6](#_Toc87559468)

[2.2. Kutatás 7](#_Toc87559469)

[3. Előkészületek 9](#_Toc87559470)

[3.1. A feladat ismertetése 10](#_Toc87559471)

[3.2. Megoldás alternatívák 10](#_Toc87559472)

[3.3. Megoldás kiválasztása 13](#_Toc87559473)

[4. A megvalósítás eszközei 13](#_Toc87559474)

[4.1. A tervezéshez használt eszközök 13](#_Toc87559475)

[4.2. A fejlesztőkörnyezet kialakítása 14](#_Toc87559476)

[4.2.1. Node.js 14](#_Toc87559477)

[4.2.2. Angular CLI 15](#_Toc87559478)

[4.2.3. Visual Studio Code 16](#_Toc87559479)

[4.2.4. Express.js 16](#_Toc87559480)

[4.2.5. MySQL és phpMyAdmin 17](#_Toc87559481)

[4.2.6. Github 18](#_Toc87559482)

[5. Az alkalmazás tervezése 18](#_Toc87559483)

[5.1. Az alkalmazás funkciói 18](#_Toc87559484)

[5.2. Kezdeti képernyőtervek 22](#_Toc87559485)

[6. Az adatbázis ismertetése 23](#_Toc87559486)

[6.1. Wine.ts 24](#_Toc87559487)

[6.2. User.ts 25](#_Toc87559488)

[6.3. Review.ts 26](#_Toc87559489)

[6.4. Roles.ts 27](#_Toc87559490)

[6.5. Cart.ts 27](#_Toc87559491)

[6.6. CartItem.ts 28](#_Toc87559492)

[6.7. Order.ts 29](#_Toc87559493)

[6.8. Carousel.ts 29](#_Toc87559494)

[7. Alkalmazás fejlesztése 30](#_Toc87559495)

[7.1. Mappastruktúra kialakítása 31](#_Toc87559496)

[7.2. *Route-*ok és *Controller*-ek létrehozása 32](#_Toc87559497)

[7.3. *Service*-ek és *model*-ek létrehozása 33](#_Toc87559498)

[7.4. Autentikációs modul 33](#_Toc87559499)

[7.5. Admin modul 35](#_Toc87559500)

[7.5.1. Borok kezelése 36](#_Toc87559501)

[7.5.2. Carousel kezelése 37](#_Toc87559502)

[7.5.3. Felhasználók kezelése 39](#_Toc87559503)

[7.5.4. Rendelések kezelése 41](#_Toc87559504)

[7.6. Borok megjelenítése és szűrése 41](#_Toc87559505)

[7.7. Kosár és rendelés 44](#_Toc87559506)

[7.8. Dizájn létrehozása 44](#_Toc87559507)

1. Bevezető

A feladat célja egy olyan online alkalmazás tervezése, implementálása, amely alkalmas lehet borok értékelését megvalósítani, az által, hogy a regisztrált felhasználók az általuk kóstolt, illetve vásárolt borokról részletes értékelést tudnak adni (képek, vélemények, valamint előre meghatározott tulajdonságok alapján).

Az alkalmazás tematikájából adódóan a felhasználók kizárólag a 18 éven felüli korosztályból kerülhetnek ki. A szoftver leendő használói olyan borkedvelők, akik bort szeretnének vásárolni, és a választáshoz szívesen felhasználják a többi vásárló véleményét, esetleg maguk is értékelnek borokat. A webshop üzemeltetési funkcióit használó user-ek jellemzően az értékesítés, beszerzés területről kerülnek majd ki, akik a beérkezett rendeléseket kezelik, és a készleteket karbantartják.

Funkciók tekintetében szétválnak az adminisztrátori és a felhasználói szerepkörök funkciói. **Az adminisztrátori funkciók** az alkalmazás karbantartására szolgálnak beleértve a felhasználókezelést, és a webshop modullal kapcsolatos feladatokat. utóbbiak lehetőséget biztosítanak a készletek kezelésére, a rendelések menedzselésére.

**A felhasználói funkciók**, az oldal látogatói számára érhetők el. Minden látogatónak lehetősége van a borok, és értékeléseik között keresni, valamint a borok között tallózni, és az oldalra regisztrálni. A regisztrált felhasználók elérik a vásárlással kapcsolatos funkciókat. Az oldal értékelés modulja azon látogatók számára érhető el, akik regisztráltak, és adott borból már vásároltak. Az értékelés komponensen belül a borok kapnak a kóstolóktól egy 1-5 közötti pontszámot, ami a felhasználó saját véleményét tükrözi a borról. Ezen felül lehetőség van a borokat tulajdonságaik mentén is értékelni (pl: savasság, testesség stb...). A szoftver a kapott pontszámok átlagát megjeleníti az alapadatok mellett. Az oldalon helyet kap egy kereső modul, ami a szabadszavas keresésen kívül lehetőséget biztosít összetett keresésre, néhány borjellemző mentén.

A szoftver minden funkciója böngészőn keresztül érhető el. A látogatók számára megjelenített oldalak mindegyike reszponzív működésű, és a modern webdizájn trendeknek megfelelően animált elemeket is tartalmaz. Az adminisztrátori oldalak tekintetében inkább a funkciók felhasználóbarát működése, és könnyű elérése a lényeges, a modern megjelenéssel szemben.

A feladat megvalósításához mindenképpen olyan technológiát szeretnék felhasználni, ami alkalmas robosztus alkalmazások implementálására. Az aktuális webfejlesztési trendeket, és az alkalmazás leendő funkcionalitását figyelembe véve, az egyik JavaScript alapú keretrendszer használatát tartom célszerűnek. Ma a három legnépszerűbb ilyen technológia az Angular, React és Vue.js. Mindhárom framework/library alkalmas hasonló alkalmazások implementálására, viszont a választásom az Angular-ra esett, mivel a rendkívül kiterjedt beépített eszközkészlete által komplex megoldást kínál a feladat elkészítéséhez. A backend oldali funkciók létrehozásához a Express keretrendszert választottam, amely egy MySQL adatbázissal történő kommunikációt fog megvalósítani. Az Angular és az Express futtatásáról Node.js, platformfüggetlen szerverkörnyezet gondoskodik.

A feladat készítésekor a fenti technológiákból a legfrissebb verziószámokat használom, amely jelenleg:

Node.js: 14.15

Angular: 11

Express 4.17

MySQL 8

|  |  |
| --- | --- |
| **Részfeladat** | **Megvalósíthatóság** |
| Az alkalmazás funkcióinak specifikálása | érthető a feladatrész, a jelenlegi tudásanyaggal megoldható |
| A funkciók közötti kapcsolatok megállapítása | érthető a feladatrész, a jelenlegi tudásanyaggal megoldható |
| Az alkalmazás egyes osztályainak specifikálása | érthető a feladatrész, a jelenlegi tudásanyaggal megoldható |
| Az osztályok közötti kapcsolatok megállapítása | érthető a feladatrész, a jelenlegi tudásanyaggal megoldható |
| Adatbázisséma térehozása | érthető a feladatrész, a jelenlegi tudásanyaggal megoldható |

Az online kereskedelem mértéke az elmúlt években folyamatosan növekvő tendenciát mutat, és ez a trend a következő években várhatóan tovább folytatódik. A járványhelyzet miatti korlátozások következtében a hagyományos kiskereskedelmi egységek forgalma jelentősen visszaesett, ezért minden üzlet igyekezett megjelenni az online kereskedelem területén. Ez a helyzet is rávilágított arra, hogy napjainkban egy vállalkozásnak egyre nagyobb figyelmet kell fordítania az online jelenlétre, tevékenységtől függetlenül.

1. ábra: Csomagküldő és internetes kiskereskedelmi forgalom havonta [1]

Felhasználói szemszögből az online vásárlások egyik legnagyobb problémája, hogy nincs fizikai kapcsolat az áruval. Ezt ellensúlyozadnó, a webshopok üzemeltetői kellően részletes leírást, fotókat és videókat próbálnak közzétenni a termékekről. A vásárlóknak ennek ellenére más felhasználók véleményi, értékelései nyújthatják az egyik legnagyobb segítséget a döntésben.

Szakdolgozatomban egy borokat árusító webshop megvalósítását tűztem ki célul, melyben a vásárlók döntéseit a többi vásárló értékelései és leírásai segítik. A témaválasztásnál a fent leírtakon kívül fontos szerepet játszottak borász ismerőseimmel folytatott beszélgetéseim, melyekből sok ötlethez jutottam az értékelő modullal kapcsolatban.

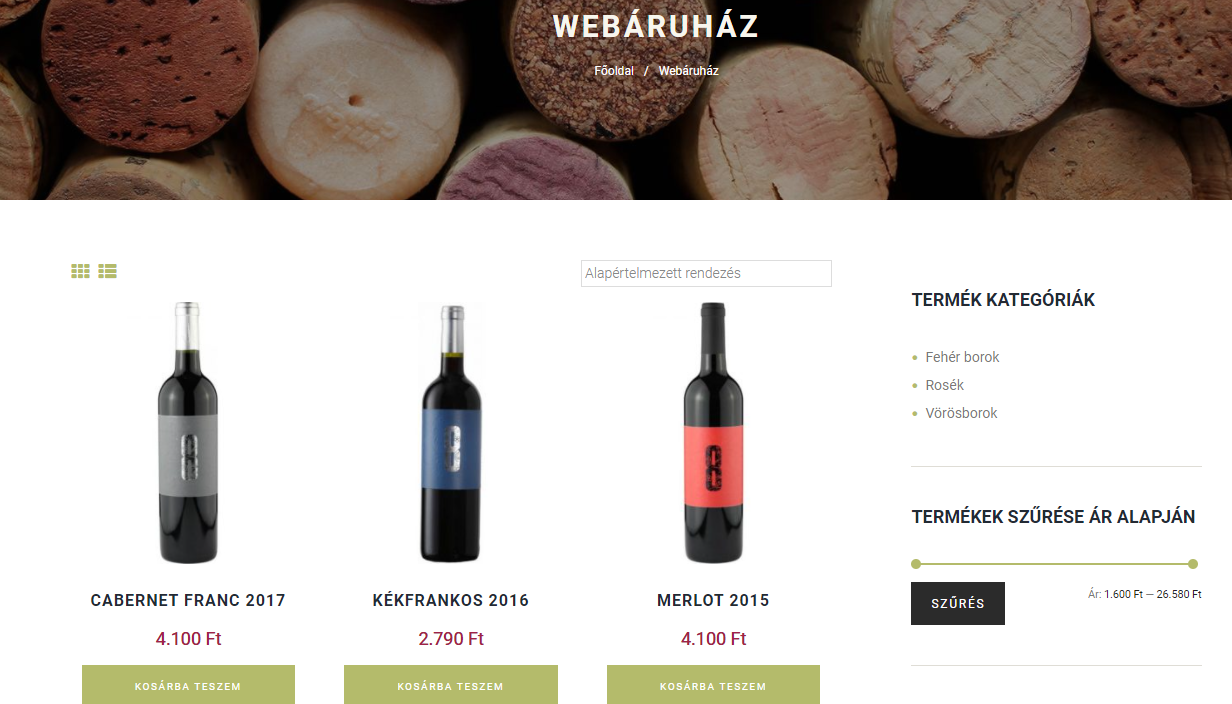
1. Technológiák áttekintése

## Motiváció

Fiatal borász ismerőseim gyakran megosztják velem beszélgetéseink során, hogy milyen ötletekben, újításokban látnak lehetőséget arra, hogy közelebb hozzák a boraikhoz az embereket. Ezeknek az ötleteknek jelentős része az értékesítéssel függ össze. Tapasztalataim azt mutatják, hogy a fiatalabb borászok az informatikára, és az infokommunikációra egyértelműen olyan eszközként tekintenek, amelyek segítségével a termékeiket meg tudják ismertetni a leendő vásárlókkal. Kivétel nélkül próbálnak folyamatosan jelen lenni a különböző közösségi oldalakon, és időről időre megújuló tartalmakat szolgáltatnak a követőiknek, melyek középpontjában a boraik állnak. Évek óta egyre nagyobb hangsúlyt fektetnek az online értékesítésre, aminek fontosságát a világjárvány miatti bezárások sajnos igazolták. Az internetes értékesítés legkézenfekvőbb eszköze egy webshop indítása, amire számtalan lehetőség áll rendelkezésre. Ezekről a „Megoldás alternatívák” című fejezetben részletesebben írok. A társalgások konklúziójaként azt állapítottuk meg, hogy egy webshop elengedhetetlen eszköze az online jelenlétnek, viszont önmagában, a vásárlók számára nem nyújt különösebb segítséget a vásárlásban. Az ötletek között szerepelt, egy olyan modul, amely a borok értékelését egy előzetesen, szakmai szempontok alapján összeállított lista szerint teszi lehetővé a felhasználóknak. A lista kellően részletes lenne ahhoz, hogy a bor minden tulajdonságát meg lehessen benne adni, és szabadszöveges mezőkkel akár hosszabb véleményt is lehessen írni egy- egy tételről.

## Kutatás

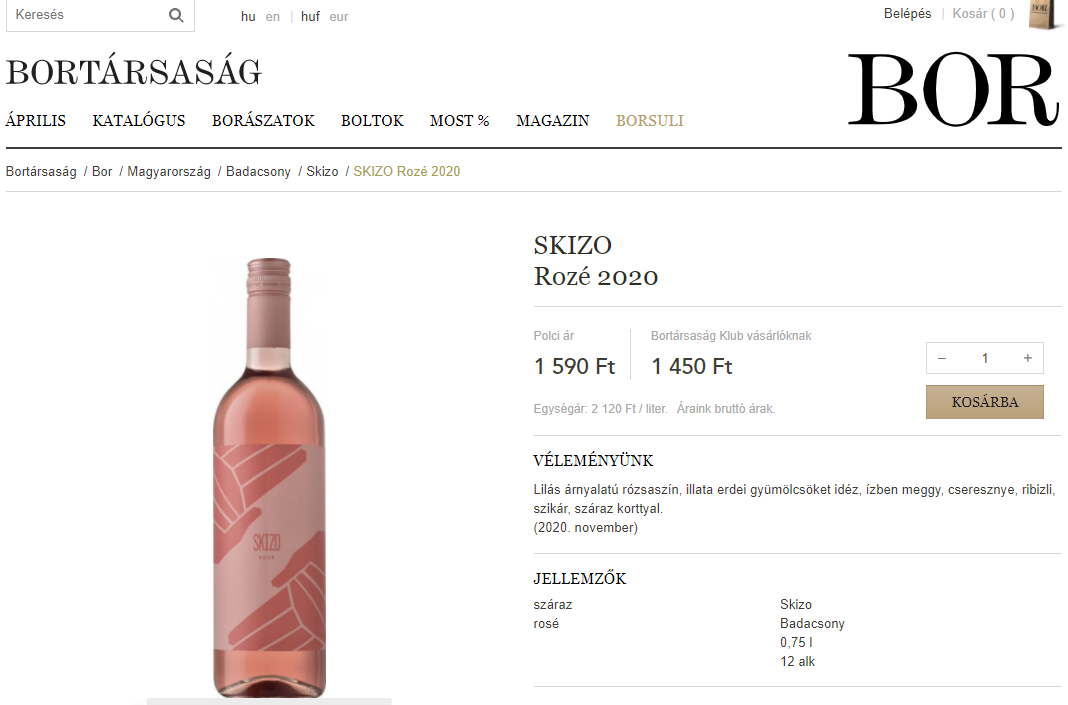
Kutatásomat a bor témájú magyar webshopok összehasonlításával kezdtem. A piacra jellemző, hogy szinte az összes pincészet rendelkezik saját webshoppal, melyeket jellemzően a pincészet saját bemutatkozó weblapjáról lehet elérni, és értelemszerűen csak a termelő saját borait lehet rajta keresztül megvásárolni. Ezek többnyire megvalósítják az összes lényeges webshop funkciót, lehetőséget biztosítanak a borok közötti keresésre, és nézelődésre.



2. ábra: Nyolcas Pincészet webshopja [2]

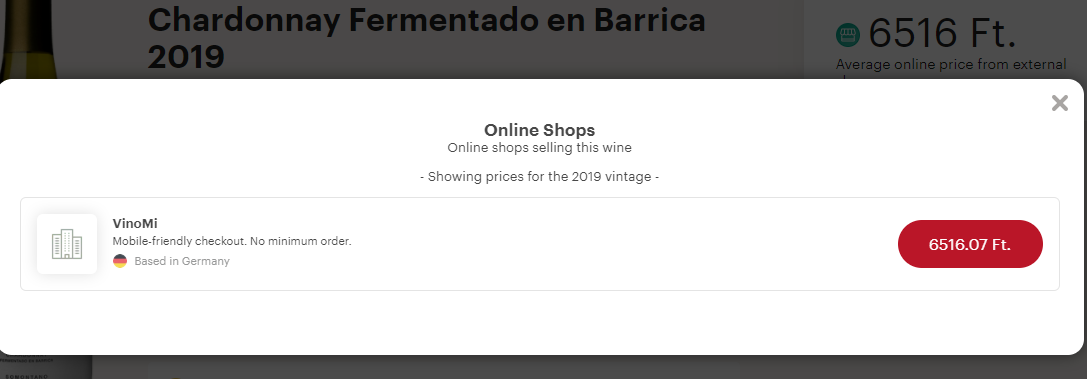
<https://nyolcasbor.shop.hu/webaruhaz/> [2]

A piacon helyet foglalnak még a kereskedők által üzemeltetett online áruházak. Ezekre jellemző, hogy számos pincészet termékeit kínálják, sok esetben a borokon kívül más népszerű termékek is a kínálatuk részét képezik. Magyarországon az egyik legnépszerűbb ilyen webshop a Bortársaság [3] online felülete. Az e-commerce funkciókat nagyon jól egészíti ki gyakran megújuló minőségi, leginkább borászati témájú tartalmakkal. Az egyes termékekről rövidebb leírást tesznek közzé fényképekkel, ugyanakkor felhasználók nem írhatnak értékelést a borokhoz.



3. ábra: Bortársaság webshop

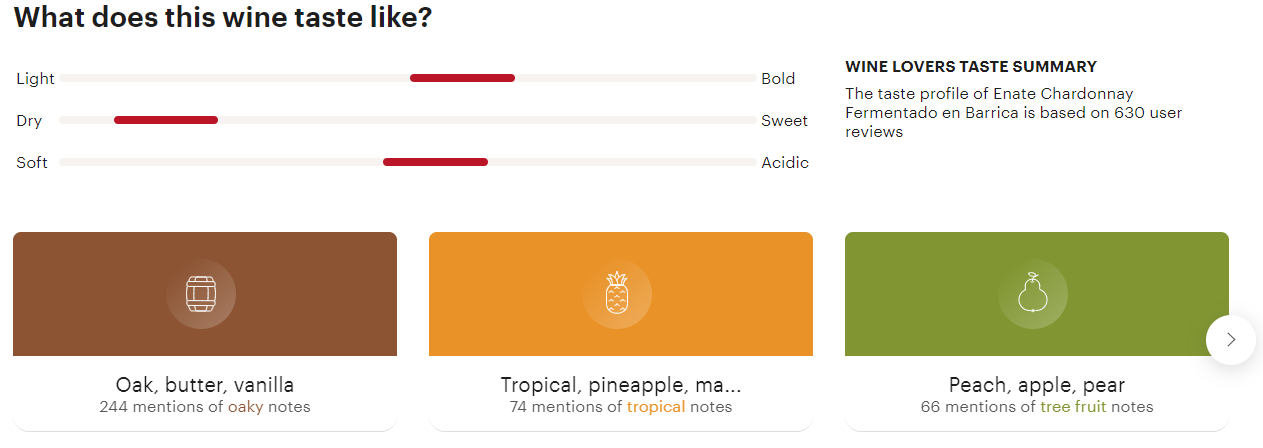
Külföldi oldalak között a vivino.com [4] oldalát találtam legkiemelkedőbbnek a témában. Ennek az oldalnak egyik érdekessége, hogy nincs saját webshop része, hanem az egyes borokhoz külső forgalmazókat lehet megadni, akiknek az aktuális áraik alapján kalkulál egy átlag árat az alkalmazás, és tájékoztatja az érdeklődőt, hogy mely webshopokban lehet megvenni a terméket.



4. ábra: vivino.com – külső webshop ajánlás

vivino.com – külső webshop ajánlás

További érdekesség, hogy a borok értékelését szinte teljes egészében a felhasználókra bízza. Szakmai szempontok szerint nagyon jól megadott tulajdonságok mentén lehet osztályozni az egyes tételeket, melyeket az összes felhasználó megtekinthet.



5. ábra: vivino.com – bor értékelés

1. Előkészületek

Ez a fejezet mutatja be a feladat megoldásához milyen lehetőségek állnak rendelkezésre. Összehasonlítom a meglévő technológiákat, fejlesztőeszközöket, melyek közül kiválasztom a megvalósításhoz leginkább alkalmas módszert.

## A feladat ismertetése

A kitűzött cél megvalósításának érdekében szükséges a feladatot jobban átgondolni, meghatározni. Ezek után lehetséges csak az információk gyűjtése, és technológiák, módszerek összehasonlítása.

Borász ismerőseimmel folytatott beszélgetéseim alkalmával többször szóba került a borok online értékesítése. Napjainkban erre számos lehetőség kínálkozik, melyeket ennek a fejezetnek a későbbi részében fogok bemutatni. Ennek kapcsán merült fel a kérdés, hogy hogyan lehetne a leendő vásárlókat segíteni a választásban, hiszen a klasszikus vásárlástól eltérően, online térben még a palack forgatásának lehetősége sem adott a vásárló számára. A fényképek, és minél részletesebb leírások közzétételén túl felmerült ötletként egy közösségi funkció, melyen keresztül a felhasználók egy adott borról a saját tapasztalataikat osztják meg. A webshop, és közösségi funkciók pontosítását követően született meg az igény, melyben egy olyan online vásárlási felület szerepelt, ami rendelkezik egy közösségi modullal, amin keresztül azok a felhasználók, akik már vásároltak egy adott borból, értékeléseket tehetnek közzé az érintett termékkel kapcsolatban. A vásárlási kritériumot fontosnak tartottuk beépíteni, mert hiányában könnyen előfordulhatna, hogy olyan felhasználók is értékelnek, akik még soha nem fogyasztottak a borból, ezzel torzítva a valóban hiteles értékeléseket. Igényként fogalmazódott meg továbbá, hogy az alkalmazásnak meg kell valósítania a modern webshop funkciókat, beleértve a termékek, felhasználók és rendelések menedzselését, valamint a látogatók számára elérhető felületek felhasználóbarát és modern megjelenését.

## Megoldás alternatívák

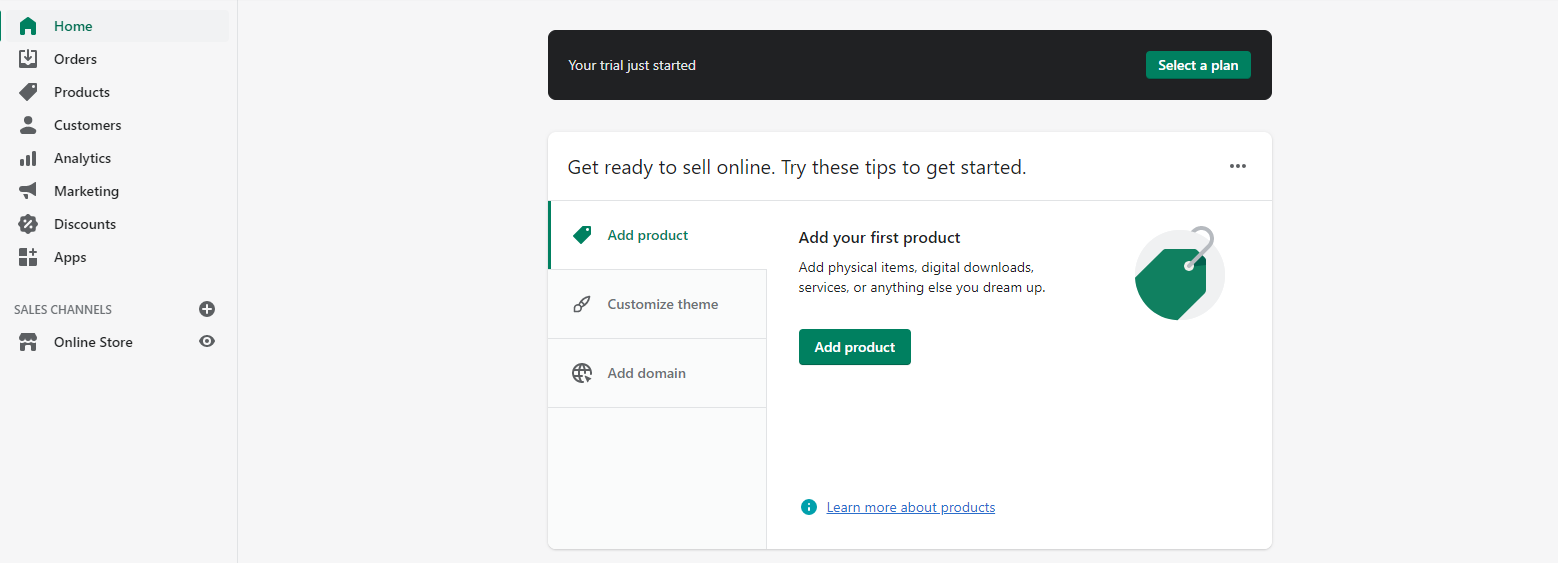
Az igények megismerése után egyértelműen látszik, hogy a megvalósításhoz nem elegendő statikus, kliens oldali technológiák használata, mint például HTML, CSS, JavaScript. Mindenképpen adatbázishátterű, tartalmakat dinamikusan megjeleníteni képes módszereket szükséges használni.

**Kész webshopok**

Első lépésben kész e-commerce megoldások után néztem. Webshop indítására számos lehetőség kínálkozik napjainkban. A szolgáltatások nagyon hasonlóak, regisztráció után kapunk egy működő webshop felületet, benne a legfontosabb funkciókkal. A termékek feltöltése és a szükséges beállítások elvégzése után tulajdonképpen egy működő online bolt áll a rendelkezésünkre, melyhez különböző kiegészítő szolgáltatások állnak rendelkezésre. Ilyen például a különböző dizájnok alkalmazása, vásárlási statisztikák mutatása és egyéb, például blog modul implementálása. Ezek a szolgáltatások nagy része fizetős.

Az egyik legnépszerűbb, és legszélesebb körben használt ilyen szolgáltatás a <https://www.shopify.com/> . Előnye, hogy minden modern webshop funkciót biztosítja a felhasználónak, és különböző plugin-eken keresztül számos extra funkció is elérhető.

Hátránya, hogy ezeknek a plugin-eknek a nagy része fizetős, és jellemzően third party termékek. Hátrány még, hogy az igényként megjelölt értékelő modulra nem nyújt megoldást.



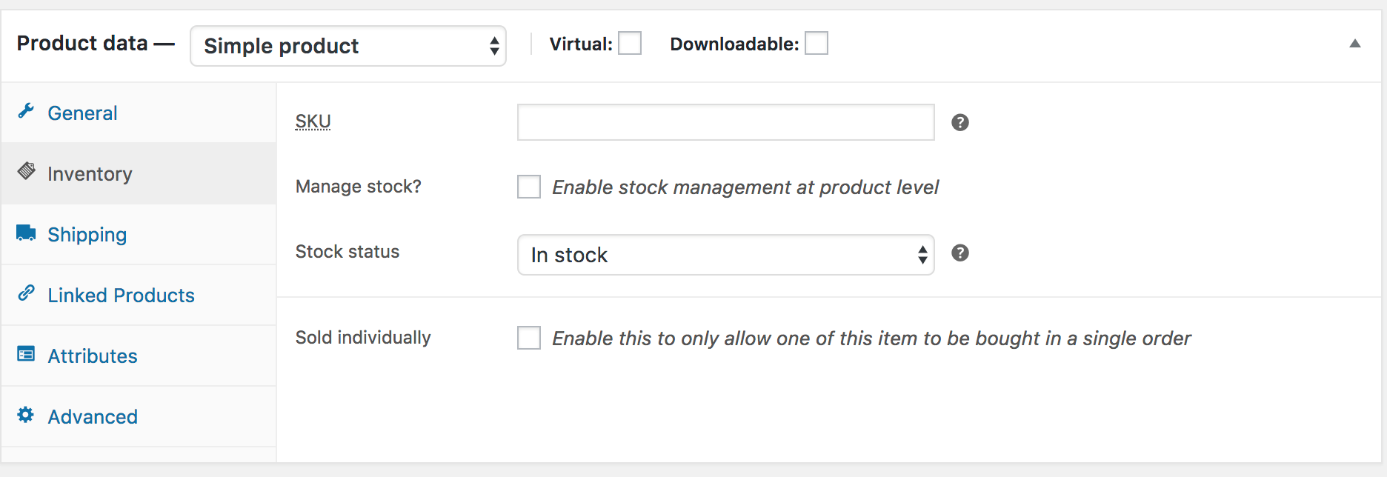
6. ábra: https://www.shopify.com/ admin felülete [5]

A gyors megoldást nyújtó instant e-commerce szolgáltatások tanulmányozása után megállapítottam, hogy olyan megoldásra lesz szükség, ami kellő rugalmasságot biztosít a beépülő modulok terén.

**PHP alapú tartalomkezelők**

Következő megoldási lehetőségként egy WordPress vagy Drupal alapú alkalmazás létrehozását vizsgáltam. A két tartalomkezelő működése rendkívül hasonló, ezért csak a WordPress-en keresztül mutatom be a tapasztalataimat. Mindkét CMS PHP alapú, jellemzően MySql adatbázis hátterű rendszer. Implementálásuk nagyon egyszerű, PHP környezet, és támogatott adatbázis szükséges hozzá. A telepítő közérthető módon végig viszi a felhasználót a folyamaton. A szükséges konfigurációs paraméterek (adatbázis adatok, mappabeállítások stb.) beállítása után rendelkezésünkre áll a kész tartalomkezelő, ami lehetőséget biztosít akár wysiwyg szerkesztő használatával is tartalmak létrehozására és publikálására. A shopify-hoz hasonlóan itt is rendelkezésünkre áll számos third party plugin, melyeknek egy része itt is fizetős. Nagy előnye viszont, hogy saját magunk is fejleszthetünk PHP nyelven különböző modulokat, és az oldalunk kinézetét is teljesen szabadon alakíthatjuk. Népszerű e-commerce modulja a WooCommerce, ami minden modern webshop funkciót biztosít. Kezelése a WordPress CMS-en belülről történik, és nagyon felhasználóbarát módon, számos beállítási lehetőséget biztosít.

Hátránya, hogy a third party modulokon keresztül nagyon sok sérülékenységet tartalmaz, és bizonyos webshop feature-ök használata fizetős.



7. ábra: WooCommerce admin felülete WordPress CMS-ben (forrás: https://docs.woocommerce.com/) [6]

A PHP alapú tartalomkezelők nyújtotta lehetőségek tanulmányozása során megállapítottam, hogy ezzel a technológiával megvalósítható lenne a feladat. Ugyanakkor hátrány, hogy a harmadik féltől származó modulok komoly sebezhetőségi felületet biztosítanak az alkalmazás számára. Hátrány még, hogy a PHP technológia, a JavaScript alapú keretrendszerek megjelenésével, mint például az Node.js, elkezdett háttérbe szorulni az utóbbi években.

**Saját fejlesztés modern keretrendszerben**

Harmadik megoldási lehetőségként vizsgáltam egy teljesen új alkalmazás fejlesztését. Ennek előnye, hogy már a tervezés folyamatától kezdve rálátásom van minden modul fejlesztésére. Nem kell egy meglévő rendszer sajátosságait figyelembe véve kialakítani az egyedi funkciókat. További előny még, hogy ebben az esetben magam választhatom meg az összes technológiát, amivel a feladatot meg szeretném valósítani.

Ebben az esetben mindenképpen egy modern, JavaScript alapú keretrendszer választását tartom célszerűnek. Ezek a megoldások skálázhatóságuk és teljesítmény béli mutatóik miatt az utóbbi években dominálnak webfejlesztés területen.

Napjainkban a három leginkább elterjed ilyen technológia az Angular, React és Vue.js. A három módszert összehasonlítva az Angular-t találtam a legalkalmasabbnak arra, hogy a fejezet elején megfogalmazott igények szerint kialakított alkalmazást hozzak létre.

## Megoldás kiválasztása

A fent részletezett megoldási lehetőségek elemzését követően arra a megállapításra jutottam, hogy a feladat megvalósítása egy saját fejlesztésű alkalmazás keretein belül valósulhat meg a leghatékonyabban.

1. A megvalósítás eszközei

Ebben a fejezetben bemutatom, hogy milyen hardver és szoftver eszközökre van szükség ahhoz, hogy a célként kitűzött webshop-ot, a felmerült igényekkel együtt meg lehessen valósítani.

## A tervezéshez használt eszközök

Az alkalmazás implementálását mindenképpen a tervezési fázisnak kell megelőznie. Egy jól átgondolt, és elkészített rendszerterv segítségével az implementálás is lényegesen hatékonyabban hajtható végre. A fejlesztési szakaszban felbukkanó esetleges hiányosságok, anomáliák kezelése sokkal több erőforrásráfordítást igényel, mintha ezek a hibák a tervezési szakaszban derülnének ki. Ezért is érdemes nagy hangsúlyt fektetni a rendszerterv kidolgozására.

A tervezés során UML diagramok segítségével fogom szemléltetni a rendszer tervezett működését. Az UML modellező nyelvvel grafikus módon lehet több aspektusból bemutatni akár a teljes alkalmazás, akár annak egyes alrendszereinek viselkedését.

A diagramok elkészítéséhez számtalan program áll rendelkezésre. Többségük, funkcionalitását tekintve szinte teljesen megegyezik. Én a <https://app.diagrams.net/> (régebbi nevén draw.io) online alkalmazást választottam erre a célra. A létrehozott diagramokat a saját gépünkre is le tudjuk menteni, majd onnan visszatöltve böngészőn keresztül folytathatjuk a munkát. Az összes UML diagramtípus elemeit tartalmazza, emellett rendkívül felhasználóbarát szerkesztőfelületet biztosít.

## A fejlesztőkörnyezet kialakítása

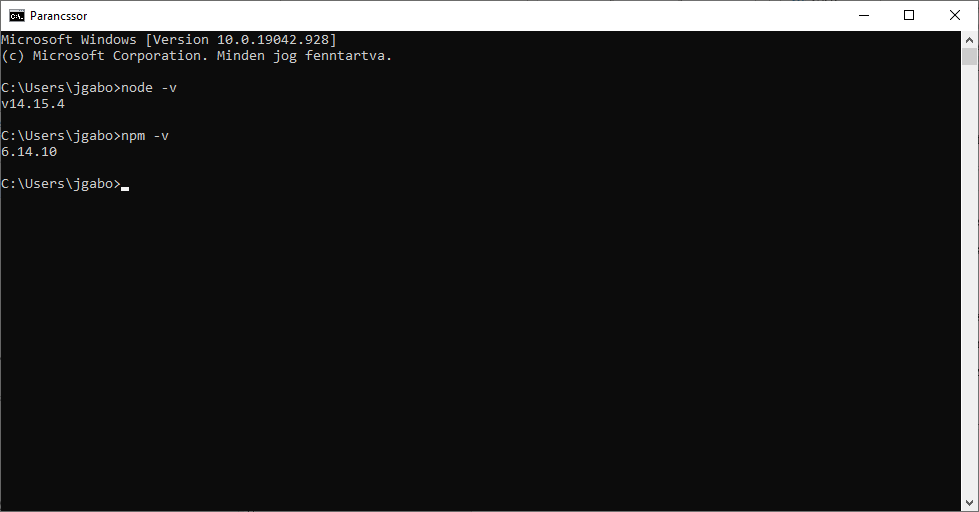
Az előző fejezetben azt a döntést hoztam, hogy Angular keretrendszerben szeretném megvalósítani a feladatot. A fejlesztői környezet kialakításához szükséges alkalmazások telepítését, beállítását az alábbiakban fogom bemutatni. A fejlesztést Windows 10 operációs rendszeren fogom végezni, ezért a szükséges szoftverek telepítését és konfigurálását ennek megfelelően fogom bemutatni.

A fejezetben nem fogom minden komponens telepítését részletezni, mivel ezek túlnyomó része az npm csomagkezelőn keresztüli telepítéssel történik a megszokott npm install paranccsal. Az alábbiakban az alkalmazásom gerincét alkotó főbb egységek telepítését és beállítását fogom bemutatni, valamint az alkalmazott segédprogramok installálását és konfigurálását.

### Node.js

Első lépésként a Node.js alkalmazást szükséges telepíteni, amely egy szerver oldali platform. Segítségével JavaScript kódot lehet futtatni szerver környezetben. A hivatalos dokumentáció szerint a Node.js egy olyan platform, amely a JavaScript futtatójára épül, a gyors és skálázható hálózati alkalmazások egyszerű létrehozását teszi lehetővé. A Node.js egy eseményvezérelt, nem blokkoló I /O modellt használ, tehát aszinkron. Adatintenzív, valós idejű alkalmazások szerver oldali támogatására jól használható.

Telepítése a hivatalos oldalon található telepítőfájl legfrissebb stabil verziójának letöltésével indul. A telepítő használata egyértelmű, az alapbeállításokat használva sikeres telepítés várható. A telepítőcsomag tartalmazza a Node.js alkalmazást, és az npm csomagkezelőt, mely segítségével JavaScript csomagokat tudunk majd hozzáadni a projektünkhöz. Az installálás végeztével parancssorban a node -v, és npm -v parancsok futtatásával tesztelhetjük, hogy sikeres volt-e a telepítés.

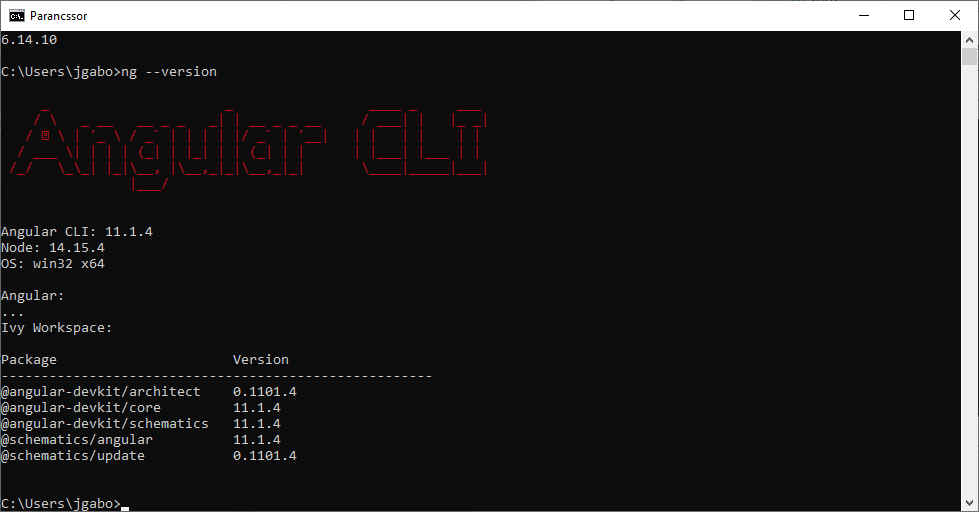


Node.js telepítés ellenőrzése

### Angular CLI

Az Angular CLI tulajdonképpen egy parancssori eszköz, mellyel az Angular alapú alkalmazások fejlesztése történik.

Telepítése globálisan az npm install -g @angular/cli paranccsal történik, melynek sikerességét a verzió lekérdezésével ellenőrizhetjük a ng –version utasítással.

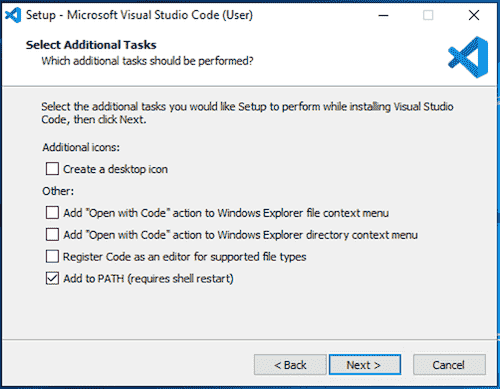


Angular CLI telepítésének ellenőrzése

### Visual Studio Code

A Visual Studio Code a Microsoft által fejlesztett ingyenes kódszerkesztő. Angular projektek fejlesztéséhez sokan használják. Az én választásom is erre a programra esett, mert számos kiegészítővel támogatja a munkát.

A hivatalos oldalról (<https://code.visualstudio.com/>) letöltött legújabb stabil verzió telepítőjét futtatva, a felajánlott beállításokat alkalmazva történik az telepítés. Az „Add to PATH” checkbox bejelölését érdemes ellenőrizni, ezen felül a többi beállítás magától értetődő.



„Add to PATH” opció ellenőrzése

### Express.js

Az alkalmazás szerver oldali kiszolgálását az Express framework fogja végezni, aminek a telepítése az npm csomagkezelőn keresztül történik.

A telepítés előtt létrehozunk az alkalmazás backend részének egy külön mappát a projektstruktúrában, hogy szeparáljuk a szerver és kliens oldali komponenseket. A mappába lépve az alábbi parancsokkal végezzük a telepítést, mely az Express-en kívül telepíti a body-parser komponenst is, mely a bejövő HTTP POST request-ek feldolgozásában lesz majd segítségünkre. A tsc -init parancs hozza létre a tsconfig.json fájlt.

tsc –init

npm install @types/node express @types/express body-parser –save

A telepítés után szükséges létrehozni és konfigurálni az index.ts fájlt, ami többek közt a szerver indítását hivatott végrehajtani. A backend/src mappában hoztam létre a fenti fájlt. Fontos beállítás, hogy a szerver melyik portot használja. Én a 3000-es portot állítottam be, amit az alábbi kódrészlet eredményez:

1. app.listen(3000, () => {
2. console.log('Listening on 3000 ...');
3. });

Látható, hogy sikeres indítás után az alapértelmezett konzolra a „Listening on 3000…” szöveg kerül kiírásra, melyet a ts-node-dev src/index.ts parancs parancssorban történő futtatása után ellenőrizni is tudjuk:



Elindult a szerver

### MySQL és phpMyAdmin

A programom adatbázis-hátterét a MySQL 5.7.31-as verziója biztosítja. Telepítése történhet a hivatalos oldalról letöltött telepítőn keresztül, azonban a későbbiekben az adatbázis tábláinak könnyebb áttekinthetősége kedvéért a phpMyAdmin nevű segédprogramot fogom használni. Ezért a két alkalmazás telepítését és konfigurálását célszerűnek tartom olyan szoftveren keresztül, amely tartalmazza a fent említett programokat, és közös telepítőn keresztül történik a telepítés. Több ilyen program érhető el az interneten. Én a WampServer nevűt választottam. A telepítő egyértelműen végigvezet a szükséges konfigurációs lépéseken. Befejeztével rendelkezésünkre áll a MySQL és a phpMyAdmin, amiknek segítségével könnyen létre lehet hozni az induló adatbázist.

### GitHub Desktop

A verziókövetés miatt kezdetektől egy GitHub repository-ban dolgoztam, amihez letölthető egy asztali kliens, nagyban megkönnyítve ezzel a használatot. A telepítés rendkívül egyszerű, csupán egy élő GitHub regisztráció kell hozzá. [**irodalom**]

1. Az alkalmazás tervezése

Szakdolgozatom ezen részében kerül dokumentálásra az alkalmazás tervezése. A tervezés során létrehozott diagramokat szintén ez a fejezet fogja tartalmazni.

A szoftver célja egy olyan borokat áruló webshop létrehozása, amely megvalósítja egy modern online áruház fontosabb funkcióit, ugyanakkor implementálásra kerül egy borokhoz kapcsolódó értékelő modul, melyben azok a felhasználók tudnak értékeléseket rögzíteni, akik már vásároltak bort.

A szoftverrel szemben támasztott elvárások:

* Autentikációs funkciók
* Jogosultságkezelés
* Webshop funkciók biztosítása
* Készletkezelő funkciók megvalósítása
* Borok közti összetett keresés
* Felhasználó saját profiljának kezelése
* Felhasználó saját értékeléseinek kezelése
* Felhasználóbarát megjelenés

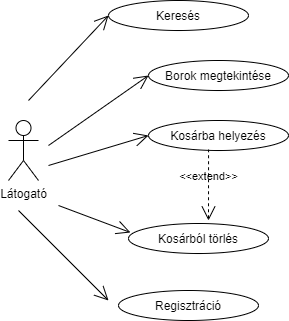
## Az alkalmazás funkciói

A szoftver funkcióinak reprezentálásár a use case diagramot fogom használni. Erre a diagramra jellemző, hogy könnyen áttekinthetően és szemléletesen lehet rajta keresztül bemutatni a rendszer funkcióit. A diagram elemei a relációk, aktorok és use case-ek. A aktorok egy szerepkört jelölnek a rendszerrel kapcsolatba kerülő felhasználókat reprezentálják. A use case-ek írják le az alkalmazás funkcióit. A relációk pedig az aktorok és a use case-ek közötti kapcsolatot jelölik.

A rendszerben kezdetben 3 aktor van jelen: Látogató, Felhasználó, Admin. Ezek tulajdonképpen három jogosultsági szintet jelentenek. Az alkalmazásban azonban lehetőség lesz újabb szerepkörök hozzáadására, meglévők módosítására és törlésére.

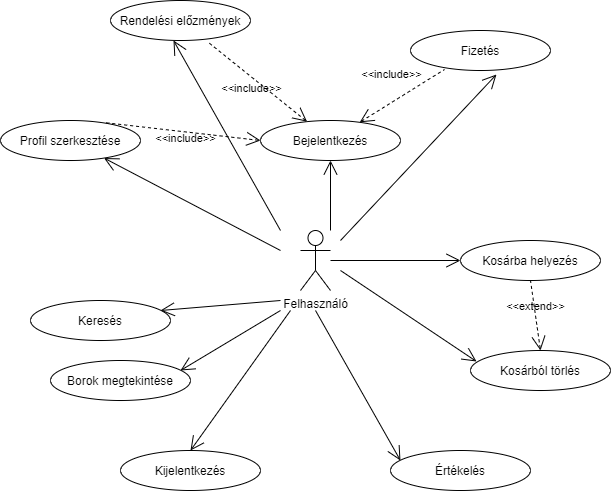
**A rendszer aktorai**

Látogató: A weboldalt meglátogató, be nem jelentkezett felhasználó. Lehetősége van a borokat megtekinteni, köztük keresni, és kosárba rakni, onnan törölni, és az oldalra regisztrálni. A vásárlást azonban az alkalmazás nem engedi, ha valaki Látogató szerepkörben szeretne fizetést indítani, a rendszer figyelmezteti, hogy bejelentkezés szükséges a vásárlás befejezéséhez.



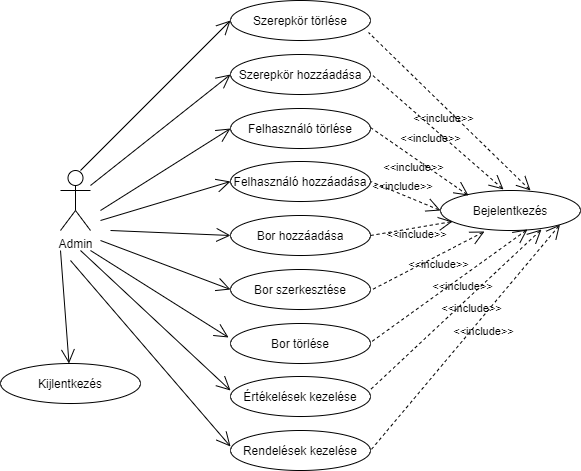
Use case diagram – Látogató aktor

Felhasználó: Ez az aktor tulajdonképpen egy bejelentkezett Látogató. Minden funkciót elér, amit a Látogató, de azon felül engedélyezett neki a kijelentkezés, fizetés, rendelési előzmények megtekintése, profil szerkesztése, értékelés leadása.



Use case diagram – Felhasználó aktor

Admin: Az Admin szerepkörrel lehetőség van felhasználó hozzáadására, törlésére, szerepkör hozzáadására, törlésére, szerkesztésére, borok hozzáadására, törlésére, szerkesztésére, értékelések és rendelések kezelésére.



Use case diagram – Admin aktor

**Use case-ek**

Keresés: Ez a funkció lehetőséget biztosít minden látogatónak a borok közötti keresésre. Nincs jogosultsághoz kötve, regisztrációtól függetlenül eléri minden felhasználó

Borok megtekintése: Ezen funkció szolgál a borok listázására, megjelenítésére. Rajta keresztül érhető el a borok adatlapja, és a kapcsolódó értékelések. Nem jogosultsághoz kötött funkció.

Kosárba helyezés: Az oldal minden látogatójának lehetősége van borokat a kosába helyezni. Ezt megtehetik bejelentkezés nélkül, azonban vásárlási folyamat további lépéseit már nem engedélyezi a rendszer bejelentkezés hiányában.

Kosárból törlés: Bor törlése a kosárból.

Regisztráció: Ez a funkció tulajdonképpen a belépési pont a Látogató és a Felhasználó szerepkör között. A regisztráció a be nem jelentkezett felhasználók számára elérhető, hatására létrejön a user profilja, és lehetőség nyílik a bejelentkezésre.

Kijelentkezés: A kijelentkezés funkció megszűnteti a felhasználó bejelentkezett állapotát. Hatására Felhasználó szerepkörből Látogató szerepkörbe váltunk. Értelemszerűen csak bejelentkezett állapotnál elérhető funkció.

Értékelés: Ez a funkció biztosít lehetőséget a felhasználónak a borok értékelésére. A bejelentkezett állapoton kívül szükséges feltétel még, hogy a felhasználó az adott bor vásárlója legyen.

Profil szerkesztése: A felhasználó saját profiljának kezelésére szolgáló funkció. Eléréséhez bejelentkezés szükséges.

Bejelentkezés: Regisztrált felhasználók számára elérhető funkció. Hatására elérhetővé válnak a Felhasználó szerepkörhöz tartozó funkciók is.

Fizetés: Bejelentkezéshez kötött funkció, mely a kosárba rakott borok kifizetését teszi lehetővé. Fizetés után a rendelés státusza változik.

Rendelési előzmények: Bejelentkezéshez kötött funkció, amelyik lehetőséget biztosít a felhasználónak az aktuális és korábbi rendeléseinek megtekintésére.

Szerepkör törlése: Admin jogosultsághoz kötött funkció, mely a szerepkörök kezelése között a törlést valósítja meg.

Szerepkör hozzáadása: Admin jogosultsággal rendelkező felhasználó számára biztosít lehetőséget új szerepkör definiálására.

Felhasználó törlése: A felhasználókezelés egyik funkciójaként egy felhasználó törlését teszi lehetővé. Admin jogosultsághoz kötött.

Felhasználó hozzáadása: Admin jogosultságú felhasználónak ad lehetőséget felhasználó hozzáadására, kiválasztott jogosultsági szinttel.

Bor hozzáadása: Új bor hozzáadását teszi lehetővé ez a funkció, Admin szerepkörű felhasználónak. Itt jön létre a bor adatlapja, és ezen funkción keresztül kerülnek beállításra a bor alap adatai.

Bor szerkesztése: Feltöltött bor adatainak módosítására szolgáló funkció. Admin jogosultsághoz kötött.

Bor törlése: Adatbázisban már szereplő bor törlését teszi lehetővé, Admin szerepkörű felhasználónak.

Értékelések kezelése: Admin szerepkörű felhasználó számára nyújt lehetőséget értékelések moderálására.

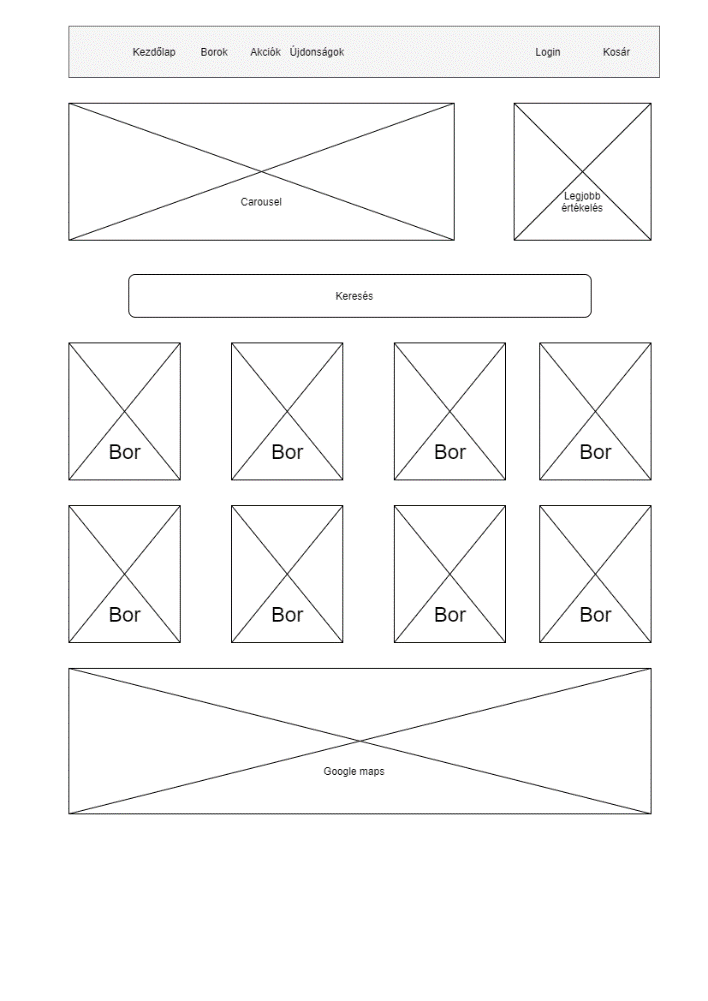
Rendelések kezelése: Admin szerepkörű felhasználó számára nyújt lehetőséget a rendelések kezelésére, mint pl státusz állítás.

## Kezdeti képernyőtervek

A kódolás előtt célszerűnek tartottam az alkalmazás drótvázát megtervezni, hogy lássam, az egyes komponensek nagyvonalakban hogyan fognak testet ölteni az alkalmazásban. Kezdeti lépésben meghatároztam azokat az oldalakat, amiket meg fog jeleníteni a programom: *Főoldal, Borok, Akciók, Újdonságok, Login, Regisztráció, Rendelés, Admin, Borok saját oldala.* Ezután az <https://app.diagrams.net/> webes alkalmazás segítségével elkészítettem az oldalak mockup-ját. A főoldal drótvázát a [**ábra**] mutatja. A további elkészített terveket a dolgozat mellékletében lehet megtekinteni.

Az ilyen struktúraterveket rendkívül hasznosnak tartom, mert a funkciók implementálásánál nagyon jó iránymutatást adnak, valamint segítenek meghatározni azokat a komponenseket, amiket az alkalmazáson belül több helyen is fel tudok használni.

Esetemben nagyon szépen látszott, hogy a borok listázását érdemes komponensben megvalósítani, mert a *Főoldal, Borok, Akciók, Újdonságok* oldalak mindegyikén használható, csak különböző paraméterezéssel.



Főoldal drótváza

1. Az adatbázis ismertetése

A fejlesztés során az adatbázissal kapcsolatos műveleteket hasznosnak találtam ORM technikával kezelni. Az ORM (Object-Relational Mapping) lényege, hogy az adatbázisban tárolt adatokat objektumorientált megközelítéssel kezelhessük. Az általam használt Node.js környezetben a TypeORM és a Sequelize ORM közkedvelt mapper megoldások. Mindkét program segítségével TypeScript osztályokat tudunk különböző adatbázisokkal kezelni anélkül, hogy konkrét adatbáziskezelő specifikus parancsokat használnánk.

A választásom a TypeORM-re esett, mert megítélésem szerint kiterjedtebb dokumentációval rendelkezik, mint a Sequelize ORM.

A TypeORM az adatbázis tábláit és a köztük lévő kapcsolatokat a definiált entitások alapján generálja. Ezért első feladatom az volt, hogy meghatározzam a szükséges entitásokat, melyeket a projekt backend/src mappáján belül egy entity mappában tároltam az átláthatóság kedvéért. Az alábbiakban bemutatom az egyes entitásokat, és azok kapcsolatait, valamint bemutatom a TypeORM által generált táblákat.

## Wine.ts

A borokhoz kapcsolódó információk tárolására alkalmas entitás. Az itt tárolt adatokat jeleníti meg az alkalmazás a borok adatlapján. Szerkesztésükre az adminisztrátornak van lehetősége, és aktuális étékük kerül egy vásárlás alkalmával a megrendeléssel kapcsolatos táblákba. Az entitásból generált tábla a következő mezőket tartalmazza:

|  |  |  |
| --- | --- | --- |
| **Oszlop** | **Típus** | **Leírás** |
| id | int(11) AI PK | Elsődleges kulcs |
| name | varchar(255) | Bor neve |
| imgs | varchar(255) | Feltöltött kép fájlneve |
| winery | varchar(255) | Borász/pincészet |
| description | varchar(255) | Rövid leírás a borról |
| type | varchar(255) | A bor típusa |
| year | varchar(255) | Évjárat |
| alcohol | varchar(255) | Alkoholtartalom |
| color | varchar(255) | Szín |
| sku | varchar(255) | Cikkszám |
| price | int(11) | Aktuális ár |
| discount | int(11) | Kedvezmény |
| quantity | int(11) | Elérhető mennyiség |
| shop | tinyint(4) | Vásárolható vagy sem |
| landing | tinyint(4) | Főoldalon megjelenik |
| createTime | datetime | Létrehozás dátuma |
| updateTime | datetime | Frissítés dátuma |

## User.ts

Ez az entitás a felhasználó adatait hivatott tárolni. Akkor jön létre új bejegyzés, amikor egy látogató regisztrál. Regisztrációkor a felhasználónéven és jelszón kívül nem kell mást megadni, így a generált id-n és az előbb említett két mezőn kívül kezdetben nem lesz más adat egy bejegyzésben. A felhasználói profilhoz tartozó többi adatot később sem kötelező megadni, azonban rendeléskor az alkalmazás ezeket az adatokat tölti be az űrlapba, biztosítva ezzel kisebb kényelmi funkciót a látogatónak. Ebben a fájlban elhelyeztem két metódust, amely a jelszó tárolásával kapcsolatos. Későbbi fejezetben az authentikációs modult részletesebben bemutatom, de itt is lényeges megemlíteni ezt a két metódust. Mivel a jelszavakat hash-elve tárolom, szükséges egy metódus, amely ezt a hash-elést elvégzi, valamint szükséges egy másik, amelyik a bejelentkezésnél összehasonlítja a bejelentkező felületről érkező jelszót az adatbázisban lévő titkosított értékkel.

Ennél a táblánál fontosnak tartom megemlíteni, hogy a Roles és User entitások N:M kapcsolatban állnak egymással, ami azt eredményezi, hogy kapcsolótábla létrehozása szükséges, amibe a két érintett tábla elsődleges kulcsai kerülnek. A TypeORM ennek a kezelésére is lehetőséget biztosít. Az alábbi kódrészlet mutatja, hogy szükséges az N:M relációkat definiálni:

1. @ManyToMany(() => Roles, {
2. eager:**true**,
3. cascade: **true**
4. })
5. @JoinTable()
6. roles: Roles[];

A TypeORM által létrehozott MySQL tábla, és oszlopai az alábbiak szerint alakul:

|  |  |  |
| --- | --- | --- |
| **Oszlop** | **Típus** | **Leírás** |
| id | int(11) AI PK | Elsődleges kulcs |
| firstName | varchar(50) | Keresztnév |
| middleName | varchar(50) | Középső név |
| lastName | varchar(50) | Vezetéknév |
| username | varchar(255) | Felhasználónév |
| password | varchar(255) | Jelszó hash értéke |
| email | varchar(255) | E-mail cím |
| tel | varchar(50) | Telefonszám |
| postcode | varchar(10) | Irányítószám |
| city | varchar(50) | Város |
| address | varchar(100) | Cím |
| cl\_status | tinyint(4) | Státusz |

## Review.ts

A review entitásból létrejött tábla tartalmazza a felhasználói értékeléseket, amelyeket az egyes borokhoz tudnak a felhasználók adni. Az egyes oszlopok ismertetését, azok számossága miatt a dolgozat **1. számú mellékletében** végzem. Ennél az entitásnál fontos megemlíteni, hogy a Wine és a User entitásokkal 1:N kapcsolatban áll, mert egy borhoz több értékelés is tartozhat, azonban egy értékelés csak egy borhoz, ugyanígy egy felhasználónak több értékelése lehet, de egy értékelés egy felhasználóhoz tartozik. Ilyen esetben tudjuk, hogy adatbázis oldalról a review táblába be kell kerülnie egy-egy idegen kulcsnak, amik a wine és user táblákra mutatnak. TypeORM-mel az ilyen kapcsolatok definiálása két részből áll, amit az alábbiakban a Review és User entitások vonatkozásában mutatok be. Először a Review entitásnál kell jelezni, hogy N:1 kapcsolatról van szó, amit az alábbi kódrészlettel tudunk megtenni:

1. @ManyToOne(type => User,
2. {eager: **true**, cascade: **true**})
3. user: User;

A definíció második része, a User entitásban való hivatkozás beállítása:

1. @OneToMany(type => Review, review => review.user)
2. review: Review[];

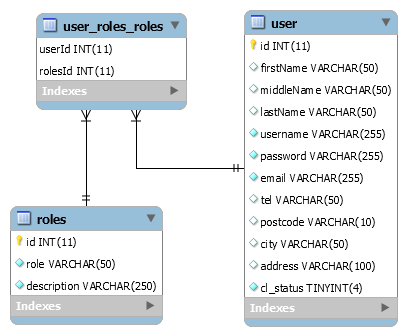
A fejezet későbbi részeiben nem térek ki az ilyen típusú kapcsolatok részletezésére, a megvalósítás ugyanezzel a módszerrel történik.

## Roles.ts

A Roles.ts entitás tartalmazza a különböző szerepkörök megnevezését, és rövid szöveges leírását, ahogy az alábbi **[táblázat]** táblázatban részleteztem, **[ábra]** pedig a tábla kapcsolatait mutatja. Ennek a táblának az alkalmazás authentikációs és felhasználókezelő moduljánál lesz lényeges szerepe, amit a dolgozat későbbi fejezetében fogok kifejteni.

|  |  |  |
| --- | --- | --- |
| **Oszlop** | **Típus** | **Leírás** |
| id | int(11) AI PK | Elsődleges kulcs |
| role | varchar(50) | Szerepkör megnevezése |
| description | varchar(250) | Szerepkör rövid leírása |

Roles tábla felépítése



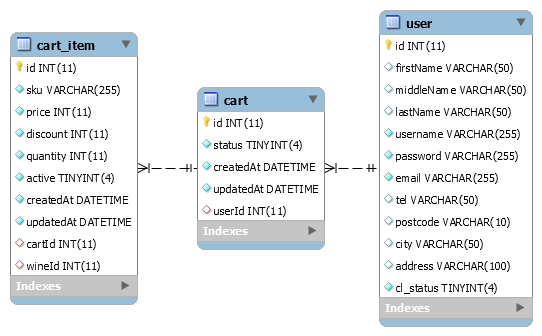
Roles tábla kapcsolatai

## Cart.ts

A Cart entitásból előállított tábla tartalmazza a felhasználókhoz tartozó kosarakat. Egy időben egy felhasználóhoz egy aktív kosár tartozhat, ezért a kosár állapotát a *status* oszlop értéke tárolja. A kosarakba kerülő elemek a C*artItem*-ek, mellyel 1:N kapcsolat áll fenn. A *User* entitással pedig N:1 reláció állapítható meg. A létrejött táblák közötti kapcsolatokat a [**ábra**] szemlélteti, a tábla oszlopainak részleteit pedig a [**táblázat**]

|  |  |  |
| --- | --- | --- |
| **Oszlop** | **Típus** | **Leírás** |
| **id** | int(11) AI PK | Elsődleges kulcs |
| status | tinyint(4) | Kosár státusza (aktív/nem aktív) |
| createdAt | datetime | Létrehozás dátuma |
| updatedAt | datetime | Frissítés dátuma |
| **userId** | int(11) | Idegen kulcs a User táblához |

Cart tábla oszlopai



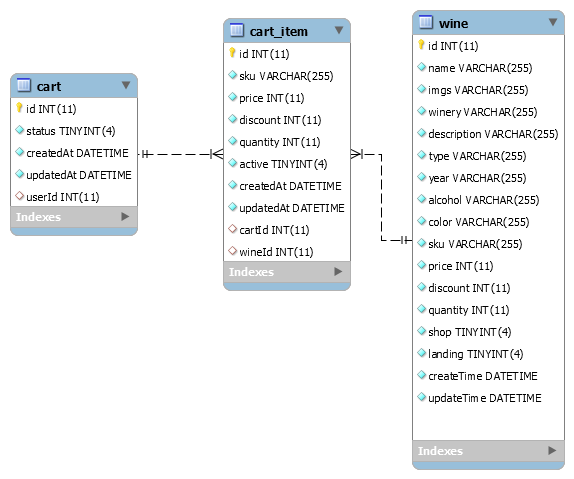
Cart tábla és kapcsolatai

## CartItem.ts

A *Cart*-hoz szorosan kapcsolódó entitás a *CartItem*, amely egy vásárlói kosárba bekerülő bor aktuális paramétereit reprezentálja. Szerencsésnek tartottam erre egy külön entitást létrehozni, ahelyett, hogy a *Wine* -t kapcsoltam volna közvetlenül a *Cart*-hoz, mert például abban az esetben, ha egy bor ára változik, az kihatással lett volna a korábbi rendelések áraira így, ha egy korábbi rendelés adataira lettünk volna kíváncsiak, akkor ott nem az akkori árak szerepelnének. Az ezek alapján generált tábla részleteit a [**táblázat**], kapcsolatait a [**ábra**] mutatja.

|  |  |  |
| --- | --- | --- |
| **Oszlop** | **Típus** | **Leírás** |
| **id** | int(11) AI PK | Elsődleges kulcs |
| sku | varchar(255) | Cikkszám |
| price | int(11) | Kosárba kerüléskori ár |
| discount | int(11) | Kosárba kerüléskori kedvezmény |
| quantity | int(11) | Mennyiség |
| active | tinyint(4) | Státusz |
| createdAt | datetime | Létrehozás dátuma |
| updatedAt | datetime | Frissítés dátuma |
| **cartId** | int(11) | Idegen kulcs a Cart táblához |
| **wineId** | int(11) | Idegen kulcs a Wine táblához |

CartItem tábla oszlopai



CartItem tábla kapcsolatai

## Order.ts

Az *Order* entitás reprezentálja a leadott rendeléseket. A létrehozott táblában foglalnak helyet a számlázási, és szállítási információk, egy állapot oszlop és idegen kulcsok a *Cart* és *User* táblákhoz. Az oszlopok részletezését a **2. számú melléklet mutatja, a táblák közötti kapcsolatokat pedig a 3. számú melléklet.**

## Carousel.ts

Ez az entitás tartalmazza a főoldalon megjelenő carousel-ek adatait. A többi táblával nincsen kapcsolata, ezt a táblát nem használom több táblát érintő lekérdezésekben, csupán adatbevitel és módosítás, valamint a sorok kiolvasása történik. A [**táblázat**] részletezi a tábla oszlopait.

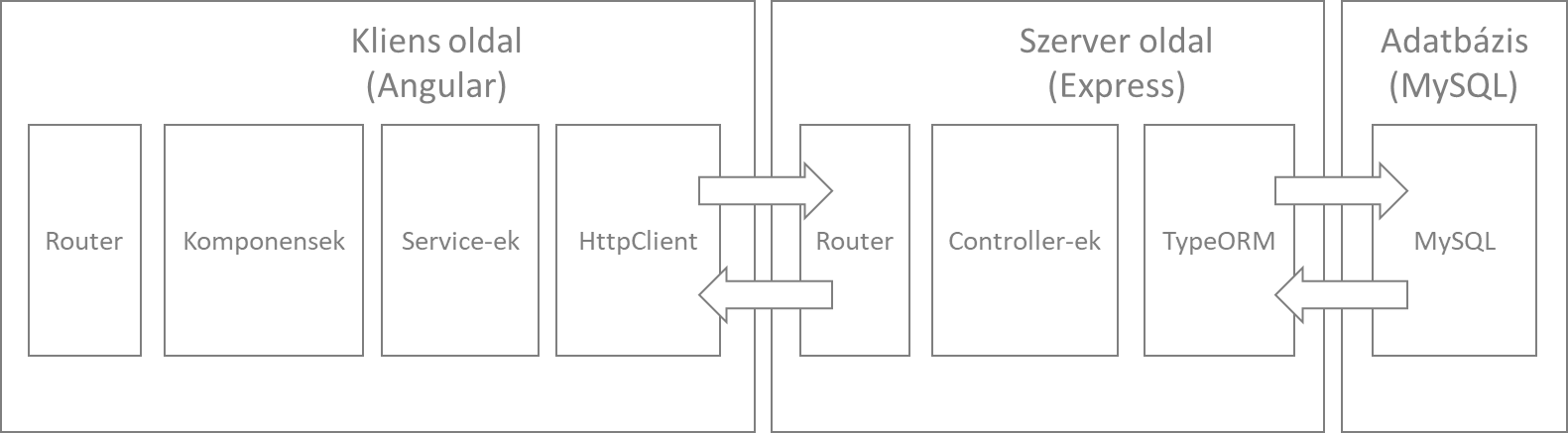
|  |  |  |
| --- | --- | --- |
| **Oszlop** | **Típus** | **Leírás** |
| **id** | int(11) AI PK | Elsődleges kulcs |
| text | varchar(255) | Carousel lapon megjelenő szöveg |
| img | varchar(255) | Megjelenő kép fájneve |
| active | tinyint(4) | Aktív/inaktív flag |
| link | varchar(255) | Szövegre kerülő url |
| position | int(11) | Carousel lap pozíciója |

Carousel tábla

[Új phpMyAdmin ablak nyitása](http://localhost/phpmyadmin/db_datadict.php?db=shop&table=&server=1&target=)

1. Alkalmazás fejlesztése

A fejlesztői környezet kialakítása, valamint az adatbázis megtervezése és létrehozása után a szükséges funkciók implementálásával folytattam a munkát. Az alkalmazott technológiákból kifolyólag jól elkülöníthető részekre lehet bontani a fejlesztést. A főbb funkciók fejlesztése egyaránt érinti a szerver és kliens oldalt. **[https://angular.io/guide/http]** Az Angular keretrendszer a HTTP protokollon keresztül kommunikál a backend-del. HTTP kérések és válaszok formájában. Ehhez a HttpClient service osztályt biztosítja a fejlesztők számára az alkalmazás kliens oldalán. Szerver oldalon az Express eszköz REST API-kon keresztül végzi a kommunikációt a MySQL adatbázissal. A működést az alábbi, egyszerűsített ábrán mutatom be:



Alkalmazáson belüli kommunikáció

A fenti folyamatból az előző fejezetben részleteztem a TypeORM és adatbázis kapcsolatot, és meghatároztam a szükséges entitásokat. A folyamatábra többi részegységét ebben a fejezetben fogom az program főbb funkcióin keresztül.

## Mappastruktúra kialakítása

Az Angular rendszer építőegységei a komponensek. Ez a szemléletmód és architektúra kiváló eszköz újrafelhasználható kódok létrehozására. Az egyes funkciók implementálása előtt az előbb leírtak miatt szükséges volt átgondolni, hogy frontend oldalon milyen komponensekbe érdemes szervezni a funkciókat. Ezeket a komponenseket egy külön components nevű mappába rendeztem, hogy ne keveredjenek az alkalmazás többi részével az src/app folderben. A komponensek számossága miatt nehezen lehetett volna átlátni az Angular app mappájának tartalmát, ezért törekedtem arra, hogy itt saját fájlok ne szerepeljenek. Ezek után született az alábbi mappastruktúra:

A képen szöveg látható

Automatikusan generált leírás

Angular fő mappastruktúra

A backend struktúrájának kialakításánál is igyekeztem figyelembe venni, hogy minél nagyobb rendezettséget szolgáltassanak a mappák. A szerver oldal fő mappája a backend/src, amit igyekeztem úgy szervezni, hogy alatta csak az index.ts, valamint a routers.ts fájlok maradjanak. Előző fejezetben említettem, hogy az index.ts fájl indítja az Express szervert, és tölti be a szükséges programokat. A routes.ts fájl tartalmazza a szerver oldali routingot. A szerver gyökér mappája a backend folder konfigurációs fájlokat, valamint az uploads mappát tartalmazza, ahová a borokhoz feltöltött képeket helyezi a feldolgozás során a program. A fentiek alapján a kialakított mappastruktúra a következő:

A képen szöveg látható

Automatikusan generált leírás

Backend mappastruktúra

## *Route-*ok és *Controller*-ek létrehozása

*Routing*-nak azt a módszert nevezzük, ami meghatározza, hogy mely metódusok kerüljenek meghívásra amikor a kliens oldalról egy *request* érkezik egy adott URL-re. Ennek kezelésére az Express.js express.Router osztályát használom, mellyel moduláris útvonalkezelőket tudunk létrehozni.

Az útvonalak létrehozásához nagyon szorosan hozzátartozik a controller-ek definiálása. Ezek lesznek azok a callback funkciók, amiket meghív a router, ha az URL-re HTTP kérés érkezik.

Mivel az alkalmazás egyes funkciói számos adatbázishoz kapcsolódó műveletet végeznek, mindenképpen célszerűnek tartottam ezeknek a hívásoknak a rendszerezését. Legkézenfekvőbb volt entitások szerint szervezni, annál is inkább, mert TypeORM-ben a az entitások elérése úgynevezett repository-n keresztül történik a getRepository()metóduson keresztül. Például a User entitáshoz kapcsolódó repository objektum az alábbi kóddal érhető el:

repository = getRepository(User);

Amint létrejött az objektum, elérhetővé válnak az entitás objektum adatbázis műveletei. Az alap CRUD műveleteket megvalósító metódusokat egy Controller osztályban definiáltam, az entitásokat reprezentáló osztályokat pedig ebből származtattam le. A legtöbb entity az alapműveletek metódusain felül használ továbbiakat, amiket szintén ebben a fázisban implementáltam, hogy az útvonalakat aztán létre tudjam hozni. Miután elkészültem a Controller-ekkel, a backend routes.ts fájljában szükséges volt ezek importálása, majd a példányosítás után a metódusokat a megfelelő útvonalakhoz rendeltem. Szemléltetésként az alábbi kódrészletben bemutatom az összes felhasználó listázásához használt backend oldali műveletet, ami a frontend irányából a /users URL-re érkező GET kéréshez illeszti userController osztály getAll() metódusát:

1. **const** router = Router();
2. **const** userController = **new** UserController();
3. router.get('/users', userController.getAll);

Ennek a folyamatnak a végeztével a szerver oldal készen állt arra, hogy a frontenddel kommunikáljon, és igény szerint adatbázisműveleteket hajtson végre.

Következő lépésben az alkalmazást arra kellett felkészíteni, hogy ugyanezen kommunikáció kliens oldalát is képes legyen kezelni.

## *Service*-ek és *model*-ek létrehozása

A service-eket leginkább arra használom az alkalmazásban, hogy a bennük deklarált metódusok HTTP kéréseket küldjenek a backend bizonyos útvonalára, és a válaszokat fogadják, ezeket pedig átadják a komponenseknek. Ahogy a mappastruktúránál bemutattam, a service-ket is egy külön mappában tárolom, ezért a felhasználókhoz kapcsolódó service létrehozása a service mappába történő belépés után következő paranccsal történik:

ng generate service user

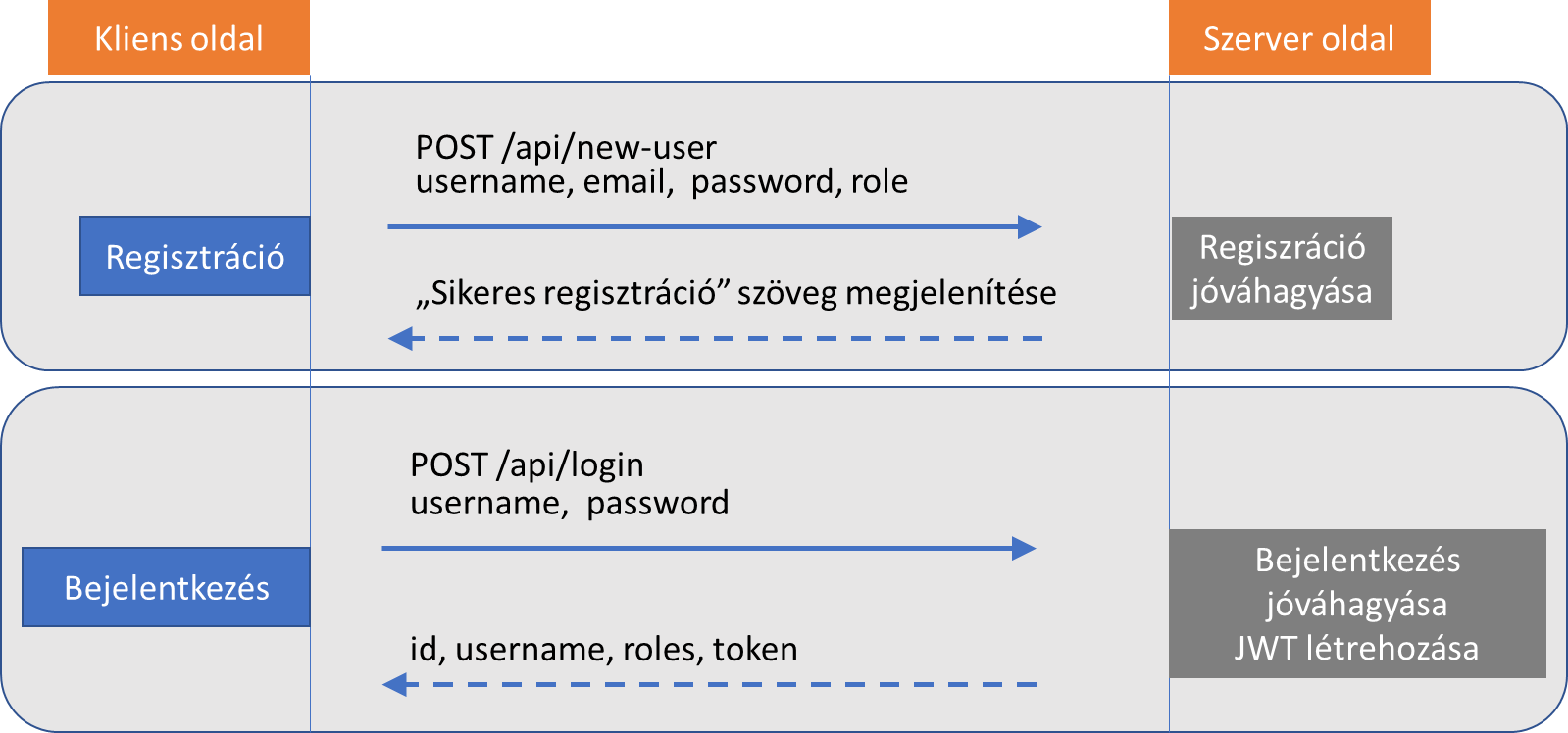
## Autentikációs modul

A program működésének talán leglényegesebb funkciója az autentikáció, ezért az ehhez kapcsolódó feladatokkal kezdtem a kódolást. Mindenekelőtt meg kellett határozni a felhasználó-azonosítási folyamatot, melyet az előző fejezetekben részleteztem. Leegyszerűsítve a folyamatot: bizonyos oldalakat csak bejelentkezett felhasználók nyithatnak meg. Ha a felhasználó már regisztrált, akkor egy login formon tud bejelentkezni, ha nem, akkor először egy regisztrációs form sikeres kitöltését kell kezdeményeznie.

Mindkét művelethez szükséges egy-egy HTML űrlap a felhasználói adatok beküldéséhez. A bevitt adatokat frontenden és backenden is vizsgálom, és csak sikeres validáció után történik meg az adatbázisba mentés. A regisztrációnál megkötés, hogy olyan e-mailcímmel már nem lehet regisztrálni, amivel már történt sikeres regisztráció. A jelszó adatbázisban tárolása előtt fontos azt valamilyen eljárással titkosítani. **[https://www.npmjs.com/package/bcryptjs]** Erre a Node.js bcrypt modulját használtam, ami úgynevezett „sózó” módszert alkalmaz a szivárványtáblás támadások ellen. A sikeres regisztráció után egy e-mailt küld az alkalmazás a megadott e-mailcímre.

**[https://nodemailer.com/about/]** Az e-mail küldést a nodemailer modulon keresztül végzem az alkalmazásban. Ezzel a modullal rendkívül könnyen lehet Node.js programokból e-mail-eket küldeni, és további nagy előnye, hogy minimális konfiguráció szükséges a használatához.

A bejelentkezés során vizsgálom, hogy a megadott felhasználónév jelszó párossal létezik e bejegyzés az adatbázisban. Ha igen, akkor a JWT modul használatával egy tokent készít a program, és elhelyezi benne a felhasználó azonosítót, felhasználónevet, és a felhasználó szerepköreit, és beállítja a token lejárati idejét. A JWT **[https://jwt.io/introduction**] (JSON Web Tokens) egy módszer, mellyel biztonságosan, JSON objektumként lehet továbbítani információt a felek között. A backend az aláírt tokent küldi response-ként a frontendnek, ami ezt egy service-en keresztül a session-storage-ba menti. A folyamat egyszerűsített ábrája alább látható:



Regisztrációs és autentikációs folyamat

A hitelesítési folyamathoz szükséges a frontend és backend között közlekedő token ellenőrzése. Ezt a jsonwebtoken eszköz verify metódusával lehet elvégezni. amely a request header-ből kapott tokent, és egy secret key-t vár paraméterül. A secret key-t egy külön konfigurációs állományban tárolom.

Az alkalmazáson belül bizonyos oldalakat csak adott szerepkörrel rendelkező felhasználók érhetnek el. Ennek megvalósítására az Angular *CanActivate interface*-ét használtam. [**https://angular.io/api/router/CanActivate**] Ezzel az interface-szel útvonalak védelmét lehet végrehajtani, ami a gyakorlatban annyit jelent, hogy ha a definiált feltételek mindegyike true-val tér vissza, akkor folytatódik a navigáció a route szerint, ha bármelyik feltétel false, akkor megszakad a navigáció. Az ellenőrzéshez egy külön service-t használok, ami a *sessionstorage*-ban tárolt token dekódolása után megkapja a felhasználóhoz tartozó szerepköröket, és ezt összeveti az *app-routing.module.ts*-be beállított *data property*-ben lévő szerepkörrel. A lenti kódrészlet mutatja, hogy az *app-routing.module.ts*-ben hogy történik a *data property* beállítása:

1. { path: 'admin',
2. component: BoardAdminComponent,
3. canActivate: [GuardService],
4. data:{roles: ['ADMIN']} },

## Admin modul

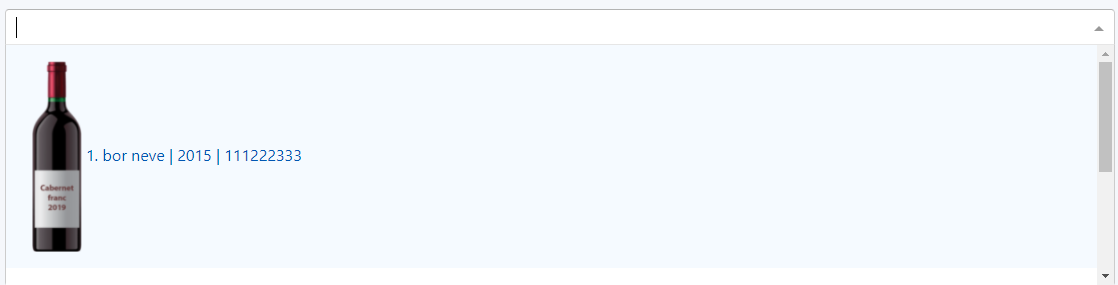
Az alkalmazás admin modulját értelemszerűen csak admin jogosultsággal rendelkező felhasználók érhetik el. Az ellenőrzés az előző fejezetben említett CanActivate interface-en keresztül történik. Az admin dashboard-on a felhasználó együtt lát minden elérhető funkciót, melyek:

1. Borok kezelése
2. Carousel kezelése
3. Felhasználók kezelése
4. Rendelések kezelése

Az alábbiakban ezeket a funkciókat fogom részletezni.

### Borok kezelése

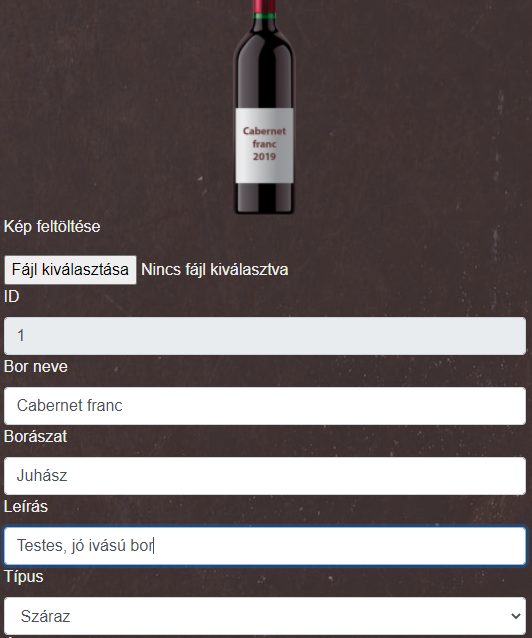
Itt van lehetősége az adminisztrátornak a borok szerkesztésére, törlésére, hozzáadására. A kezelési metodikát igyekeztem leegyszerűsíteni, így a folyamat belépési pontjaként egy keresőmezőt hoztam létre, amit autocomplete funkcióval kibővítve lehet használni a teljes készlet szűrésére.



Borok közötti keresés

Ennek megvalósítására az [**https://github.com/ng-select/ng-select**] ng-select nevű library-t használtam, melyet a borok listájának adataival töltöttem fel. A borokkal kapcsolatos műveleteket egy külön oldalon lehet elvégezni. Itt van lehetőség a bor minden paraméterének beállítására, amik nagy része értelemszerű. Két tulajdonságot checkbox-ok billentésével lehet módosítani. Egyik az „Elérhető” paraméter, másik a „Főoldalra”. Előbbit akkor használjuk, ha egy bor átmenetileg kikerül a kínálatból így, ha ez a checkbox jelöletlen marad, úgy a bor nem fog szerepelni a listázásokban. Ez valósítja meg a törlés funkciót, ami esetünkben nem fizikai törlést jelent, hanem egy státuszváltozást. A „Főoldalra” opció adja meg, hogy egy adott bor a főoldalon kapjon e helyet vagy sem. Mivel nem tartottam célszerűnek a főoldalon a teljes borkínálatot szerepeltetni, ezért implementáltam ezt az egyszerű funkciót, ami az adminisztrátor kezébe adja a döntés lehetőségét, hogy csak bizonyos szempontok alapján kiválasztott borok jelenjenek meg a főoldalon.

Itt van lehetőség a borok képeinek cserélésére is, amit az [**https://github.com/valor-software/ng2-file-upload**] ng2-file-upload modullal valósítottam meg. A modul a kiválasztott fájlt továbbítja a backend-nek, ahol a tárolás előtt timestamp-pel látja el a fájlnevet a program, így biztosítva az egyediséget. A fájlokat a backend uploads mappájában tárolom, az adatbázisba csak a generált fájlnév kerül mentésre. Minden adatfrissítés időpontja mentésre kerül az adatbázisban is.



Borok adatlapjának szerkesztése

### Carousel kezelése

A főoldal egyik kiemelet blokkjában helyeztem el egy carousel-t, amibe korlátozott számú elem kerülhet, kifejezetten figyelemfelhívó célzattal. Jellemzően akciók, események és újdonságok promotálására alkalmas elem, ami folyamatos mozgásával a látogató tekintetét a kívánt tartalomra irányítja. Ehhez az *ng-bootstrap* [**https://ng-bootstrap.github.io/#/home**] nevű csomag carousel komponensét használtam. Az *ng-bootstrap*, a Bootstrap által biztosított widget-eket szolgáltatja, kifejezetten Angular keretrendszerre optimalizálva. Telepítése a már sokszor használt *npm* csomagkezelőn keresztül történik az *ng add @ng-bootstrap/ng-bootstrap* paranccsal. Fontosnak tartottam, hogy ennek a blokknak a kezelése ki legyen vezetve az adminisztrátori kezelőfelületre. Ezt a komponenst alapvetően szerkesztett képek prezentálására céljából implementáltam, ugyanakkor 3 lényeges funkciót biztosítani kívántam az admin szerepkörű felhasználóknak:

1. Új elem hozzáadása, törlése
2. Opcionálisan szöveg és link hozzáadása egy elemhez
3. Elemek sorrendjének változtatása

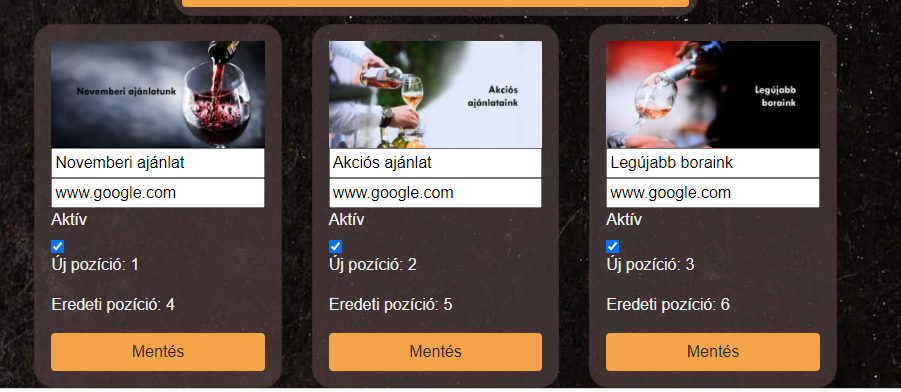
Az első két pont megvalósítása az előzőekben megismert módon, egy űrlapon keresztül valósul meg, ami tartalmaz egy képfeltöltő mezőt, és két szövegbeviteli mezőt:

A képen szöveg, beltéri, képernyőkép, bezárás látható

Automatikusan generált leírás

Carousel elem feltöltése

Az elemek sorrendjének változtatása ez előzőnél kicsit több átgondolást igényelt. A legegyszerűbben implementálható megoldásnak kezdetben az tűnt, hogy egy mezőben manuálisan adjuk meg, hogy az adott elem hányadik helyen jelenjen meg. Nagyobb számú carousel lapnál azonban körülményes lenne a kezelése, és a véletlenül azonos sorszámot kapott lapok kezelését is meg kellett volna oldani. Ezért egy drag and drop-ra épülő megoldás kialakítását kezdtem el vizsgálni. Ennek a működési elve az, hogy a drag and drop-pal berendezett elemek aktuális pozícióját tárolom az adatbázisban, és a lekérdezésnél a pozíció alapján rendezem sorrendbe az aktív elemeket, majd jelenítem meg frontend oldalon a carousel-ben. „Fogd és vidd” módszerre számos eszköz létezik, azonban én az *ng2-dragula* [**https://github.com/valor-software/ng2-dragula**] csomagot választottam. Számomra nagy előnye a többi csomaggal szemben, hogy viszonylag egyszerűen lehet kezelni az elemek aktuális helyét, mert minden változást szinkronizál egy tömbbe, ami a konténer elemen belül történik. Az adminisztrátor dolgát megkönnyítendő, az egyes lapoknál az eredeti pozíciót is jelöltem:



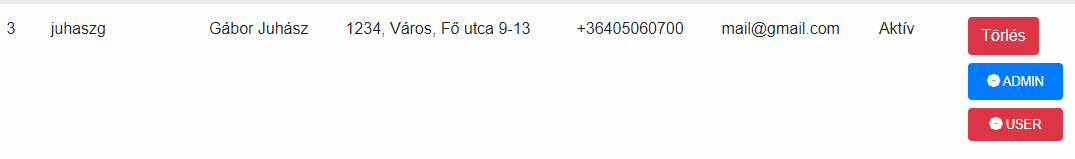
Carousel sorrendező felület

Ahogy a fenti képen is látszik, a sorrendezés mellett az egységek szövegezését, a linkeket, és az aktív státuszát is lehet kezelni a felületen. Az aktív checkbox itt is az elemek státuszát reprezentálják. Mivel itt sem alkalmazok fizikai törlést, ezzel a pipával lehet módosítani egy-egy carousel-lap állapotát. Értelemszerűen, az inaktívak, nem kerülnek listázásra.

### Felhasználók kezelése

Az admin dashboard következő funkciója a felhasználók kezelése. Kialakításánál itt is törekedtem az egyszerűségre, ezért egy táblázatban jelenítettem meg a felhasználók adatait, és minden rendelkezésre álló művelethez egy akciógombot rendeltem. A felhasználók szűrését nem aszinkron adatbázis lekérdezésekkel valósítottam meg, hanem frontend oldalon szűröm a felhasználó objektumok tömbjét. Így a keresőmezőn belül egy időben van lehetőség szűrni a felhasználó azonosítóra, felhasználónévre, e-mailcímre. Ehhez a JavaScript *Array.filter()* metódusát [**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array/filter**] használtam, amely azon elemek tömbjét adja vissza, amelyek megfelelnek a callback funkcióban meghatározott feltételeknek.

A felhasználók közötti keresésen túl itt van lehetőség különböző, felhasználókat érintő műveletek végrehajtására, úgymint: aktiválás/deaktiválás, szerepkör hozzáadása eltávolítása. A deaktiválás művelet hatására egy adott profil tulajdonosa nem fog tudni bejelentkezni az oldalra. Nem tekinthető végleges törlésnek, mert az aktiválás művelettel visszaállítható. Lényeges funkció a szerepkörök kezelése, amit szintén akciógombok segítségével oldottam meg, ahogy az alábbi képen látható:



Felhasználókezelés

Ennek a blokknak a logikája, hogy az összes szerepkört megjelenítem a felhasználók mellett, és annak függvényében lehet hozzájuk rendelni vagy törölni, hogy az adott szerepkörrel már rendelkezik vagy sem. A fenti képen például a teszt felhasználó „*USER*” szerepkörrel rendelkezik, így azt csak törölni lehet, erre utal a piros színezés és a piktogram. Az „*ADMIN*” szerepkör még nincs a felhasználóhoz rendelve, így a kék gomb, és az ikon jelzi, hogy ezt a *role*-t hozzá lehet adni.

Ennek a folyamatnak a technikai háttere, hogy a szerepköröket egy service-en keresztül lekérem az adatbázisból, ugyanúgy, mint a felhasználókat. Frontend oldalon végzem azt az ellenőrzést, hogy egy szerepkör tartozik-e egy *user*-hez. Ehhez első lépésbe a template állományban az *NgFor* direktívát használom, ami az összes *role*-t tároló property-n végigmegy, és egy *checkRoles()* nevű metódus segítségével megállapítja, hogy az aktuális szerepkör benne van e a felhasználó *roles[]* tömbjében. A visszatérő *true* vagy *false* értéktől függően jeleníti meg az akciógombokat. A funkció kódrészletei a HTML és a *ts* fájlból alább láthatók:

1. <div \*ngFor="let role of listRoles">
2. <button \*ngIf="checkRoles(user.id, role.id); else norole"
3. **class**="my-2 btn btn-danger btn-block small"
4. (click)="unsetRoles(user.id, role.id)">
5. <i **class**="bi bi-patch-minus-fill"></i> {{role.role}}
6. </button>
7. <ng-template #norole>
8. <button **class**="my-2 btn btn-primary btn-block small"
9. (click)="setRoles(user.id, role.id)">
10. <i **class**="bi bi-patch-plus-fill"></i> {{role.role}}
11. </button>
12. </ng-template>
13. </div>

Kódrészlet a HTML template-ből

1. checkRoles(userId:any, roleId:any){
2. **this**.currentUser =  **this**.allUsers.filter((user:any) =>
3. user.id === userId && user.roles.some((role:any) =>
4. role.id === roleId));
6. **if** (**this**.currentUser.length > 0){
7. **return** **true**;
8. }
9. **else**{
10. **return** **false**;
11. }
12. }

Kódrészlet a ts fájlból

### Rendelések kezelése

Az admin dashboard utolsó nagy funkciója, a rendelések kezelése. Az előzőekben megismert módon itt is igyekeztem az egyszerűségre törekedni, ezért hasonlóan a felhasználókezeléshez, most is szűrhető táblázatba rendezve jelenítem meg a rendelések adatait. Egyetlen gombot helyeztem el a sorok végén, amivel a rendelés részleteit lehet megtekinteni.

A keresés és szűrés funkcióhoz teljesen ugyanazt a megoldást implementáltam, mint a felhasználók esetében, ezért ezt ebben a fejezetben nem részletezem.

A rendelés részletek panelt egy *modal*-on jelenítem meg, ami tulajdonképpen egy overlay, és nagy előnye, hogy nem navigál el az oldalról, ugyanakkor dinamikusan lehet benne adatokat megjeleníteni. Az előző fejezetekben említett *ng-bootstrap* csomag *modal* komponensét használtam erre a célra, mely nem igényelt különösebb konfigurálást, csupán a komponens HTML állományában egy *ng-template* elemet kell létrehozni, és megfelelően felparaméterezni, a következők szerint:

<ng-template #content let-modal>

## Borok megjelenítése és szűrése

A programom felhasználók számára leginkább előtérben lévő funkciója a borok megjelenítése. Az alkalmazás jellegéből adódóan már a főoldalon találkozik vele a látogató, de a *Borok, Akciók, Újdonságok* oldalakon is használom.

A borok listájának megjelenítését a Bootstrap Card elemének segítségével végeztem, melynek a használata egyszerű, mégis modern megjelenési formát kölcsönöz az oldal ezen részének. Az adatbázisból lekérdezett adatokat az NgFor direktívával jeleníti meg az alkalmazás a kártyákon. Nem tarottam célszerűnek a borról elérhető összes információt a listázó oldalakon megmutatni, ezért csak a lényeges adatok szerepelnek ezekben a szekciókban, úgymint a borhoz feltöltött képet, ár, készletinformáció, bor neve, pincészet, évjárat, valamint egy „Kosárba” és egy „Részletek” gomb. Szemléltetés céljából alább a végleges dizájnról készült képernyőkép is mutatja egy kártya felépítését.

A képen szöveg, elektronika látható

Automatikusan generált leírás

Ehhez a komponenshez szorosan kapcsolódik a szűrés funkció is, ami a felhasználóknak engedi a listát szűrni a beállított feltételek szerint. Ennek megvalósítására a korábban említett *Array.filter()* metódust használtam. A template fájlban az egyes szűrési feltételekhez gombokat rendeltem, amiknek a click eseménye meghívja a search metódust, ami egy paramétert vár. A paraméterként megkapott string érték alapján szűri a fent említett *Array.filter()* metódus a borok tömbjét.

A termékek nem csak listában jelennek meg az alkalmazásban, hanem saját oldalukon is. Itt minden tárolt információ elérhető róluk, ugyanakkor itt is lehetőség van vásárlásra, és értékelés hozzáadására. Az értékeléseket is ezekről az oldalakról lehet elérni. Bár egy oldalon jelennek meg, külön komponensekben implementáltam ezt a két funkciót, egyrészt, mert logikailag jól szétválasztható a kettő, másrészt pedig azért, mert mindkettő viszonylag nagy számú adatot jelenít meg, és az esetleges megjelenítéssel kapcsolatos változtatásokat egyszerűbb kezelni. Kód szinten a két komponens működése nagyon hasonló, egy service az URL-en keresztül kapott id alapján lekérdezi az adatokat, majd az eredményt a template-be illeszti a korábban bemutatott módszerrel. A borhoz tartozó értékelések tömbjét a ts fájlban az alábbi kódrészlettel érjük el:

1. **const** id = **this**.activatedRoute.snapshot.params.id;
2. await **this**.userService
3. .getWineReviews(id)
4. .then((data: Review[]) => {
5. **this**.reviews = data;
6. });

Az értékelések listázásánál inkább megjelenés béli problémákat megoldani. Mivel 15 adatpontot kell ábrázolni értékelésenként, nem tartottam szerencsésnek az összeset egyszerre mutatni, mert akkor egy rendkívül hosszan vertikálisan görgethető oldallá alakult volna az alkalmazás ezen része. Ezért keresnem kellett egy olyan technikát, ami csak a lényeges információkat mutatja a betöltődéskor, és a többit csak akkor, ha arra a felhasználó valóban kíváncsi. Ennek megvalósítására a Bootstrap rendelkezik beépített elemmel, mégpedig az úgynevezett Accordion-nal. Működése egyszerű: a title részben kell elhelyezni azokat az adatokat, amik mindig megjelennek, a content részben pedig azokat, amik a title-re történő kattintás után jelennek meg. A title-be én az értékelés pontszámát és az ízről készített szöveges megjegyzést helyeztem el, a többi információ a content-be került Az Accordion feltöltését a már előzőekben is említett NgFor direktívával végeztem. A jobb szemléltetés miatt az alábbi képen látható egy kinyitott elem a végleges dizájn szerint:

A képen szöveg látható

Automatikusan generált leírás

## Kosár és rendelés

Egy webshop alkalmazás talán legsarkalatosabb funkciói a rendeléshez köthetők. Rendelést csak bejelentkezett felhasználó adhat le, és egy felhasználónak egy időben csak egy aktív kosara lehet. Egy kosár egészen addig aktív státuszban van, amíg a rendelés elküldésével nem véglegesíti vásárlási szándékát a látogató.

A kosár funkció implementálással kezdtem ezt a részfeladatot. A kosár kialakításánál figyelembe vettem, hogy szerencsés lenne minden aloldalról elérni, ezért a menüben helyeztem el a bejelentkezés gomb mellett. Az előző fejezetben említett „Kosárba” gombra kattintással az adott termékből egy darabot helyez a kosárba az alkalmazás, amelyek a virtuális kosárban meg is jelennek, onnan pedig törölni tudja a felhasználó, vagy a rendelést elindítani. A kosárba helyezés folyamata az aktív kosár meglétének ellenőrzésével indul. Ez annak a lekérdezésnek az eredménye, hogy az adott felhasználó azonosítóval van-e olyan bejegyzés a cart táblában, aminek a státusza aktív. Szerver oldalon a TypeORM az alábbi kóddal végzi a lekérdezést:

1. **const** id = req.params.id;
2. **const** entities = await **this**.repository
3. .findOne({where:[
4. {user:id,
5. status:**true**}
6. ]}

Ha nincs aktív kosár, akkor létrehoz egyet, majd hozzáadja a kiválasztott bor, ha van már, akkor a bor a kosárba kerül. Utóbbi művelet technikailag egy bejegyzés létrehozását jelenti a cart\_item táblában, cartId és wineId idegen kulcsokkal.

## Dizájn létrehozása

A tervezés fejezetben már elkészült a szoftver drótváza, ebben a részben a végleges dizájnt fogom bemutatni. A megvalósítás előtt az oldal színvilágát határoztam meg. Igyekeztem egymással harmonizáló színeket találni, amihez számos online alkalmazás is segítséget nyújt, azonban én a Behance [**irodalom**] nevű oldalról gyűjtöttem inspirációkat ezen a téren. Végeredményképpen a következő színeket, valamint ezek áttetsző változatait határoztam meg:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| #f3a446 | #a06235 | #504040 | #efefef |

Az oldal hátterét kezdetben #141313 színkóddal képzeltem el, helyette inkább egy hasonló színű háttérképet szerkesztettem, aminek textúráját felveszik a borokat tartalmazó card elemek hover állapotban. A háttérkép két szélére szerkesztettem egy-egy bor tematikájú dekorációs célú elemet is.

Szükséges volt az alapértelmezett Bootstrap megjelenést felüldefiniálnom, amit egy új CSS fájl létrehozásával és importálásával oldottam meg, valamint az egyes komponensek saját CSS fájljaiban. Az előbbi fájlban helyeztem el az oldal egészét érintő kódokat, mint például a szövegek, linkek, gombok színeit, utóbbiban pedig a konkrét komponens esetleges külalak változásait kezeltem. A dizájn implementálásnál figyelembe vettem, hogy az elemek reszponzív módon működjenek. A felhasználói élményt kicsit javítva, néhány elemre beállítottam a transition effect-et, ami akkor indul, amikor az elem valamelyik CSS property értéke megváltozik, és ezt a változást a beállított időtartam erejéig animálja. Az előzőekben leírtak a css-ben a következőképpen néznek ki:

1. .card{
2. **background-color**: #504040b8;
3. transition:0.2s;
4. border-radius:1.25rem;
5. }
6. .card:hover{
7. **background-color**: #a06235b8;
8. }

Figyelmet fordítottam az oldal reszponzív megjelenésére, amihez a Bootstrap beépített eszközkészletét használtam. Az előre meghatározott töréspontok mindenhol megfeleltek, ezért újakat nem kellett létrehoznom.

1. Tesztelés

A program tesztelése végig kísérte a teljes fejlesztést, mert egy funkcióról már menet közben is szerettem volna tudni, hogy működőképes e. Mivel az alkalmazás szinte összes funkciója adatbázishoz kapcsolódik, ezért a tesztelések jelentős részével azt ellenőriztem, hogy megtörténtek-e a szükséges adatbázisműveletek, és az elvárt eredményt adták e vissza. Szintén sok időt felölelt annak ellenőrzése, hogy minden gomb és link azt a feladatot végzi, amit kell, illetve oda irányít, ahova szükséges.

Nagyon lényeges feladat volt annak a tesztelése, hogy minden megfelelően jelenik-e meg. Sok esetben okozott gondot például, hogy az alapértelmezett Bootstrap dizájn szerint jelentek meg elemek. Ebbe a tesztfeladatba sorolható a reszponzív működés ellenőrzése. Ehhez nem fizikai eszközt (például mobilkészülék) használtam, hanem a Chrome böngésző DevTools funkcióját, ami akár különböző készülékek emulálásával is tudja támogatni a tesztelést. A tesztelés ezen fázisában számos esetben nem az elvártnak megfelelően működtek bizonyos elemek, ezért ezeket a hibákat javítanom kellett.

Természetesen a szoftver gerincét alkotó fő funkciók tesztelése sem maradhatott el. Ezek szinte mindegyikéhez szükséges user, és admin szerepkörű felhasználó. User szerepkörű felhasználó létrehozása közben a regisztrációs folyamatot is tudtam tesztelni. Admin szerepkörű felhasználót pedig a program első indulásakor generálok, a TypeORM migration segítségével. Itt ellenőrizni tudtam, hogy a leveleket megfelelően küldi ki a rendszer. Ehhez szükséges volt, hogy a saját e-mailcímemmel történjen a regisztráció.

Miután a regisztrációs és authentikációs funkciókat rendben találtam, a vásárláshoz kötődő funkcionalitásokat teszteltem. Itt figyelemmel kellet kísérni, hogy a termékek az adatbázisba helyesen kerülnek be, és a kliens oldalon is helyesen jelenik meg a kosár tartalma. Ehhez phpMyAdmin-ban folyamatosan ellenőrizni kellett az inzertált és frissített adatokat. Miután sikeresnek ítéltem kosár működését, a rendelést teszteltem, ami a kosár tesztelésével megegyező módon történt.

Ezen felül az admin funkciókat is ellenőrzésnek vetettem alá. Rögtön azzal kezdtem, hogy az admin dashboard-ra csak admin jogosultsággal lehet-e belépni. Ezután végig mentem az előző fejezetekben bemutatott összes adminisztrátori funkción, és ellenőriztem, hogy az adatbázisba az elvárásoknak megfelelő adatok kerülnek be, illetve kerülnek kiolvasásra és megjelenítésre. A módszer az előzőekben ismertetett módon zajlott.

Szintén nagyobb figyelmet fordítottam az értékelés komponens tesztjére. Itt a megjelenésre és a háttérműködésre is különös figyelmet fordítottam. Borokhoz adtam le értékeléseket több felhasználótól, és ellenőriztem, hogy a megfelelő helyeken jelennek-e meg.

Fontosnak tartottam, hogy a tesztelésnek legyen egy olyan része is, amikor nem én tesztelem a saját alkalmazásomat, hanem egy olyan személy, aki potenciális látogatója lehet az oldalnak a későbbiekben. Ezért a feleségemet kértem meg, hogy nézze meg az oldalt leendő vásárlói szemmel, és vigyen végig egy rendelési folyamatot. Az észrevételeit figyelembe véve változtattam néhány felhasználói élményt érintő dolgon.

1. Felhasználói dokumentáció

A felhasználói dokumentációt alapvetően három részre lehet osztani:

1. Látogatóknak szóló
2. Regisztrált felhasználóknak szóló
3. Adminisztrátoroknak szóló

## Látogatóknak szóló dokumentáció

A látogató tulajdonképpen egy be nem jelentkezett felhasználó. Ahogy a korábbi fejezetekben részleteztem, ilyen szerepkörben nem érhető el az oldal minden funkciója

Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | „https://www.ksh.hu,” 17 04 2021. [Online]. Available: https://www.ksh.hu/heti-monitor/kereskedelem.html. |