

**Debreceni Egyetem**  
Informatikai Kar

# **WEBALKALMAZÁS FEJLESZTÉSE**

Témavezető: Dr. Vágner Anikó

Beosztása: Adjunktus

Készítette: Bartalis Vilmos

Szak megnevezés:

Programtervező informatikus

DEBRECEN, 2022

# Tartalomjegyzék

<b>Tartalmi összefoglaló</b>	<b>4</b>
<b>Bevezetés</b>	<b>5</b>
A webalkalmazások fontossága napjainkban	5
A megoldandó feladat	6
A dolgozatról	8
<b>Felhasznált eszközök és technológiák bemutatása</b>	<b>9</b>
Felhasznált eszközök	9
Git	9
GitHub	9
IntelliJ IDEA	10
Visual Studio Code	10
Google Chrome	10
Felhasznált technológiák	11
Java	11
Maven	11
Spring	12
H2	12
Swagger UI	13
HTML, CSS, TypeScript	13
Node.js	13
Angular	14
Bootstrap 4.6	14
NG Bootstrap	14
Font Awesome	14
<b>Architektúra/Tervezés</b>	<b>15</b>
A konkrét feladat	15
Követelmények	15
Első felhasználói réteg	15
Második felhasználói réteg	16
Harmadik felhasználói réteg	17
A rendszer architektúrája	17
A Backend architektúrája	18

A kontroller réteg	18
A szerviz réteg	19
Az adatelérési réteg	19
A frontend architektúrája	19
A prezentációs réteg	19
A kontroller réteg	20
A szerviz réteg	20
Az adatbázis séma tervezete	20
<b>Önálló munka bemutatása</b>	<b>22</b>
A Backend bemutatása	22
Az adatbázis	22
Az adatelérési réteg	23
Az adat modellek bemutatása	23
A Repository interfészek bemutatása	25
A szerviz réteg	28
A kontroller réteg	30
Autentikáció, autorizáció	34
A Frontend bemutatása	36
A Frontend Felépítése	36
Az autentikáció felületének bemutatása	37
A vendég felületének bemutatása	39
A felhasználó felületének bemutatása	44
Az adminisztrátor felületének bemutatása	48
<b>Összefoglaló</b>	<b>53</b>
<b>Irodalomjegyzék</b>	<b>55</b>

# Tartalmi összefoglaló

A mai világban az internet szinte már mindenhol megtalálható. Mindenki zsebében található egy okos telefon. Minden háztartás és cég rendelkezik internet kapcsolattal. Ezen kívül a webalkalmazások sok más előnnyel is rendelkeznek a hagyományos, asztali alkalmazásokkal szemben, mint például platformfüggetlenség és alacsony erőforrásigény (a végfelhasználó szemszögéből). Ennek köszönhetően a webalkalmazások lassacskán átveszik a teret a hagyományos, asztali alkalmazásoktól.

Egy modern webalkalmazás készítése viszont nem egy banális dolog. Egy applikációnak az internetre való migrációja, vagy akár egy vadonat új applikáció elkészítése számos csapdát is rejthet magában.

E dokumentum összefoglalja, hogyan is lehet egy modern webalkalmazást elkészíteni. Összefoglalja egy modern webalkalmazás architektúráját, felépítését. Bemutatja, hogyan lehet egy a REST alapuló bekenetet elkészíteni - Java nyelven, a Spring Boot keretrendszer felhasználásával - és miként lehet az biztonságossá tenni. Emellett a bekenedhez párosuló frontendet is tárgyalja, amely az Angular keretrendszer felhasználásával jött létre.

Az itt átadott információ nem csupán megismerteti az olvasót egy modern webalkalmazás felépítésével, hanem akár támpontot is nyújthat neki egy saját webalkalmazás elkészítésében.

# 1 Bevezetés

## 1.1 A webalkalmazások fontossága napjainkban

Napjainkban már szinte minden háztartásban megtalálható az internet, szinte mindenkinek van egy okostelefonja és sokan még a munkahelyükön is számítógépek segítségével végzik a munkájukat. Már egy hétköznapi háztartás is rendelkezhet olyan gépekkel, amelyeket távolról el lehet érni és felügyelni (mosógép, robot porszívó, fűtési rendszer vagy klíma, stb. Ennek köszönhetően mutatkozott igény arra, hogy a fejlesztők áthelyezzék az internetre az applikációkat. Ezeket nevezzük webalkalmazásoknak vagy webapplikációknak. Véleményem szerint a lokális, asztali alkalmazások internetre történő migrációja is a globalizáció egyik megnyilvánulása.

A webalkalmazások a hagyományos, asztali applikációkhoz képest rengeteg előnnyel rendelkeznek.

Egy webapplikációt akármikor és akárhonnan el lehet érni. Nincs szükségünk másra, csak egy készülékre, amely képes kommunikálni az internettel és rendelkezik egy böngészővel. Egy webapplikációt manapság akár még egy okos autóból is használni lehet.

A webapplikáció további előnye, hogy nagyon alacsonyak az erőforrás igényei, hiszen maga az üzleti logika egy távoli szerveren található.

A használt operációs rendszer sem képez többé akadályt, hiszen az applikáció maga a böngészőben fut. Véleményem szerint ez a valódi platformfüggetlenség egyik legkiválóbb példája.

A webapplikáció további előnye a rugalmasság, vagyis nem szükséges minden egyes készülékre feltelepíteni ugyanazt az alkalmazást és nem szükséges azt minden egyes alkalommal újrakonfigurálni. Egyszerűen csak beírjuk a megfelelő címet a böngészőbe és az applikáció már fut is. Nincs szükség letöltésre, telepítésre. Nem foglal helyet a készüléken és ha a készülékünk meghibásodik, könnyen folytathatjuk a munkát egy másikon.

A frissítés kérdése is a webapplikáció mellé teszi a voksát. Egy hagyományos asztali alkalmazást általában a felhasználónak kell manuálisan frissíteni. Még ha automatikusan játszódik is le a frissítési folyamat, akkor is elég sok időt vehet igénybe: el kell fogadni a

frissítést, meg kell történnie a frissítésnek , sokszor - a folyamat befejezését követően - még az alkalmazást is újra kell indítani.

Ezeket a szempontokat figyelembe véve dönt úgy sok cég, hogy az általa használt applikációt az internetre költözteti. Nem feltétlenül az a céljuk, hogy az internetről bármikor és akárhonnan el lehessen érni az alkalmazást, ugyanakkor nagy előnyt jelent az, hogy nem kell minden egyes alkalmazottnak a számítógépére feltelepíteni ugyanazokat a szoftvereket és nem kell azokat folyton karbantartani. Az internetre helyezett applikáció (vagyis a webapplikáció) menedzselése sokkal egyszerűbb feladat, hiszen minden egy központi helyen található.

Ejtsünk pár szót a webalkalmazások két fő típusáról: belső használaton keresztül elérhető alkalmazások és interneten keresztül elérhető alkalmazások. Egyes web alkalmazásokat különböző szervezetek, cégek kizárólag belső hálózatban használnak fel. Másokat interneten keresztül lehet elérni. Erre - a hétköznapi emberek számára - legismertebb példák talán a különböző szociális médiák, banki alkalmazások, email kliensek vagy akár az online cserebere oldalak.

Rengeteg féle és fajta web alkalmazás létezik és bár ezek közül sok hasonló technikával készül, tulajdonképpen mindegyik valamilyen úton-módon egyedi jelleggel rendelkezik.

## **1.2 A megoldandó feladat**

A szakdolgozatom tárgyához az ihletet az online cserebere oldalak böngészése közben kaptam. Nagyon hamar rájöttem, mennyire körülményes ezeknek az oldalaknak a használata. Tele vannak fölösleges elemekkel, amelyeket az átlag felhasználó soha nem fog használni. Bár ebben a tekintetben lassan kezdenek javulni az említett oldalak, engem még mindig zavar a sok plusz információ.

A fent említett igencsak népszerű cserebere oldalak további nehézsége a kínált termék helytelen kategória besorolása. Az eladó sokszor nem figyel oda, hogy helyes adatokat adott-e meg, amikor megalkotta a hirdetését. Sőt, sokszor az is megtörténik, hogy szándékosan hibás adatokat ad meg. Ezzel szemben pedig az oldal adminisztrátorai sokszor nem tesznek semmit. Ha mégis fellépnek az ilyesmi ellen, akkor sincs a szóbanforgó hirdetőnek nehéz dolga. Csak

készít magának egy új felhasználót és azzal újabb hibás, vagy éppen egyenesen kamu, hirdetéseket tud feltenni a cserebere oldalra.

Egy másik probléma ezekkel az oldalakkal, hogy nem a specializálódott hirdetőik számára van kitalálva. Inkább használt tárgyak cseréjére, eladására, vásárlására alkalmas. Egy alkalommal amikor kitaláltam, hogy házi kedvencet szeretnék, akkor döbbsentem rá, hogy mennyire nehézkes online kis állatokat találni. A vásárlók nem tudják hol kellene keresniük, az eladók pedig azzal nincsenek tisztába, hogy hol kellene hirdessenek. A legnagyobb esélyt a találatra az előbb említett cserebere oldalak nyújtják, de - amint ezt említettem - azokon is nehézkes a keresés és tele vannak hamis, illetve hibás hirdetésekkel.

Innen származik az ötlet, hogy létrehozzak egy olyan platformot, amelyen kapcsolatba kerülhetnek az állattenyésztők és házi kedvencekre vágyó vásárlók.

Egy olyan applikációt szeretnék készíteni, amelyet bárki meglátogathatna, aki úgy érzi, hogy szüksége lenne egy kis kedvence. Viszont azt nem szeretném, hogy bárki hirdetéseket tudjon az oldalra felhelyezni. Fontos, hogy minden hirdetés igazság tartalommal rendelkezzen. Szerintem erre a legjobb módszer az, hogy ha (én, mint rendszergazda) megsűrűm azokat a személyeket, akik hirdetni szeretnének. Az alapgondolat az, hogy csak azok hirdethetnek, akiknek van felhasználói fiókjuk és csak azoknak lehet felhasználói fiókjuk, akik átestek a szűrésen. Ez lehet, hogy elsőre kissé túlzásnak tűnik, ugyanakkor a hirdető célközönség számára a folyamathoz szükséges pár email váltása nem lenne túl nagy kellemetlenség. Ez a potenciális felhasználói réteg (hirdető célközönség) nem más lehetne, mint az állat tenyésztők és állatmenhelyek. A tenyésztők hirdethetnék a fiatal kisállataikat, amíg az állatmenhelyek az örökbefogadható házi kedvenceiket hirdethetnék az applikáció segítségével.

Egy másik felhasználói réteg a házi kedvencekre vágyó felhasználókból állna. Ők a tenyésztők és az állatmenhelyek által feltett hirdetések között kereshetnének az alkalmazás segítségével.

Természetesen ezen kívül szükség lenne még egy adminisztrációs felületre is. Itt az adminisztrátor tudna új felhasználói fiókokat készíteni és itt történne a meglévő felhasználók jogosultságainak kezelése. Az adminisztrációs felület kezelője képes lenne módosítani az adatokat, zárolni a hirdetéseket vagy akár a felhasználói fiókokat is.

A szakdolgozatom célja egy olyan webalkalmazás készítése, amelynek tartalmaznia kell minden olyan elemet, amely egy modern webalkalmazás számára kötelező lenne. A célom nem az, hogy egy, a későbbiekben rendeltetésszerűen működő webapplikációt kreáljak, hanem az, hogy dokumentáljam és megosszam az olvasóval egy modern webapplikáció elkészítését.

Természetesen ez nem jelenti azt, hogy maga az applikáció nem lesz üzemképes. Valószínűleg kell majd benne egy pár apró változtatást ejteni ahhoz, hogy tényleg megbízhatóan és biztonságosan használni lehessen. Ugyanakkor ezek a problémák már nem föltétlenül köthetők az alkalmazás fejlesztéséhez, inkább az alkalmazás telepítésére és üzemeltetésére vonatkoznak.

### **1.3 A dolgozatról**

Ebben a fejezetben egy rövid betekintést szeretnék nyújtani az olvasónak a dolgozat további részeiről. Ezek három fő fejezetből állnak össze.

Az első fejezetben az applikáció készítése során alkalmazott eszközökre és technológiákra fogok kitérni.

A második fejezetben az applikáció architektúrális tervezésébe szeretném bevezetni az olvasót.

Az harmadik fejezetben az elkészített applikációt szeretném bemutatni az olvasónak. Ezt két részben teszem meg. Az első részében a backend applikációt mutatom be, illetve ennek a felépítését. A második részben pedig a frontend applikációt szeretném bemutatni az olvasónak, azon belül pedig az applikáció grafikus felületét.



## **2 Felhasznált eszközök és technológiák bemutatása**

### **2.1 Felhasznált eszközök**

Ebben a fejezetben be szeretném mutatni az alkalmazás fejlesztése közben felhasznált fontosabb eszközöket.

#### **2.1.1 Git**

Az alkalmazás elkészítése közben szükségem volt egy verziókezelő rendszerre. Azért volt erre szükség, hogy a forráskódban történt változásokat követni tudjam, illetve azokat meg lehessen őrizni, ha netán a későbbiekben szükségem lenne a használatukra.

Azért esett a választásom a Git-re [1], mivel egy nagyon népszerű rendszer, hiszen ingyenes és nyílt forráskódú. Ezen kívül meg kell még említeni két fontos előnyét. Az első az, hogy a Git rendkívül biztonságos az adatintegritás szempontjából. A felhasználónak nem kell attól tartania, hogy az adatai elveszüljenek, vagy esetleg korrumpálódnak. A második előnye, hogy a Git viszonylag gyors más hasonló rendszerekhez képest, mint például az SVN.

#### **2.1.2 GitHub**

A GitHub [2] egy online tárhely szolgáltató platform és egyben egy Git alapú verziókezelő rendszer.

Ennek a használatára azért volt szükségem, hogy a forráskódot ingyenesen tárolhassam online. Erre első sorban azért volt szükségem, hogy az applikáció fejlesztése közben könnyedén tudjak több, különböző számítógépről is dolgozni. Másodsorban azért használtam, hogy a forráskód egy biztonságos közegben legyen eltárolva.

Az applikáció fejlesztése közben megtörtént velem az a szerencsétlen eset, hogy a számítógépem véglegesen elromlott. Viszont annak köszönhetően, hogy a forráskódot a GitHub-on is eltároltam, az applikáció fejlesztését tovább tudtam folytatni egy új számítógép beszerzése után. Nem kellett az egész munkát az elejétől újratekinteni.

A szakdolgozat forráskódját az alábbi GitHub linken lehet megtekinteni:  
<https://github.com/vbartalis/PetShop>.

### **2.1.3 IntelliJ IDEA**

Az IntelliJ IDEA [3] egy integrált fejlesztői környezet. A backend forráskódját ennek az eszköznek a segítségével készítettem el.

A választásom azért esett erre az eszközre, mivel számomra úgy tűnt, hogy ez a legkifinomultabb integrált fejlesztői környezet, amely kifejezetten Java nyelven való programozás van kitalálva.

Az IntelliJ-nek a nagy előnye, hogy rengeteg bővítményt lehet hozzáadni. Ha esetleg valamire szüksége lenne a fejlesztőnek, de az IntelliJ nem lenne képes rá natív módon, akkor nagy annak az esélye, hogy létezik egy bővítmény, amely meg tudja oldani a fennálló problémát.

Léteznek más népszerű integrált fejlesztői környezetek is, amelyek támogatják a Java nyelven való fejlesztést. Ezek közül alternatívát kínálhat talán az Eclipse, vagy éppen a NetBeans.

### **2.1.4 Visual Studio Code**

A Visual Studio Code [4] egy forráskód szerkesztő szoftver. Ez nem egy teljes értékű integrált fejlesztői környezet. A frontend forráskódját ennek az eszköznek a segítségével készítettem el.

Azért esett erre az eszközre a választásom, mivel a Visual Studio Code egy sokoldalú szoftver, szinte bármely nyelven való programozásra felhasználható. Bővítményeket lehet hozzáadni és - mivel ez az egyik legnépszerűbb fejlesztői eszköz - rengeteg kész bővítményt lehet találni hozzá.

Ezekén kívül még meg kell említeni, hogy a Visual Studio Code egy teljesen ingyenes eszköz, sok platformon elérhető és más hasonló eszközökkel ellentétben kevés helyet foglal és kifejezetten gyors.

### **2.1.5 Google Chrome**

A Google Chrome egy webböngésző, amelyet a Google fejlesztette ki. A frontend elkészítésében jelentős szerepet játszott, hiszen ennek segítségével tudtam megtekinteni és kipróbálni a fejlesztés alatt álló projektet.

Választásom azért esett pont erre a webböngészőre, mivel a szakdolgozat témájának választása idején ez volt a világ leghasználtabb böngészője.

Mivel különböző böngészők különböző módokon reagálhatnak ugyanarra a weboldalra, ezért fontos volt számomra, hogy egy olyan böngészőt használjak amelyet minél több potenciális felhasználót tud lefedni. Ez természetesen nem garantálja, hogy minden egyes felhasználó számára ugyanúgy fog kinézni és működni a weboldal. Ezért a későbbiekben célszerű lehet más webböngészőkkel is kipróbálni az adott weboldalt.

Fontos megemlíteni, hogy a Google Chrome, mint más fontos modern webböngészők, nemcsak weboldalak megjelenítésére képes, hanem fel van szerelve fejlesztői eszközökkel is. Ezek az eszközök nagymértékben elősegítették az alkalmazás fejlesztését.

## **2.2 Felhasznált technológiák**

Ebben a fejezetben az alkalmazás fejlesztésében felhasznált fontosabb technológiákat szeretném bemutatni.

### **2.2.1 Java**

Az alkalmazás backendjét Java [5] nyelven írtam meg. Ez egy magas szintű, objektumorientált programozási nyelv, amelyet általános célú felhasználásra fejlesztettek ki.

Azért választottam a Java nyelvet, mivel ennek nagy hagyománya van már a különböző webapplikációk fejlesztésében. Ennek köszönhetően sok olyan, erre épített eszköz található, amelyek kiforrottak, illetve jól dokumentáltak.

### **2.2.2 Maven**

Az Apache Maven [6] egy szoftver projekt menedzsment eszköz. Ezek az eszközök képesek menedzselni a projekt buildelését, reportolását, dokumentációját stb. Ezeket az adatokat a Maven egy úgynevezett *projekt objektum modell*, vagy röviden POM, fájlban tárolja. Az egyike azon adatoknak, amelyet a POM tárol nem más, mint a projekt függőségei. A Maven-nek köszönhetően nem volt szükség arra, hogy ezeket manuálisan letöltssem. Elegendő volt az, ha POM-ban megadtam, hogy melyik függőségre van szükségem és a Maven azt letöltötte számomra.

A projekt elkészítéséhez elengedhetetlen volt egy hasonló technológia alkalmazása. Egy alternatíva a Maven helyettesítésére a Gradle lehetett volna. Azért választottam végül a Maven-t, mivel ez, a Gradle-el ellentétben, egy régebbi és kicsivel kiforrottabb technológia .

### **2.2.3 Spring**

A Spring [7] egy nyílt forrású alkalmazás keretrendszer, amelyet kiváló módon lehet használni Java alapú webapplikációk, vagy akár asztali alkalmazások fejlesztésére is.

Az applikáció elkészítéséhez szükségem volt olyan keretrendszerre, amely tartalmaz minden olyan elemet, amelyek elengedhetetlenek egy modern webapplikáció elkészítéséhez. Pl. kapcsolatot teremteni az adatbázissal, az applikáció biztonságát konfigurálni, a végpontok segítségével külső kapcsolatot létesíteni stb.

Azért esett a választásom a Spring keretrendszerre, mivel az egy robosztus, széles körben használt technológia, amelynek használatával hihetetlenül fel lehet gyorsítani a fejlesztést. Ez nem csak nagyon jól van dokumentálva, de tartalmaz beépített webszervert is. Sokszor minimális konfiguráció is elegendő ahhoz, hogy a Spring keretrendszerben el lehessen kezdeni a fejlesztést, hiszen a Spring mindent, amit csak tud, automatikusan konfigurál a fejlesztő számára.

### **2.2.4 H2**

Az applikációnak szüksége volt egy adatbázisra. Legelőször a mySQL adatbázis-kezelő rendszert szerettem volna felhasználni, de végül a H2 [8] rendszer mellett döntöttem.

A H2 egy memória alapú adatbázis, mely beépül magába a szoftverbe. Ezt nem lenne ajánlatos használni egy produkciós környezetben található web applikációban, viszont az általam készített applikáció valószínűleg soha nem lesz a publikum elé tárva. Ennek a tudatában választottam a H2 rendszert. A H2 adatbázis-kezelő rendszer sokkal könnyebb alkalmazni egy fejlesztői környezetben hiszen, amint már említettem, beépül magába a szoftverbe. Így nincs arra szükség, hogy a számítógépre külön telepítsek egy adatbázis-kezelő rendszert.

Ha esetleg a jövőben szükség lenne egy robusztusabb rendszerre akkor a jelenlegi H2 adatbázis-kezelő rendszert viszonylag könnyedén le lehet cserélni egy másikra a Spring keretrendszernek köszönhetően.

### **2.2.5 Swagger UI**

A Swagger [9] egy nyílt forrású eszköz, amely az OpenAPI Specifikáció köré épült. Ennek a technológiának segítségével tudtam készíteni egy interaktív API dokumentációt. Az interaktív API dokumentáció segítségével a felhasználó direkt API hívásokat küldhet a böngészőből a backendnek. Nagyon hasznos az applikáció dokumentációja, illetve manuális tesztelés szempontjából is előnyös eszköz.

### **2.2.6 HTML, CSS, TypeScript**

A frontend elkészítésében három alap technológiát használtam.

A HyperText Markup Language, avagy HTML [10], az internet központi leíró nyelve.

A Cascading Style Sheets, avagy CSS [11], egy stíluslap-nyelv, amely az előbb említett HTML nyelven írt dokumentum megjelenítésének leírására alkalmas. Ezeknek a technológiáknak a használata szinte elkerülhetetlen egy frontend applikáció elkészítésében.

A TypeScript [12] egy nyílt forrású programozási nyelv amely a JavaScript nyelven alapszik. A TypeScript az eredetileg dinamikus és gyengén típusos JavaScript nyelvet bővíti ki, így statikusságot és erős típusosságot ad a nyelvhez. Alkalmazására elsősorban azért volt szükség mivel az általam választott keretrendszer ezt a technológiát támogatja a hagyományos JavaScript nyelvvel ellentétben. Másodszorban kényelmesebb volt számomra egy típusos nyelvben fejleszteni, hiszen azokhoz voltam szokva, illetve korábban még nem használtam a JavaScript technológiát.

### **2.2.7 Node.js**

A Node.js [13] egy nyílt forrású, platform független, JavaScript futtató környezet. Segítségével képes voltam arra, hogy JavaScript kódot futtassak a web böngészőn kívül. Ezt általában a szerver oldalon alkalmazzák, illetve fejlesztés közben.

### **2.2.8 Angular**

Az Angular [14] egy nyílt forrású web applikáció keretrendszer, melyet az Angular csapat fejleszt a Google-nél.

Egy jól felépített keretrendszer használata hihetetlenül meg tudja könnyíteni a fejlesztést. Ezért is volt fontos, hogy egy széles körben elterjedt és jól dokumentált frontend keretrendszert válasszak. Kettő keretrendszer jöhetett szóba számomra. Az egyik az Angular volt, míg a másik a React. A végső választásom azért esett az Angular keretrendszerre, mivel az egy teljesebb rendszert alkot, mint a React. Ugyan a React nagyon nagy szabadságot nyújt a fejlesztőnek, ezzel egyidejűleg önmagában nem képes mindarra, amire az Angular.

### **2.2.9 Bootstrap 4.6**

A Bootstrap [15] egy ingyenes és nyílt forráskódú CSS keretrendszer. Ennek segítségével viszonylag könnyedén tudtam elkészíteni egy jól mutató és reszponzív applikációt.

### **2.2.10 NG Bootstrap**

Az NG Bootstrap [16] olyan komponenseket tartalmaz, amelyek az Angular keretrendszerben felhasználhatóak és a Bootstrap css keretrendszerre épülnek. Ennek egy alternatívája lehetne például a Angular Material komponens csomag.

### **2.2.11 Font Awesome**

A Font Awesome [17] egy font és ikon eszközkészlet tartalmaz, amely a CSS és Less technológiákon alapszik. Ennek a segítségével tudtam különböző ikonokat megjeleníteni az applikációban.

## **3 Architektúra/Tervezés**

Ebben a fejezetben be szeretném mutatni azokat a terveket amelyek alapján a projektet elkészítettem.

### **3.1 A konkrét feladat**

A feladat célja egy olyan web alkalmazás elkészítése, amelyet kutyatenyésztők használhatnak eladásra szánt kiskutyák reklámozására és amelyen érdeklődők kereshetnek maguknak házi kedvencet.

### **3.2 Követelmények**

Az alkalmazáson 3 felhasználói réteg kell, hogy jelen legyen. Az első a vendég felhasználók rétege, nekik nincs saját felhasználói fiókjuk, így nem is tudnak bejelentkezni az applikációba. Ez a réteg a házi kedvencet kereső személyekből áll. A második réteget a kutyatenyésztők és állatmenhelyek alkotják, ők egy átlagos felhasználói fiókkal kell rendelkezzenek. A harmadik réteget az adminisztrátorok alkotják. Ők egy speciális felhasználói fiókkal kell rendelkezzenek.

Mind a három felhasználói rétegnek rendelkeznie kell egy-egy saját felhasználói felülettel.

#### **3.2.1 Első felhasználói réteg**

Az első felhasználói réteg, vagyis a vendégek, nem tudhatnak új adatokat felvezetni a rendszerbe. Számukra az applikáció használata nagyon le kell legyen korlátozva. Az egyetlen dolog amire képesek kell, hogy legyenek az, hogy a meglévő hirdetések között kutathatnak. Ha találnak egy olyan hirdetést, amely számukra érdekesnek tűnik, akkor azt megnyithatják és így megtekinthetik a teljes hirdetést. Arra is lehetőségük kell legyen, hogy megtekintsék a hirdetés feladójának a profilját. Ha esetleg kapcsolatba szeretnének jutni a hirdetővel, akkor a profilján található információ segítségével ezt megtehetik, de ez a kapcsolatfelvétel már az applikáción kívül kell megtörténjen, például telefonon, elektronikus vagy postai levélben.

### 3.2.2 Második felhasználói réteg

Azok a felhasználói fiókok, amelyeket kutyaatenyésztők és állatmenhelyek használnak, képesek kell legyenek új hirdetések közzétételére, saját meglévő hirdetéseik módosítására, vagy akár törlésére is. Ezeken kívül, minden tenyésztőnek és állatmenhelynek lehetősége kell legyen arra is, hogy a saját profilján feltüntetett információkat módosítsa.

A tenyésztők és állatmenhelyek által készített hirdetéseknek többféle adatot kell tartalmazniuk. Első sorban minden hirdetésnek kell legyen egy címe. Ez kötelezően jelen kell legyen. Ezen kívül minden hirdetés tartalmazhat egy darab képet. Ezt a felhasználónak nem kötelező megadnia. A harmadik adat, amit egy hirdetés tartalmazhat az egy leírás. Itt a tenyésztő vagy menhely részletesebben is bemutathatja hirdetését. A negyedik adat, amit egy hirdetésnek tartalmaznia kell az maga a hirdetés státusza. Az, hogy a hirdetés publikus és a vendég felhasználók megtekinthetik vagy, hogy a hirdetés privát és csak a szerző tekintheti meg.

Az ötödik adat, amit egy hirdetésnek tartalmaznia kell, az egy címkékből álló lista. A rendszerben sok fajta címkének kell jelen lennie. Ezek közül a hirdető kiválaszthatja azokat, amelyek legjobban jellemzik hirdetését. Minden címke tartalmaz egy rövid szöveget, ezek lehetnek például: kutya, barna, tarka, agár, vizsla. A vendégeknek majd lehetőségük kell legyen arra, hogy ezeknek a címkéknek a segítségével célzottan is tudjanak keresni a hirdetések között.

Ezekon kívül még a rendszernek tudnia kell azt is, hogy melyik hirdetést mikor készítették és mikor módosították utoljára.

Hogyha a hirdető szeretné látni hogy hogyan néz ki a hirdetés a vendégek számára, akkor erre is lehetősége kell legyen.

Minden tenyésztőnek vagy állatmenhelynek kell legyen egy profilja, amin a személyére vonatkozó publikus információkat tárolhatja. Ezt a vendégek is megtekinthetik és az ezen lévő elérhetőségek alapján tudják majd felvenni a kapcsolatot a hirdetővel. A profilon megjeleníthető adatok:

- Egy darab profilkép.
- Egy név.



- Egy email cím.
- Egy postacím.
- Egy leírás amiben a hirdető bemutatkozhat vagy akár további elérhetőségeket tehet közzé.
- A platformra való csatlakozás dátuma.

Ezen kívül a hirdető még egy dologra képes kell legyen. Ha saját fiókjának a jelszavát ki szeretné cserélni, akkor arra is lehetősége kell legyen.

### **3.2.3 Harmadik felhasználói réteg**

A harmadik felhasználói réteg, vagyis az adminisztrátorok, minden egyes felhasználó adataihoz hozzá kell tudjanak férni.

Egy adminisztrátor készíthet új felhasználókat. Meglévő felhasználók adatait módosíthatja. Módosíthatja a jelszavát bármely felhasználónak, zárolhatja bármely felhasználó fiókját, hogy azon ne tudjon többet adatokat módosítani, feltölteni. Beállíthatja bármely fiók lejáratát, illetve tagsági idejét. Ezeken kívül jogosultságok kiosztására és megvonására is képes kell legyen.

Az adminisztrátornak lehetősége kell legyen arra, hogy megtekintse bármely felhasználó hirdetéseit és azokat módosíthassa. De nincs lehetősége törölni hirdetéseket.

Ezen kívül az adminisztrátor bármely felhasználó profilján lévő adatait megváltoztathatja.

Az adminisztrátor második fontos feladata, a felhasználók menedzselése után a címkék kezelése. Minden adminisztrátor készíthet, módosíthat vagy akár törölhet címkéket.

Egy címke két adatot kell tartalmazzon. Az első a címke neve és a második egy leírás, amiben pontasítani lehet azt, hogy mit is takar pontosan a címke neve.

## **3.3 A rendszer architektúrája**

A rendszert két fő szerkezeti egységre lehet lebontani:

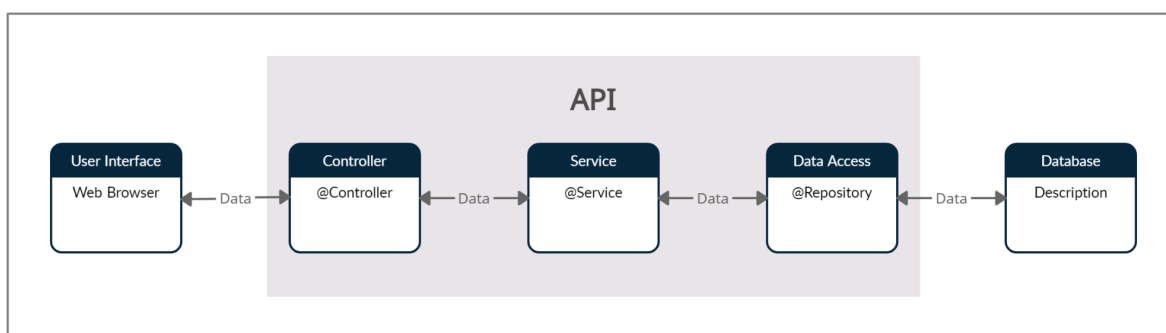
- A Backend, más néven a szerver
- A Frontend, vagy máshogy nevezve a kliens.

### 3.3.1 A Backend architektúrája

A backend architektúráját megpróbáltam minél jobban igazítani a hagyományos REST applikáció architektúrához dizájn [18] és biztonság [19] szempontjából.

A backend architektúráját három fő részre lehet lebontani.

- A kontroller réteg (controller layer)
- A szerviz réteg (service layer)
- Az adatelérési réteg (data access layer)



*1. ábra - A backend architektúrájának vázlata*

#### 3.3.1.1 A kontroller réteg

A kontroller réteg célja, hogy kapcsolatot teremtsen a frontendel, a klienssel.

Feladatai közé tartozik:

- Kérések fogadása a kientől.
- Adathordozó objektummá konvertálni a kientől beérkezett adatokat.
- A keletkezett adathordozó objektumok validálása.
- Az adathordozó objektumokat átkonvertálni egy olyan objektummá, amit az üzleti logikában használunk, például egy entitás objektummá (entity object)
- Hívásokat intéz a szerviz réteghez.
- Válaszol a kliensnek egy adathordozó objektum segítségével, amelyet egy entitás objektumból konvertált át, vagy egy más olyan objektumból, amelyet az üzleti logikában használunk.

### 3.3.1.2 A szerviz réteg

A szerviz réteg az a része az aplikációnak, ahol az üzleti logika megtalálható.

Feladatai közé tartozik:

- Az összes üzleti logika elvégzése.
- Az adatelérési réteghez történő hívások segítségével entitás objektumokat tud módosítani, létrehozni, törölni.
- Válaszol a kontroller rétegnek egy entitás objektum vagy egy másik objektum segítségével.

### 3.3.1.3 Az adatelérési réteg

Az adatelérési réteg célja az adatbázissal való kommunikációnak a megteremtése.

Feladata az, hogy lekérdezéseket, illetve adatmanipulációt végezzen az adatbázison és entitás objektumok formájában továbbítsa a választ a szerviz rétegnek.

## 3.3.2 A frontend architektúrája

A frontend az Angular keretrendszerben lett készítve, így az architektúrája is az Angular keretrendszer által jól meghatározott stílust és kereteket követi.

A backend-hez hasonlóan a frontend-et is fel lehet osztani rétegekre. Mivel az általam elkészített applikáció viszonylag egyszerű, így nincs arra szükség, hogy egy komplex architektúrát alkalmazzak.

A keretrendszer úgynevezett angular komponensek segítségével építi fel az applikációt. Ezek a komponensek az általam említett rétegekből kettőt tartalmaznak magukban.

### 3.3.2.1 A prezentációs réteg

A prezentációs rétegben található meg az, amit a felhasználó lát. Itt képernyő elemek vagy az úgynevezett *ui kontrollok*, találhatóak meg, amelyek html nyelven vannak megírva. Ezeknek a stílusa pedig általában .css nyelven van meghatározva.

Az angular keretrendszerben prezentációs réteget általában a **.component.html** kiterjesztésű fájlokban találhatjuk meg. Ezenkívül megtörténhet, hogy egy stílus definíciókat

tartalmazó fájl is hozzátartozik az előbb említett html fájlhoz. Ennek a stílus definíciókat tartalmazó fájlunk egy lehetséges fájl kiterjesztése a **.component.css**.

Amikor a felhasználó interakcióba lép ezekkel az elemekkel, akkor eventek generálódnak, amelyek változásokat idézhetnek elő az oldal tartalmában.

### **3.3.2.2 A kontroller réteg**

Itt található az a kód, amelyik reagál a prezentációs rétegben található elemekre. Ez az a hely, ahol a ui kontrollok működése le van programozva. Az angular keretrendszerben **.component.ts** kiterjesztéssel rendelkező fájlok felelnek meg ennek a rétegnek.

Az ennek a rétegnek megfelelő fájlok és a prezentációs rétegnek megfelelő fájlok együttesen egy-egy angular komponenst alkotnak.

### **3.3.2.3 A szerviz réteg**

A szerviz réteg a backend végpontjaival tartja a kapcsolatot. Ennek a rétegnek megfelelő fájlok az angular keretrendszerben általában a **.service.ts** fájl kiterjesztéssel vannak ellátva.

## **3.4 Az adatbázis séma tervezete**

Az applikáció által felhasznált adatok egy hagyományos relációs adatbázisban lesznek eltárolva. Ennek az adatbázisnak szeretném röviden bemutatni a séma tervezetét.

Az adatbázisban összesen kilenc tábla található. Ezeknek nevei USER, ROLE, USER\_ROLES, PROFILE, PROFILE\_IMAGE, POST, POST\_IMAGE, TAG, POST\_TAGS.

A USER táblában a felhasználók adatai találhatóak meg. Például jelszó, felhasználónév stb.

A PROFIL tábla a felhasználók profilján található információkat tartalmazza. Például a felhasználó teljes nevét, elérhetőségeit stb.

A PROFILE\_IMAGE tábla a felhasználók profilján található képeket tartalmazza.

A ROLE tábla azokat a szerepköröket tartalmazza, amelyeket a felhasználók betölthetnek, ez lehet például adminisztrátori, vagy átlag felhasználói szerepkör.

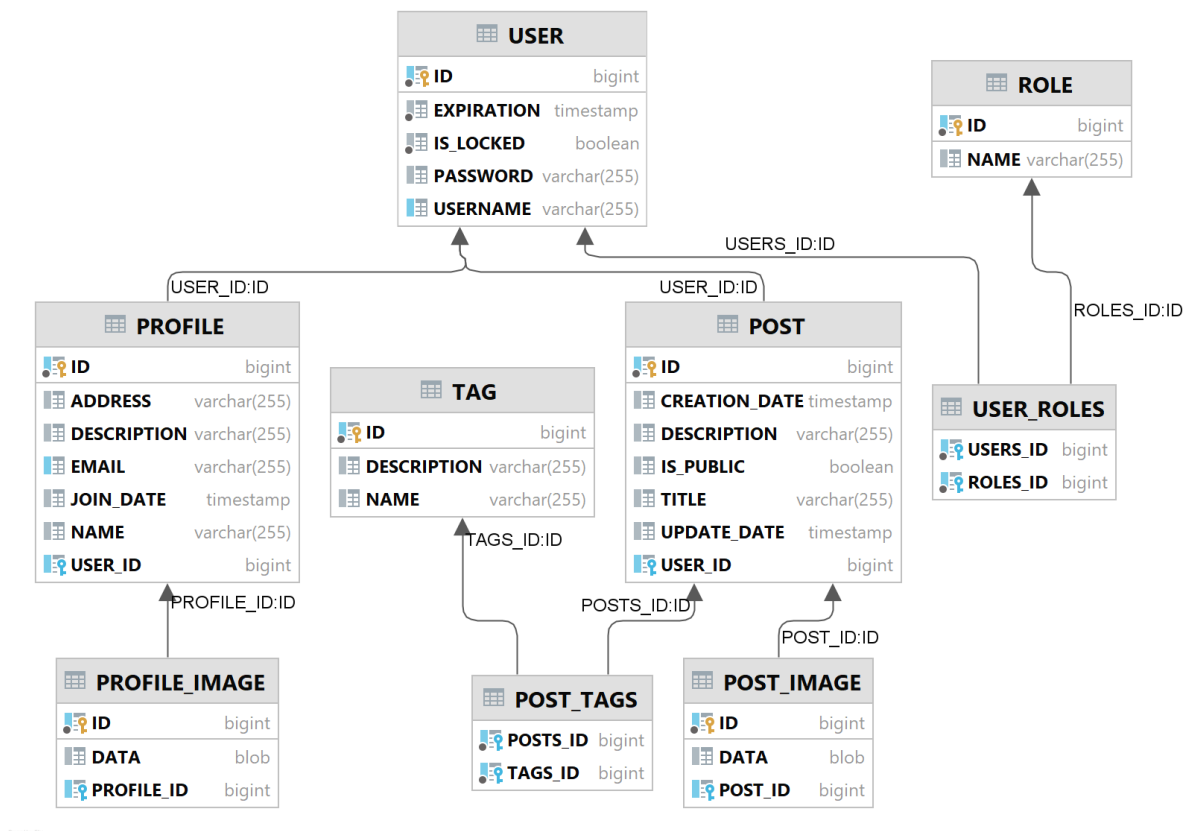
A USER\_ROLES tábla egy kereszttábla, ez tartalmazza azokat a szerepeket, amelyeket egy bizonyos felhasználó betölt. Egy felhasználó akár több szerepkört is betölthet.

A POST tábla a felhasználók által készített hirdetések információit tartalmazza. Például a hirdetés címét, leírását, módosításának dátumát stb.

A POST\_IMAGE tábla a hirdetésekhez hozzárendelt képeket tartalmazza.

A TAG tábla címkéknek az információit tartalmazza. Ezeknek a címkéknek a tárgya az olyan szavak lehetnek, mint például a kutya, macska, fekete, fehér, labrador, farkaskutya stb. Ezeknek a célja, hogy a hirdetésben feltüntetett állatokat jellemezze.

A POST\_TAG tábla egy kereszttábla, ez azt tartalmazza, hogy mely címkék melyik hirdetésekhez lettek hozzárendelve.



2. ábra - Az adatbázis séma

## 4 Önálló munka bemutatása

Ebben a fejezetben be szeretném mutatni az általam elkészített applikációt és annak felépítését.

### 4.1 A Backend bemutatása

#### 4.1.1 Az adatbázis

A H2 adatbázis-kezelő rendszer konfigurálását viszonylag könnyedén sikerült elvégezni a spring keretrendszernek, illetve azon belül a spring jpa függőségnek köszönhetően. A szükséges konfigurációkat az applikáció properties file-jában adtam meg.

```
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.datasource.url=jdbc:h2:file:./data/demo
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop
spring.h2.console.enabled=true
spring.h2.console.path=/admin/h2-console
```

Az első sorban az adatbázis kezelő rendszer driver-ját volt szükséges megadnom.

A második sorban az adatbázis kezelő rendszer dialektusát.

A harmadik sorba állítottam be azt, hogy hol tárolja az adatbázis az adatokat. Itt két lehetőségem volt. Az első az, hogy egy memória alapú adatbázisként alkalmazom a h2 adatbázis kezelő rendszert, a második lehetőség pedig, hogy egy file alapú adatbázis kezelő rendszerként alkalmazom. Természetesen számomra csak a file alapú rendszer jöhetett szóba, hiszen a memória alapú felhasználás nem képes perzisztensen megőrizni az adatokat a rendszer újraindítását követően.

A következő két sorban egy felhasználónevet és egy jelszót szolgáltatam a rendszernek. Mivel csak lokálisan használom az applikációt nem volt szükségem bonyolultabb felhasználónév és jelszó párosra, viszont egy produkcióban lévő applikáció esetében erre komolyabb figyelmet kell fordítani.

A következő beállításban a create-drop kulcsszó segítségével értem el azt, hogy a rendszer szabadon hozhasson létre és törölhessen táblákat az adatbázisból. Ez egy

produkcióban lévő alkalmazás esetében nem lenne jó ötlet. Akkor már adatbázis migrációt is szükséges lenne implementálni.

Az utolsó két sorban az adatbázis-kezelő rendszer úgynevezett konzolának használatát engedélyeztem, illetve megadtam az utat, amelyen keresztül el lehet érni ezt a konzolt. Erre azért van szükség mert a h2 adatbázis-kezelő rendszer egy az applikációba beépített rendszer, nem egy önálló, nem egy teljesen különálló rendszer.

## 4.1.2 Az adatelérési réteg

### 4.1.2.1 Az adat modellek bemutatása

Az adatelérési réteg tartalmazza azokat az osztályokat amelyek egy egy adatbázis táblát modelleznek. Ezeket az osztályokat hívjuk entity osztályoknak.

Több ilyen osztályt is létrehoztam az applikáció elkészítése során. Ilyenek például a következő osztályok: Post, PostImage, Profile, ProfileImage, Role, Tag, User. Ezek közül be szeretném mutatni a Profile osztályt.

```
@Entity
@Getter
@Setter
@NoArgsConstructor
public class Profile {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @Column(unique = true)
    private String email;
    private String address;
    private String description;
    private Date joinDate;

    @OneToOne(mappedBy = "profile", cascade = CascadeType.ALL, fetch =
FetchType.EAGER)
    private ProfileImage profileImage;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn
    private User user;

    public Profile(User user) {
```

```

        this.joinDate = new Date();
        this.user = user;
        this.profileImage = new ProfileImage(this);
    }
}

```

A Profile osztálynak nyolc privát attribútumát definiáltam: id, email, address, description, joinDate, profileImage és user. Az ezekhez tartozó getter és setter metódusokat, egy-egyombok annotáció segítségével készítettem el. Ezt úgy tudtam megtenni, hogy az osztályt magát jelöltem meg egy @Getter és egy @Setter annotációval. A @Getter annotációval az indikáltam aombok függőségnek, hogy az osztály összes getter metódusát generálja le automatikusan. A @Setter annotáció ugyanúgy működik, mint az előző, azzal a különbséggel, hogy felhasználásával aombok függőség az osztály setter metódusait generálja le.

Ezeknek aombok annotációknak a felhasználásának köszönhetően nem volt már többé arra szükségem, hogy minden egyes osztályban megírjam a getter és setter metódusokat, ha csak az adott metódusnak nem kellett eltérnie a normától.

A Profile osztályt el láttam még két konstruktorral is. A két konstruktor metódus közül az egyik egy User objektumot vár el, mint paramétert. Míg a másik konstruktor nem vár el semmiféle paramétert sem. Ezt az alapértelmezett, üres konstruktor metódust kizárólag a Spring JPA számára hoztam létre. Fontos lenne még azt is megemlíteni, hogy nem manuálisan hoztam létre ezt az üres konstruktor metódust, hanem a @NoArgsConstructorombok annotáció segítségével.

Más esetekben is megtörtént az applikáció elkészítése közben, hogy hasonlóan rövid és gyakran használt metódusokat helyettesítettem egy-egyombok annotációval. Ezeknek az annotációknak köszönhetően nagy mértékben le tudtam rövidíteni a forráskódot, illetve az sokkal tisztábbá és áttekinthetőbbé vált.

A Spring keretrendszer számára egy @Entity annotációval tudtam jelezni azt, hogy a Profile osztályt egy JPA entity-ként kell kezelnie. Vagyis a Profile osztály az adatbázis egyik tábláját reprezentálja.

Nem volt szükségem arra, hogy a lemodellezett adatbázis tábla nevét is megadjam, hiszen az adatbázis táblák nevei és az entity osztályok nevei alapértelmezett módon megegyeznek. Ez ellen nekem nem volt semmiféle kifogásom sem, hiszen nem egy meglévő



adatbázishoz akartam igazítani az applikációt. Hanem a célom az volt, hogy maga az applikáció építsen fel számomra egy teljesen új adatbázist az entity osztályok alapján.

A következőkben be szeretném mutatni azokat az annotációkat amelyekkel el láttam az attribútumokat.

A Profile osztály id attribútumát az @Id annotációval láttam el, így a JPA felismeri azt mint az objektum ID-át. Az id attribútumot még elláttam a @GeneratedValue annotációval is, ennek köszönhetően a JPA tudja, hogy az id attribútumot automatikusan kell le generálja.

Az email attribútumot ugyancsak elláttam egy annotációval. A @Column(unique = true) annotáció segítségével tudtam jelezni a JPA számára azt, hogy az adott attribútum egy egyedi értéket kell tartalmazzon. Vagyis az adatbázisban is egy egyedi megszorítást kap az email oszlop.

Ezen kívül még a user és a profileImage attribútumokat is elláttam annotációkkal. A @ManyToOne annotáció segítségével tudtam jelezni azt a JPA-nak, hogy a megjelölt attribútumok nem egy egy adatbázisoszlopot reprezentálnak, hanem a Profile tábla és egy egy másik adatbázis tábla közötti, egy az egyhez való, kapcsolatot. A @JoinColumn annotáció pedig a @ManyToOne annotációval párosítva azt jelzi, hogy a Profile entitás id attribútuma tölti be a kulcs szerepét a kapcsolatban.

A többi attribútumot, amelyeket nem jelöltem meg annotációkkal, azokat a JPA egy megegyező nevű adatbázisoszlophoz köti.

#### 4.1.2.2 A Repository interfészek bemutatása

Minden egyes @Entity annotációval ellátott osztálynak készítettem egy repository interfészt. Ezeket az interfészeket a JPA futás közben arra használja fel, hogy egy repository implementációt készítsen. Ezek pedig tulajdonképpen nem mások, mint az adatelérő objektumai az applikációnak, más néven *data access object* vagy röviden csak DAO.

Minden egyes entity osztály számára készítettem egy repository interfészt: PostRepository, PostImageRepository, ProfileRepository, ProfileImageRepository, RoleRepository, TagRepository és UserRepository. Ezek közül be szeretném mutatni a UserRepository interfészt.

@Repository

```
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}
```

Ezt az interfészt elláttam egy `@Repository` annotációval, ezzel azt tudtam indikálni a Spring keretrendszernek, hogy az interfészt hogyan kezelje. Tulajdonképpen az adatelérési réteg minden logikája egy `@Repository` annotációval kell legyen ellátva.

A `UserRepository` interfészt a `JpaRepository` interfészből származtatam. A `JpaRepository` interfész paramétereinek a `User` és a `Long` osztályt adtam meg. A `User` osztályt már korábban már el láttam egy `@Entity` annotációval. Illetve a második paraméter, a `Long` osztály, nem más mint a `User` osztály id attribútumának típusa.

Ennek az öröklődésnek köszönhetően, a `UserRepository` szert tett több olyan metódusra is, amelyekkel keresni, illetve akár perzisztensen menteni és törölni is tud a `User` entity objektumok közül. Ezek közül be szeretnék mutatni néhányat. A `save` metódus perzisztensen mentésre alkalmas. A `findById` metódus egy id alapján képes keresni az adott entity objektumok között és a megegyező objektumot visszatéríti. A `findAll` metódus minden egyes entitást visszatérít. A `delete` metódus pedig törlésre képes. Természetesen ezek a metódusok mind csak a repository számára megadott entity objektumokra alkalmazhatóak.

Ezekon a metódusokon kívül a Spring JPA lehetőséget adott számomra arra is, hogy saját kereső metódusokat készítsek. Sőt több féle képpen is lehet ilyeneket készíteni.

A `findByUsername` egy egyszerűbb kereső metódus. Ez egy `String` objektumot vár el mint argumentum, amely egy felhasználó nevét kell tartalmazza és egy opcionális `User` objektumot térít vissza. A Spring JPA a metódus nevéből tudja azt automatikusan megállapítani, hogy mi alapján akarok keresni a `User` entity objektumok között.

Sajnos ezzel a módszerrel viszont nehéz lett volna bonyolultabb lekérdezési metódusok megvalósítani. Emiatt készítettem két criteria repository osztályt. Ezeket az osztályokat úgy neveztem el, hogy `UserCriteriaRepository` és `PostCriteriaRepository`.

Néhány rövid szót szeretnék ejteni a `UserCriteriaRepository` osztályról. Ezt az osztályt is elláttam egy `@Repository` annotációval, hiszen az adatelérési réteg logikájának egy részét tartalmazza, illetve azt indikálom ezzel a Spring keretrendszernek, hogy az adott osztályt egy repository komponensként kezelje. Ebben az osztályban a `User` entity objektumoknak egy bonyolultabb lekérdezési formáját implementáltam a Criteria API segítségével. Sajnos ennek

a módszernek az volt a hátránya, hogy meglehetősen bőbeszédű logikát kellett írnom, amely tele van úgynevezett *boilerplate* kóddal.

A `findAllWithFilters` metódus az osztály egyetlen publikus metódusa, az osztálynak csak ezt az egy metódust szeretném bemutatni.

```
public Page<User> findAllWithFilters(UserPageCriteria userPage,
UserSearchCriteria userSearchCriteria) {
    CriteriaQuery<User> criteriaQuery = criteriaBuilder.createQuery(User.class);
    Root<User> userRoot = criteriaQuery.from(User.class);
    Predicate predicate = getPredicate(userSearchCriteria, userRoot);
    criteriaQuery.where(predicate);
    setOrder(userPage, criteriaQuery, userRoot);

    TypedQuery<User> typedQuery = entityManager.createQuery(criteriaQuery);
    typedQuery.setFirstResult(userPage.getPageNumber() * userPage.getPageSize());
    typedQuery.setMaxResults(userPage.getPageSize());

    Pageable pageable = getPageable(userPage);
    long usersCount = getUsersCount(predicate);
    return new PageImpl<>(typedQuery.getResultList(), pageable, usersCount);
}
```

Ez egy lekérdezést hajt végre és végül egy `Page` objektumot térít vissza, amely `User` entity objektumokat tartalmaz. A metódusnak két argumentuma van, a `userPage` és a `userSearchCriteria`.

A `userSearchCriteria` argumentum egy `UserSearchCriteria` objektumot vár el. Ennek `UserSearchCriteria` osztálynak három attribútumát definiáltam: `id`, `username`, `isLocked`. Tulajdonképpen ez a három attribútum definíciói a `User` entity osztályban is megtalálható. Ezeket az adatokat a metódus úgy alkalmazza, mint egy-egy megszorítást adatbázis lekérdezésben. Minden egyes megadott attribútum egy-egy megszorítást jelent a lekérdezésben és a válaszul kapott entity objektumoknak ezeknek eleget kell tenniük. Ha például az `id` és a `username` attribútumokat megadom, míg az `isLocked` attribútumot nem, akkor a metódus csak az első kettő attribútum szerint fog keresni.

A `userPage` argumentum egy `UserPageCriteria` objektumot vár. Ez olyan adatokat tartalmaz, amelyek a visszatérítendő `Page` objektumot befolyásolják. Mint például, hogy a `userSearchCriteria` argumentum által meghatározott `User` objektumok pontosan melyik részhalmazát tartalmazza a visszatérítendő `Page` objektum. Illetve azt, hogy a visszatérítendő `User` objektumok milyen sorrendbe legyenek rendezve.

### 4.1.3 A szervíz réteg

A szervíz réteg elsősorban szervíz interfészeket tartalmaz és azoknak az implementációját. Összesen hét szervíz interfészt készítettem, ezek sorban: PostImageService, PostService, ProfileImageService, ProfileService, RoleService, TagService, UserService. Tulajdonképpen minden egyes entity osztálynak készítettem egy szervíz interfészt. Ezek az interfészek nagyon egyszerűek, itt csak azt definiáltam, hogy milyen metódusokat kell a későbbiekben a szervíz implementációjának megvalósítania.

A következő sorokban mutatom be a UserService interfészt. Összesen öt metódus definíciót tartalmaz, ezeket későbbiekben implementáltam is.

```
public interface UserService {  
    Page<User> getAllUsers(UserPageCriteria userPage, UserSearchCriteria  
userSearchCriteria);  
    User createUser(User userRequest);  
    User partialUpdateUser(long id, User userRequest);  
    User getUserById(long id);  
    User getUserByUsername(String username);  
}
```

A szervíz interfészeket egy egy szervíz osztály elkészítésével implementáltam. Ezekben található meg az applikáció tulajdonképpeni üzleti logikája.

A következő sorokban be szeretném mutatni a UserServiceImpl osztályt. Ezt az osztályt a UserService interfész implementálásával hoztam létre. A UserServiceImpl osztályhoz hasonlóan implementáltam a többi szervíz interfészt is.

A UserServiceImpl osztályt, mint minden egyes más szervíz osztályt is, el láttam egy @Service annotációval, ezzel indikálom a Spring keretrendszernek, hogy az adott osztályt egy szervíz komponensként kezelje.

A UserServiceImpl osztályt három attribútummal hoztam létre, mindhármát elláttam egy-egy @Autowired annotációval. Ezzel az annotációval azt indikálom a Spring keretrendszer számára, hogy a megjelölt attribútumokat automatikusan injektálja a nekik megfelelő objektummal. Ezeket az attribútumokat be szeretném mutatni a következő sorokban.

```
@Autowired  
private UserRepository userRepository;
```

```

@Autowired
private UserCriteriaRepository userCriteriaRepository;
@Autowired
private PasswordEncoder passwordEncoder;

```

Az attribútumok közül kettőnek a típusa egy repository osztály, amelyeket már korábban bemutattam. A harmadik attribútum típusa a PasswordEncoder nevű interfész, ez az interfész a Spring keretrendszer része. Ebbe az attribútumba egy olyan bean-t injektál a Spring keretrendszer, amelyet én az applikáció fő osztályában, a PetShopApplication nevű osztályban definiáltam.

A UserServiceImpl osztály elkészítése közben implementáltam az összes metódust amelyet a UserService interfészben definiáltam. Ezeket a metódusokat az @Override annotációval láttam el, ezzel jelezve, hogy az adott metódusok felül írják az interfészben található metódus definíciókat. Összesen öt ilyen metódust tartalmaz a UserServiceImpl osztály, ezeknek nevei sorban: getAllUsers, createUser, partialUpdateUser, getUserById, getUserByUsername. A következő sorokban be szeretném mutatni a getAllUsers metódust.

```

@Override
public Page<User> getAllUsers(UserPageCriteria userPage, UserSearchCriteria
userSearchCriteria) {
    return userCriteriaRepository.findAllWithFilters(userPage, userSearchCriteria);
}

```

Ezt a metódust csak pár sorban implementáltam. Célja csupán az, hogy a kontroller és az adatelérési rétegeket összekösse. Két paramétert vár, ezekkel a paraméterekkel meghívja a userCriteriaRepository objektum findAllWithFilters metódusát, melyet már korábban bemutatam. Végül a findAllWithFilters metódus által visszatérített Page objektumot a getAllUsers metódus ugyancsak visszatéríti.

Be szeretném még mutatni a getUserById metódust definícióját.

```

@Override
public User getUserById(long id) {
    return userRepository.findById(id)
        .orElseThrow(() -> new NoSuchElementException("User by id " + id + "
not found"));
}

```

Ez a metódus hasonló az előzőhöz, azzal a különbséggel, hogy ez a userRepository objektum metódusai közül hív meg egyet. Illetve, hogyha nem sikeres a meghívott findById metódus által intézett adatbázis lekérdezés, akkor a getAllUsers metódus egy hibát dob.

A getUserByUsername metódus hasonlóképpen működik a getUserById metódushoz azzal az apró eltéréssel, hogy az argumentuma nem egy long típusú id, hanem egy string típusú username. Így az általa meghívott repository metódus sem egy id alapján intézi az adatbázis lekérdezést, hanem egy username alapján.

Két metódus maradt még hátra, amelyekről még nem beszéltünk, ezek a createUser és a partialUpdateUser metódusok. Ezeknek a metódusoknak az implementálása tartalmilag hosszabbra sikerült mint az előzőek metódusok. Ez annak köszönhető, hogy ezek a metódusok már nem csak az adatelérési réteg és a kontroller réteg közötti kapcsolatot valósítják meg, hanem üzleti logikát is tartalmaznak.

A createUser metódus célja egy új felhasználó, vagyis egy új User entitást elmentése. Először ellenőrzi, hogy az új User entitás megfelel-e bizonyos követelményeknek. Például, hogy a felhasználó neve használatban van-e már vagy még szabad. Vagy például azt, hogy a jelszó eléggé komplex-e. Ha nem sikerült túljutnia a megszorításokon, akkor egy hibát dob a metódus, ha pedig sikerült, akkor elmenti az új User entitást.

A partialUpdateUser metódus hasonlóan működik. Viszont itt egy már meglévő User entitást módosít a metódus, persze csak abban az esetben módosít értékeket, ha azok eleget tesznek a metódusban megtalálható megszorításoknak.

#### **4.1.4 A kontroller réteg**

A kontroller rétegben találhatóak a kontroller osztályok. Ezeknek a segítségével tudtam megvalósítani a kapcsolatot a backend és a frontend között. Összesen kilenc ilyen osztályt készítettem, ezekből hét egy-egy entitás kezelésére használatosak. Ezek sorban: PostController, PostImageController, ProfileController, ProfileImageController, RoleController, TagController, UserController osztályok. Az ezekben az osztályokban definiált metódusok megfelelnek egy-egy hagyományos REST végpontnak.

A maradék két osztály pedig: AuthenticationController, ServiceController. Ezek specializáltabb célokra vannak kitalálva

A következő sorokban be szeretném mutatni a UserController osztályt.

A UserController osztályt, mint minden egyes más kontroller osztályt is, el láttam egy `@RestController` annotációval, ezzel indikálom a Spring keretrendszernek, hogy az adott osztályt egy controller komponensként kezelje.

Ezen kívül még elláttam az osztályt egy `@RequestMapping("/user")` annotációval is. Ezzel állítom be az osztály által definiált végpontok elérési útvonalát.

A UserController osztályt három attribútummal hoztam létre, mindhármát el láttam egy `@Autowired` annotációval. Ezeket az attribútumokat szeretném bemutatni a következő sorokban.

```
@Autowired
UserService userService;
@Autowired
DtoEntityConverter converter;
@Autowired
AuthenticationContext authenticationContext;
```

Az első attribútum típusa a UserServiceImpl osztály, amelyet már korábban bemutattam. A második attribútum típusa a DtoEntityConverter osztály. Ezt az osztályt abból a célból készítettem el, hogy Entity objektumokat DTO objektumokba tudjam konvertálni könnyedén, illetve DTO objektumokat Entity objektumokba. Ezt a modelmapper [20] könyvtár segítségével tudtam implementálni. A harmadik attribútum típusa az AuthenticationContext osztály. Ez a későbbiekben az aktuális felhasználóról fog információkat szolgáltatni.

Az osztály elkészítése során több metódust is definiáltam. Ezek a metódusok mind egy rest végpontnak felelnek meg. Minden egyes metódus el van látva egy olyan annotációval, amely jelzi a spring keretrendszernek, hogy az adott metódus egy fajta végpontot definiál. A `@GetMapping` annotáció például egy GET végpontot definiál, míg egy `@PostMapping` annotáció egy POST végpontot.

A következő sorokban be szeretném mutatni a createUser metódust.

```
@Operation(
    summary = "Create a new User.",
    description = "Can be used by Admin.",
    security = @SecurityRequirement(name = "bearerAuth"))
```

```

@IsAdmin
@PostMapping
public UserDto createUser(@Valid @RequestBody UserCreateDto dto) {
    User user = converter.convertToEntity(dto, User.class);
    User responseUser = userService.createUser(user);
    return converter.convertToDto(responseUser, UserDto.class);
}

```

A metódus egy `@Operation` annotációval kezdődik, ezt a swagger nevű eszköz fogja felhasználni a későbbiekben egyféle dokumentáció elkészítésére.

A következő az `@IsAdmin` annotáció, ez egy saját készítésű annotáció, amelyet abból a célból írtam, hogy le tudjam ellenőrizni azt, hogy a beérkezett kérés egy admin szerepkörrel rendelkező felhasználótól jött-e.

A következő annotációról már volt szó. A `@PostMapping` azt jelzi, hogy a metódus egy POST végpontot definiál.

A metódus egyetlen paraméterrel rendelkezik, ez egy `UserCreateDto` osztály típusú objektumot vár el. Ezt a paramétert elláttam két annotációval. A `@Valid` annotáció biztosítja, hogy a kapott objektum a `UserCreateDto` osztályban megadott validációs megszorításoknak megfelel. A `@RequestBody` pedig azt jelzi, hogy a paraméter értékeit a metódus által definiált végponthoz tett kérdések testéből, más néven az úgynevezett *body*-ből kapja.

A metódusban található kódrészlet viszonylag egyszerű. A megadott paramétert egy `dto` objektumból átkonvertálja egy `entity` objektumba. Ezt az `entity` objektumot a szerviz osztály segítségével kimentti az adatbázisba, majd a válaszul kapott `entity` objektumot visszakonvertálja egy `dto` objektumba és válaszként visszaküldi a kérésre.

Pár szót szeretnék még ejteni az adat küldési objektumokról, más néven *data transfer object* vagy röviden csak *DTO*. Ezeket az objektumokat a kliens és a szerver közötti adatok küldözgetésére használom. Használatuk nem kimondottan kötelező, de nagyon ajánlatos, hiszen így nem kerülnek az entitás objektumok a szerveren kívülre, illetve így nem kell félni attól, hogy olyan adatokat osztunk meg a frontend-el amelyek kényesek lehetnének. Hátránya, hogy minden egyes *DTO* osztályt definiálni kell, így egy kicsit bőbeszédű ezek implementációja.

Be szeretném mutatni a `UserCreateDto` osztályt röviden.



```

@Getter
@Setter
public class UserCreatedDto {
    @NotBlank
    @Size(max = 50, message = "user username size validation criteria not met")
    private String username;
    @NotBlank
    @Size(max = 255, message = "user password size validation criteria not met")
    private String password;
    @NotNull
    private Boolean isLocked;
    @NotNull
    private Date expiration;
    private List<RoleDto> roles;
}

```

Amit érdemes megfigyelni az, hogy az osztály attribútumait elláttam olyan annotációkkal, amelyek validációs feltételeket definiálnak. Ezek a DTO objektumok validálásánál vannak felhasználva.

Be szeretném még mutatni a `getUserById` metódust is.

```

@Operation(
    summary = "Get User by it's Id.",
    description = "Can be used by Owner or Admin.",
    security = @SecurityRequirement(name = "bearerAuth"))
@PreAuthorize("@ownerChecker.checkUser(#id) || hasAuthority('ROLE_ADMIN')")
@GetMapping("/{id}")
public UserDto getUserById(@PathVariable("id") @NotNull Long id) {
    User user = userService.getUserById(id);
    return converter.convertToDto(user, UserDto.class);
}

```

Ez a metódus hasonlóan van definiálva az előzőhöz, de van köztük pár különbség amelyet fontos megemlíteni. Ez a metódus el van látva a `@GetMapping("/{id}")` annotációval. Ez egyrészt azt jelenti, hogy ez egy GET végpontot definiál. De emellett azt is jelenti, hogy a végpont elérési útvonalát is megváltoztattam, hiszen az attribútumot nem hagytam üresen. A metódus egyetlen paramétere el van látva, a `@NotNull` annotációval, így kötelezően kell legyen értéke. Emellett el van még látva a `@PathVariable("id")` annotációval is. Ez azt jelzi, hogy az `id` nevű paraméter és az útvonalban található `id` változó megfelelnek egymásnak, ez az annotáció hasonló a korábban említett `@RequestBody` annotációhoz.

A metódus el van még látva a viszonylag hosszú `@PreAuthorize("@ownerChecker.checkUser(#id)||hasAuthority('ROLE_ADMIN')`

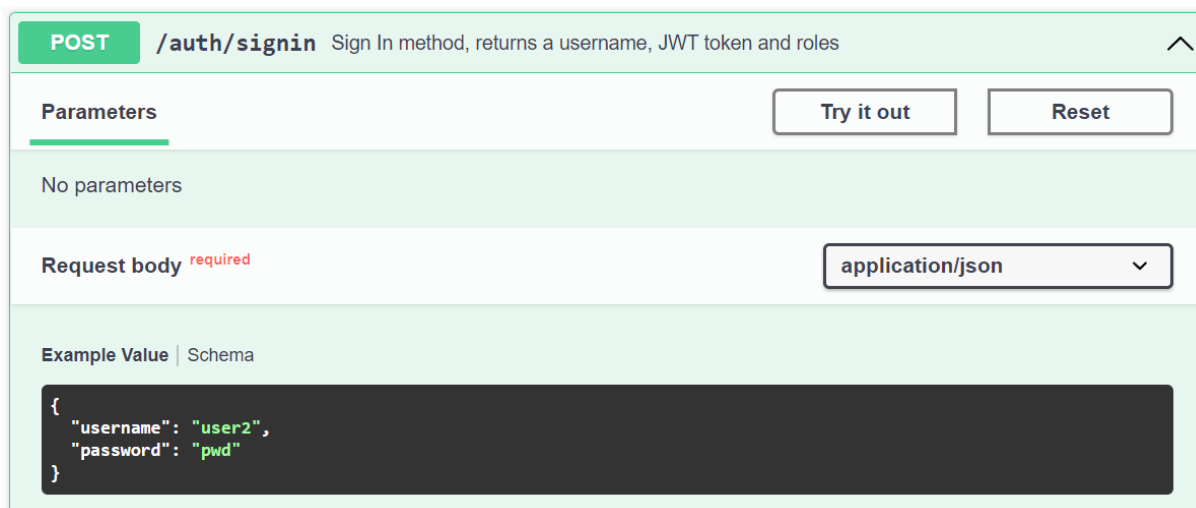
annotációval is. Ez biztosítja, hogy csak olyan felhasználókat szolgáljon ki a metódus által kreált végpont, akik vagy a lekért adatok tulajdonosai vagy adminisztrátori jogokkal rendelkeznek.

#### 4.1.5 Autentikáció, autorizáció

Abban az esetben ha az applikációt kívülről, az egyik végpontján keresztül szeretnénk elérni és az a végpont nem érhető el nyilvánosan, akkor a klienset identifikálni kell. Úgy gondoltam, hogy a JWT tokenek használata egy jó módszert jelentene ennek a problémának a megoldására.


Egy JWT token-re úgy tehet szert a kliens, hogy a felhasználónevét és a jelszavát megadva bejelentkezik a *signin* nevű végponton. Ezt a végpontot az AuthenticationController osztály createAuthenticationToken metódusa hozza létre. Miután a felhasználó elküldte helyes adatait, a végpont egy tokent térít vissza, melyet a felhasználó eltárolhat és a későbbiekben felhasználhat.

Az alábbi képeken látható, hogy a swagger eszköz segítségével végrehajtott bejelentkezés milyen paraméterekkel valósult meg.



3. ábra - Egy swaggeren keresztül intézett post kérés a signin végpontra

Illetve azt is láthatjuk, hogy milyen választ adott erre a backend.

Server response	
Code	Details
200	<div>Response body</div> <pre>{   "username": "user2",   "jwt": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJlc2VyMiIsInJvbGVzIjpbIjJPTGVfVVNFUiJdLCJpYXQiOiJlE2NTkzODY1NjIsImV4cCI6MTY1OTk5MTM2Mn0.yDRlP_6hpPqRyG_Bqgg-JzsWjYXrkdytX-iBIGSw6xo",   "roles": [     "ROLE_USER"   ] }</pre> <div>   </div>

#### 4. - Válasz a signin végpontra intézett post kérésre

Abban az esetben, hogyha egy kliens egy nem nyilvános végpontot próbálkozik meg elérni, a szolgáltatott tokent az applikáció authenticálja. Ezt a folyamatot az applikáció úgynevezett filterek másnéven szűrők segítségével hajtja végre. Ez ellenőrzi a token validitását.

A végpontokat különböző féle képpen lehet biztosítani. Elsősorban a Spring boot által biztosított globális biztonság beállításokkal. Itt egyetlen helyről be lehet állítani a különböző végpontok biztonsági státuszát. Másodsorban a végpontok biztonságát lokálisan, máshogy mondva *metódus szinten* is lehet konfigurálni. Én előszeretettel alkalmaztam ezt a módszert. Korábban a kontroller fejezetben már bemutattam, hogyan működik egy ilyen metódus szintű biztonság konfigurálása. Erre kiváló példa a `@PreAuthorize()` annotáció. Ennek egy felhasználását már bemutattam. Ezenkívül az általam készített `@IsAdmin` annotációt, melyet már korábban úgyszintén bemutattam, tulajdonképpen az előző annotációnak a felhasználásával készítettem.

Ezen kívül még fontos lenne megemlítenem az általam írt `OwnerChecker` osztályt is. Hiszen ennek az osztálynak a metódusait egyaránt használtam a `@PreAuthorize()` annotáció és a kontroller metódusok belsejében. Ezekkel a metódusokkal tudtam biztosítani azt, hogy a lekért erőforrások tulajdonosa ugyanaz legyen, mint a kéréssel beérkezett tokkenben található felhasználó.

## 4.2 A Frontend bemutatása

### 4.2.1 A Frontend Felépítése

Az angular keretrendszer moduláris dizájnon alapszik és építőköveik az úgynevezett angular komponensek. Minden egyes komponens egy-egy modulban található meg.

Az általam elképzelt applikáció több fontos, szinte központi részre osztható fel. Ezek sorban: *core*, *data*, *layout*, *shared*, *modules*.

A *core* részben találhatóak meg azok a elemei az applikációnak melyek központi szerepet játszanak. Itt található például az *AuthenticationService* osztály, amely a backend applikáció autentikációs végpontjával tartja a kapcsolatot. Itt található meg például még a *GlobalService* osztály is, amely segítségével lehet lekérdezni a jelenleg bejelentkezett felhasználó adatait, ez a böngésző úgynevezett *localStorage*-ával tartja a kapcsolatot. Itt található meg még a *JwtInterceptor* osztály is amely minden egyes kiküldött kéréshez hozzárendeli a felhasználó JWT tokenj-ét.

A *data* részben találhatóak meg az applikáció azon részei, amelyek a backend-el kommunikálnak. Ezen kívül itt találhatóak meg az adatmodellek. Az egyik itt megtalálható osztály a *UserDataService*. Ez a backend user végpontjaival kommunikál. Itt vannak definiálva a backend-ben is megtalálható adathordozó objektumok. Amint egy szervíz osztály egy http kérésre válaszul kap egy ilyen adathordozó objektumot, azt átkonvertálja egy hagyományos adatmodellre, ezután az applikáció ezt az adatmodellt fogja felhasználni. Egy példa egy ilyen adatmodell objektumra a *User* osztály lenne.

A *layout* Részben már angular komponensek találhatóak meg. Ezek a komponensek tartalmazzák az applikáció fontosabb vizuális elemei. Ezek határozzák meg, hogy a későbbiekben a többi komponens hol fog elhelyezkedni. Az itt található főkomponenseket akár sablonoknak is nevezhetnénk. Ezen kívül még itt találhatóak a header, illetve footer komponensek is. Ezek a különböző fejléceket, illetve lábléceket tartalmazzák.

Itt négy főkomponens található meg, ezek a *GuestLayoutComponent*, *AuthenticationLayoutComponent*, *UserLayoutComponent* és *AdminLayoutComponent*. Ezek megegyeznek az applikáció három felhasználói rétegével, illetve a bejelentkezés számára is van egy külön sablon.

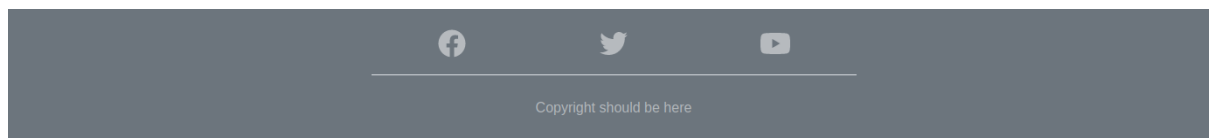
A shared részben azok az elemek találhatóak meg, amelyek tetszés szerint az egész applikációban használhatóak, viszont nem játszanak központi szerepet. Az általam elkészített applikációban csak két validator található meg itt, de akár teljes értékű komponenseket is tartalmazhatna. Például egy legördülő listát, vagy akár egy saját készítésű gombot. Olyan elemeket kellene tartalmazzon, amelyeket az applikáció fejlesztése során több helyen is fel szeretnénk majd használni.

A modul részt hagytam utoljára. Itt találhatóak azok a komponensek, amelyeket nem szeretnénk más modulokban is felhasználni, a shared részben található komponensekkel ellentétben. Az itt található modulok egy-egy komponens halmazzal rendelkeznek, illetve egy routing-al, amely megszabja a komponensek elérési útvonalát. Vannak modulok, amelyek csak egy darab komponenset tartalmaznak, mint például az AuthenticationLoginModule. De olyan modulokat is készítettem, amelyek sok-sok kisebb, illetve nagyobb komponenset tartalmaznak, mint például a UserPostsModule.

#### 4.2.2 Az autentikáció felületének bemutatása

Ebben a fejezetben be szeretném mutatni a frontend applikáció autentikációs felületét. Az autentikációs felületet az AuthenticationLayoutComponent nevű komponens köré építettem fel. Ezt a komponenset már az előző fejezetben is megemlítettem. Amint már korábban is mondtam, ezt a komponenset úgy lehet elképzelni, mint egy sablont. A komponens két fontos alkotóelemből áll. Ezek közül az egyik a footer, más néven a lábléc. A második alkotó elem pedig az autentikációs formulárt képezi.

A lábléc komponensen három ikont helyeztem el, illetve egy rövid szöveget. Abban az esetben, hogyha a kliens fel szeretné keresni az applikáció szociális médiában található oldalait, akkor ezekre az ikonokra kattintva az applikáció átirányítja a klienst az ikonnak megfelelő szociális média weboldalára. A rövid szöveg pedig a szinte már hagyományos módon kötelezővé vált *copyright* szöveget kellene tartalmazza. A jelenlegi szöveget csak mintaképpen alkalmaztam, ez csak egy helytartó szöveg.

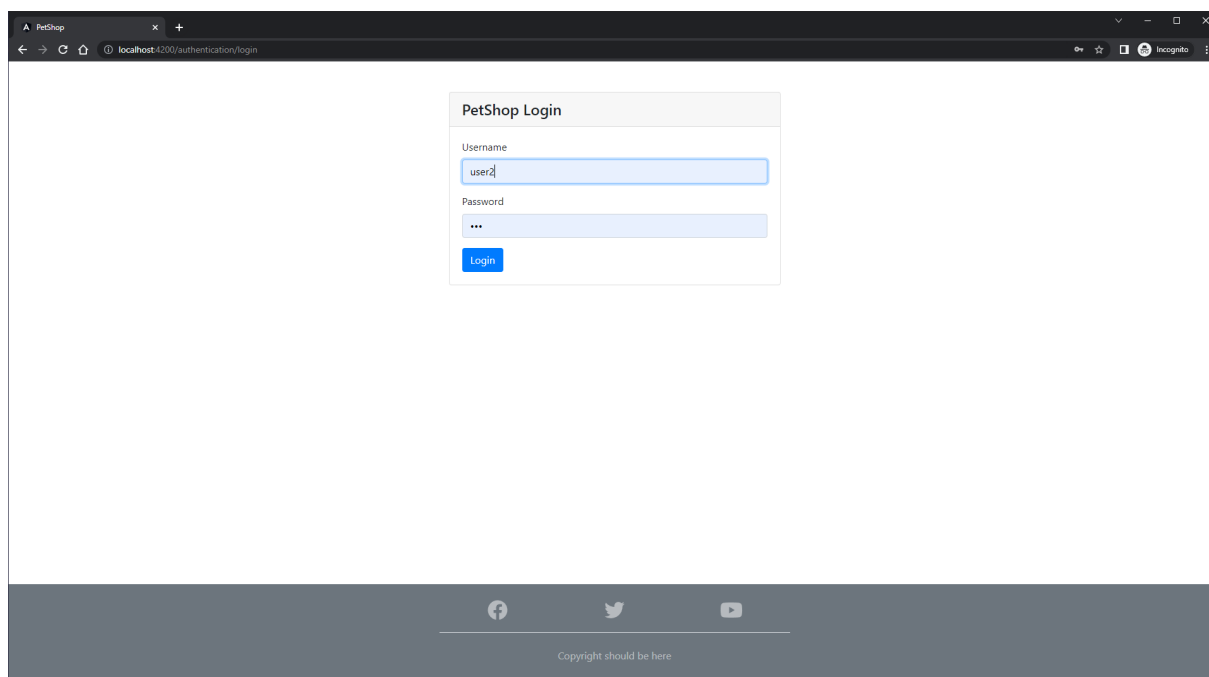


5. ábra - Lábléc

Abban az esetben, hogyha a későbbiekben az applikációt produkciós környezetben szeretném felhasználni, akkor a láblécben kellene pár módosítást végezni. Elsősorban a helytartó szöveget kellene lecserélni egy valódi copyright szövegre. Másodsorban a szociális média logókat ábrázoló ikonok is csak az adott szolgáltatások főoldalára irányítják át a klienst. Ezeknek az útvonalát is meg kellene változtatni, hogy az applikáció egy valós szociális médiában megtalálható oldalára irányítsa át a klienst, nem csak a főoldalra.

Az applikáció elkészítése során még több más helyen is alkalmaztam hasonló helytartó szövegeket. Abban az esetben, hogyha a publikum elé szeretném tárni az applikációt, ezeket a helytartó szövegeket mind le kellene cserélni egy-egy valós üzenetre.

Az `AuthenticationLayoutComponent` komponens második alkotóeleme az autentikációs űrlapot tartalmazza. Ennek az űrlapnak három nagyon fontos alkotóelemét szeretném bemutatni a következőkben. Ezek közül az első kettő egy-egy adatbeviteli mező, a harmadik pedig egy gomb. Az első mezőben a kliensnek a felhasználónevét kell megadnia, a másodikban a felhasználónévhez tartozó jelszót. A Login gomb segítségével pedig a kliens elindíthatja az autentikáció folyamatát.



*6. ábra - Az applikációs felület*

A kliens ezen az űrlapon keresztül tud belépni a rendszerbe. Ahhoz, hogy ezt sikeresen megtegye, a kliensnek ki kell töltenie az űrlapot a helyes adatokkal, ezután a login gombra

kattintva a frontend egy kérést intéz a backend-hez, amely ellenőrzi a megadott adatok helyességét. Abban az esetben, ha az adatok helyesek, akkor a backend egy jwt token-t küldd vissza válaszul a frontend-nek. A frontend ezt a token-t eltárolja az úgynevezett localStorage-ben, így a későbbiekben azt fel tudja használni.

Miután sikeresen lezajlott az autentikációs folyamat, az autentikációs felületről átirányítja az applikáció a klienst egy másik felületre. Attól függően történik meg ez az átirányítás, hogy a kliensnek milyen szerepkörei vannak. Egy user szerepkörrel rendelkező klienst a felhasználói felületre irányítja át az applikáció, míg egy adminisztrátori szerepkörrel rendelkező klienst az adminisztrációs felületre fogja átirányítani az applikáció.

### 4.2.3 A vendég felületének bemutatása

Ebben a fejezetben be szeretném mutatni a frontend applikáció vendég felületét. Ez a felület a GuestLayoutComponent nevű komponens köré építettem fel.

Ez a GuestLayoutComponent nevű komponens három fontos alkotóelemből állítottam össze. Ezek közül az első a footer komponens, más néven a lábléc. Ezt már az előző fejezetben bemutatam, hiszen ezt a komponens mind a négy felhasználói felületre elhelyeztem, így ezt itt nem fogom újra bemutatni. A második alkotó elem a header komponens vagy más néven a fejléc. Ebben helyeztem el a navigációs listát. A harmadik alkotó elem az úgynevezett Router Outlet. Ez az útválasztás eredményéül kapott komponens tartalmazza. Ez fogja tartalmazni a vendég felület központi tartalmát.



*7. ábra - A vendég felület fejléce*

A fejléc jobb oldalán négy menüpontot helyeztem el.

Az első a LogOut menüpont, avagy kijelentkezés. Abban az esetben, hogyha a kliens erre a menüpontra kattint a kurzorral, akkor az applikáció kijelentkezteti őt. Ez a menüpont csak a bejelentkezett kliensek számára látható.

A második a LogIn menüpont, avagy bejelentkezés. Abban az esetben, ha a kliens erre a menüpontra kattint a kurzorral, akkor az applikáció átirányítja őt az autentikációs felületre. Ez a menüpont csak azok számára látható, akik nincsenek bejelentkezve az applikációba.

A harmadik a Studio menüpont. Ennek segítségével tud a kliens átnavigálni a felhasználói felületre. Természetesen ezt a menüpontot is csak a bejelentkezett és user szerepkörrel rendelkező kliensek láthatják.

A negyedik az Admin menüpont. Ennek segítségével tud a kliens át navigálni az adminisztrációs felületre. Ezt a menüpontot csak a bejelentkezett és admin szerepkörrel rendelkező kliensek láthatják.

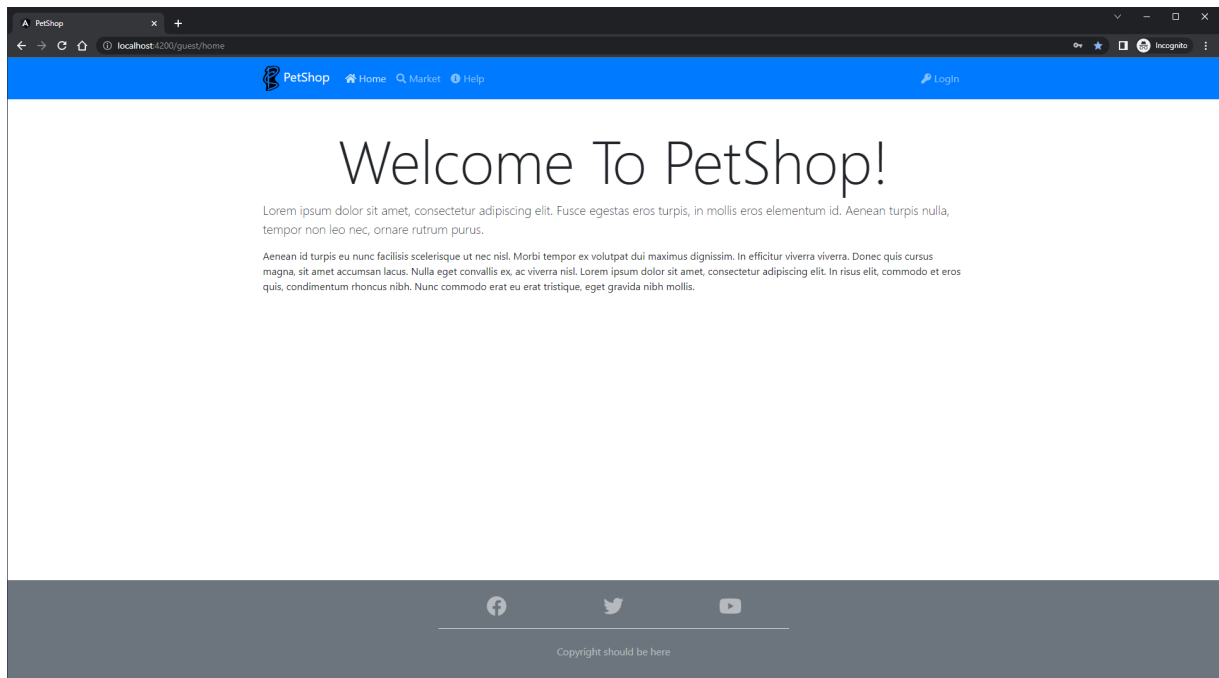
A fejléc bal oldalán ugyancsak négy menüpontot helyeztem el.

Az első ezek közül az applikáció logója, ezt nem csak esztétikai célból raktam ide. Amikor erre kattint a kliens, akkor az applikáció minden esetben a vendég felület főoldalára fogja navigálni őt. Ez azért fontos mivel ezt a gombot a következő két felületen is meg lehet majd találni, így mindhárom felületen lehetőséget adtam a kliensnek arra, hogy egyetlen kattintással visszatérjen a vendég felület főoldalára.

A logót három navigációs menüpont követi, ezeknek a nevei sorban: Home, Market és Help. Ezzel a három menüpontal tudja kiválasztani a kliens azt, hogy a vendég felületen mit is szeretne megtekinteni. Ezek a menüpontok befolyásolják, hogy a `GuestLayoutComponent` nevű sablon komponens `Router Outlet` nevű komponense mit is fog tartalmazni.

Az előzőleg említett Home menüpontra kattintva az applikáció a `guest/home` útra navigál. A Market menüpontra kattintva az applikáció a `guest/market` útra navigál és a Help menüpont pedig a `guest/help` útra történő navigálást teszi lehetővé. Ezeknek az útvonalaknak köszönhetően tudja a `Router Outlet` el dönteni, hogy milyen tartalmat jelenítsen meg.



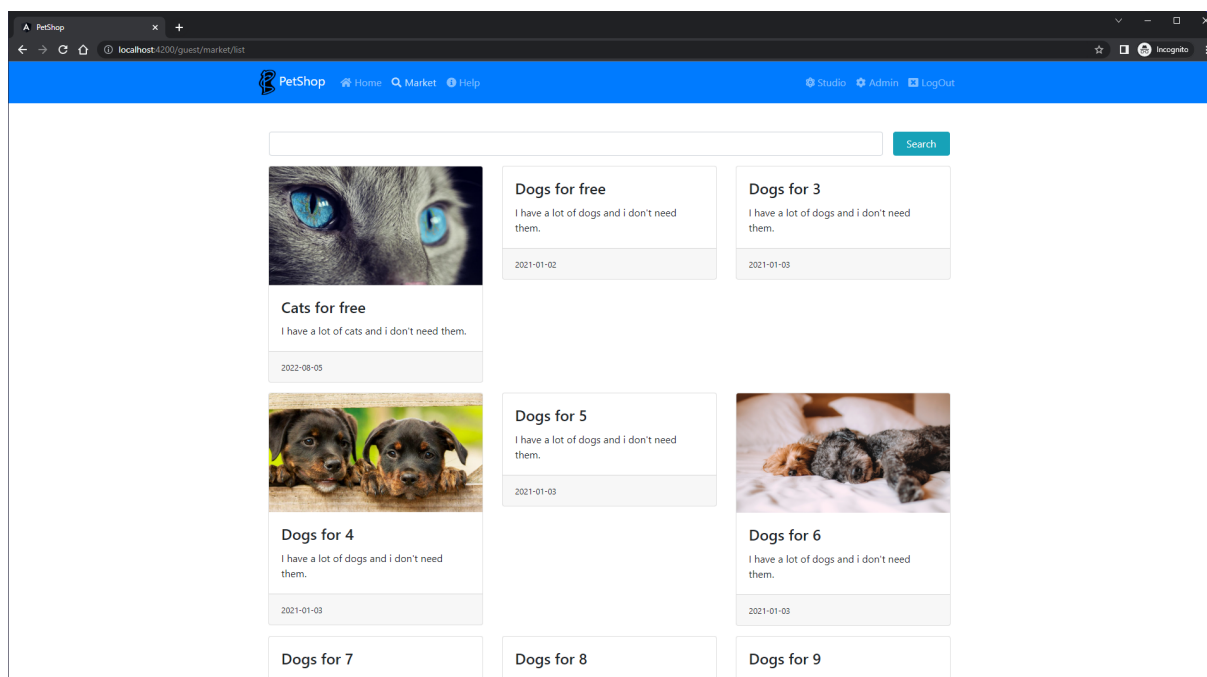


*8. ábra - A vendég felület home oldala*

A Router Outlet a home menüpont kiválasztása esetén betölti a home modult. Ebben a modulban csak egyetlen komponens található. Ide egy üdvözlő szöveget helyeztem el, illetve pár sor helytartó szöveget, ezeket be kell majd helyettesíteni egy valódi üdvözlő szöveggel abban az esetben, hogyha az applikációt valójában produkciós környezetbe szeretném hegyezni.

A Router Outlet a help menüpont kiválasztása esetén betölti a help modult. Ez hasonlóan egyszerű, mint a home modul. Ez is egyetlen komponenst tartalmaz, amelyen leginkább helytartó szövegeket helyeztem el.

A Router Outlet a market menüpont kiválasztása esetén betölti a market modult. Ez az előző két modulal ellentétben nem csak egy darab komponensből áll.



9. ábra - A vendég felület market oldala

A market oldalon három fontos vizuális elemet helyeztem el. Az első egy keresősáv. A második részben hirdetések találhatóak, avagy posztok. A harmadik elem pedig a hirdetések legalján található meg, ez a pagináció, vagyis a lapozás.

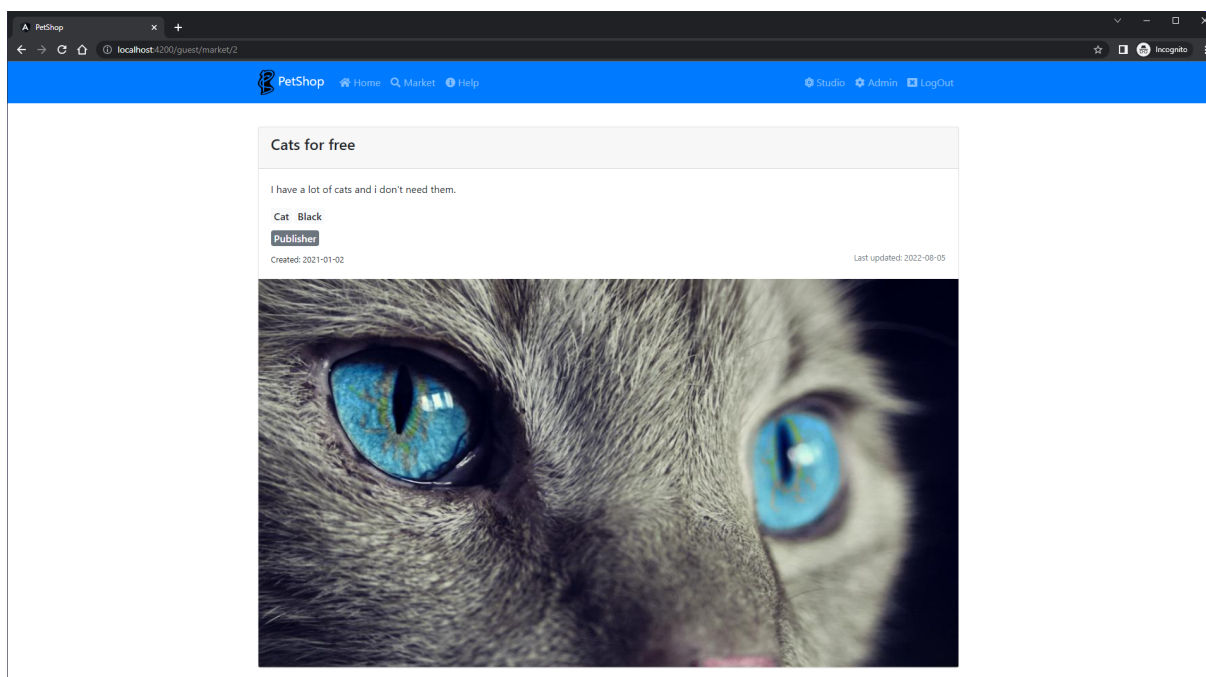
A hirdetéseket egy listába rendeztem el, ezek között lehet keresni. A listát nem teljes egészében jeleníti meg az applikáció, hanem feltördeli azt részlistákra, úgynevezett lapokra. Egyszerre csak egy lapot jelenít meg az applikáció.

A lapozás a szinte már hagyományos módon történik. Lehetséges az előre, a hátra lapozás. Ezen kívül arra is van lehetőség, hogy az előző, illetve következő pár lapra direkt át lehessen ugrani.

A keresősáv segítségével lehet a hirdetéseket megszürti. Minden hirdetés rendelkezik néhány címkével, ezek alapján lehet itt keresgélni a hirdetések közül.

A listában a hirdetéseknek feltüntettem a címét, a leírását, illetve az utolsó változtatás és a létrehozás dátumát. Ezen kívül, hogy ha rendelkezik a hirdetés egy képpel, akkor megjeleníti azt is applikáció.

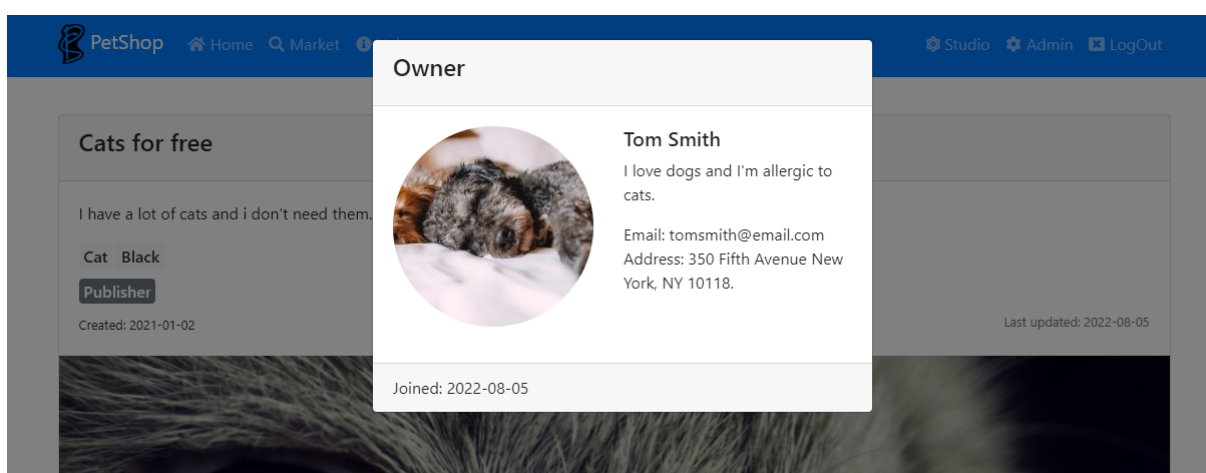
Hogyha a kliens meg szeretne nézni egy hirdetést közelebbről is, akkor rá kell kattintson a hirdetésre. Ez után az applikáció megnyitja a szóban forgó hirdetést.



10. ábra - Egy hirdetés a vendég felületen

Itt is megtalálható a hirdetés címe, leírása, illetve keletkezésének és utolsó változtatásának ideje. Ha a hirdetés tartalmaz képet, akkor azt is megjeleníti az applikáció.

Ezekon kívül még megtalálhatóak itt a hirdetés úgynevezett címkéi. Ezek alapján lehet keresni a hirdetések között. Ezek általában egy-két szót szoktak tartalmazni, mint például: kutya, macska, fekete, fehér, kis termetű, francia bulldog.



11. ábra - Egy hirdetés feladója a vendég felületen

Fontos még megemlíteni, hogy a kliens itt megtekintheti a hirdetés feladóját is. Ezt úgy teheti meg, hogy rákattint a Publisher gombra. Ez után felugrik egy ablak a hirdető adataival. Itt megtalálható a hirdető neve, elérhetőségei, profilképe, a dátum amikor először csatlakozott az applikációhoz, illetve egy rövid bemutatkozási szöveg.

#### 4.2.4 A felhasználó felületének bemutatása

Ebben a fejezetben be szeretném mutatni a frontend applikáció felhasználói felületét. Ezt a felületet a UserLayoutComponent nevű komponensre építettem fel.

Ez a UserLayoutComponent nevű komponens három fő alkotóelemből állítottam össze. Hasonlóan a GuestLayoutComponent komponenshez, itt is található egy fejléc, egy lábléc és egy harmadik komponens ami az útválasztás eredményeül kapott komponens tartalmazza, a Router Outlet.

A láblécet már a korábbiakban bemutattam az autentikációs felület bemutatása című fejezetben. Így itt nem ismétlem meg azt.

A fejléc hasonló a vendég felület fejlécéhez, de van benne pár fontos különbség. Ezek közül talán a leg szembetűnőbb az, hogy a fejléc színe nem kék hanem zöld. Az volt a célom ezzel a színbeli különbséggel, hogy a kliens már az első pillanattól kezdve érzékelje, hogy az applikációnak pontosan melyik felületén van jelenleg.



*12. ábra - A felhasználói felület fejléce*

A fejléc jobb oldalán két menüpontot helyeztem el. Az első a LogOut menüpont avagy kijelentkezés, ez ugyanúgy működik, mint a vendég felület fejlécében megtalálható LogOut menüpont. A második az Admin menüpont. Ez ugyancsak úgy működik, mint a vendég felület fejlécében megtalálható Admin menüpont.

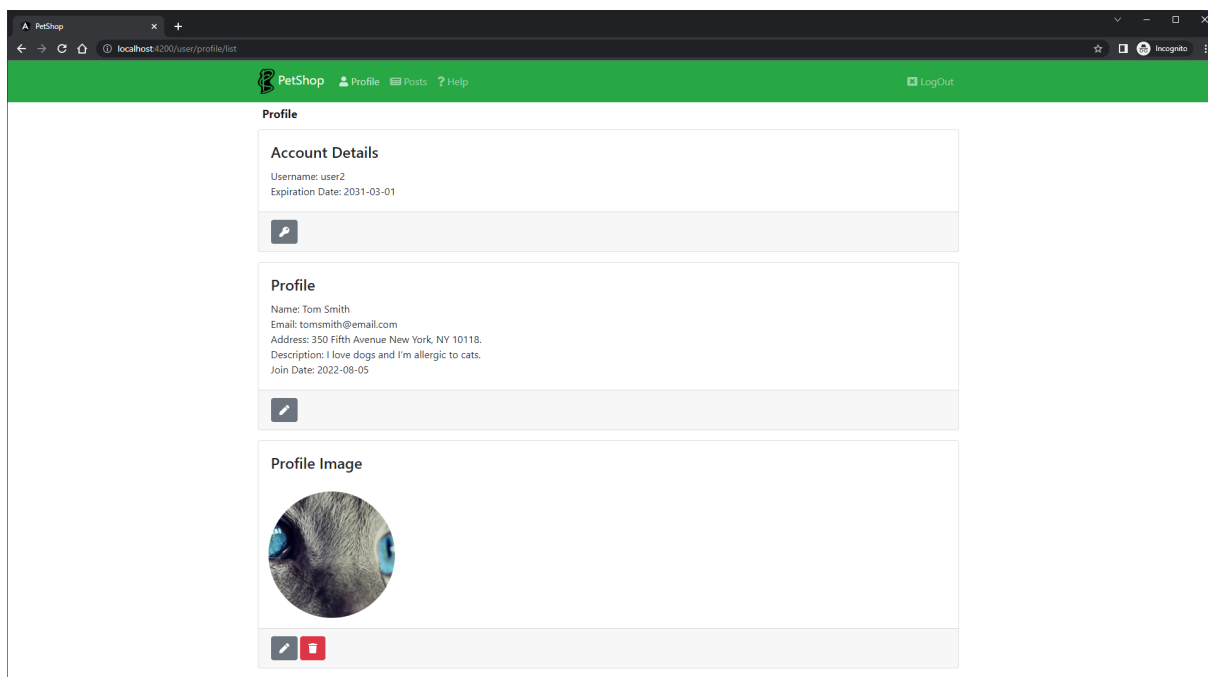
A fejléc bal oldalán négy menüpontot helyeztem el. Az első ezek közül az applikáció logója, ez ugyancsak úgy működik mint a vendég felület fejlécében megtalálható Logo. Amint már korábban is mondtam, ennek a menüpontnak a segítségével a felhasználó visszatérhet a vendég felület főoldalára. A logót három navigációs menüpont követi, ezeknek a nevei

sorban: Profile, Posts, Help. Ezzel a három menüponttal tudja a kliens befolyásolni, hogy a Router Outlet nevű alkotóelem mit is fog tartalmazni.

A Profile menüpontra kattintva az applikáció a user/profile útra navigál. A Posts menüpontra kattintva az applikáció a user/posts útra navigál. Amíg a Help menüpontra kattintva az applikáció a user/help útra navigál.

A Router Outlet a help menüpont kiválasztása esetén betölti a help modult. Ez hasonló a vendég felületen található help modulhoz, ez is egyetlen komponenst tartalmaz amelyben leginkább helytartó szöveget helyeztem el.

A Router Outlet a profile menüpont kiválasztása esetén betölti a profile modult. Ez már egy összetettebb modul, nem csak egyetlen egy komponensből áll.



*13. ábra - Egy profil a felhasználói felületen*

A profile oldalon lehetősége kínálkozik a kliensnek a saját felhasználói fiókjához tartozó adatokmenedzselésére. Három fontos alkotóeleme van ennek az oldalnak. Az első a felhasználói fiók adatait tartalmazza, a második a felhasználó profiljának adatait tartalmazza, míg a harmadik a felhasználó profiljának képét tartalmazza. Mindhárom alkotóelemet egy-egy úgynevezett kártyán helyeztem el.

Az első kártyán a kliens felhasználónevét helyeztem el és a felhasználói fiókjának lejáratí idejének dátumát. Abban az esetben, ha egy felhasználói fiók lejár, akkor a szóban

forgó fiókon végrehajtható műveletek korlátozva lesznek. Ezt a dátumot az adminisztrációs felületen meg lehet hosszabbítani. Ezen kívül a kártya láblécében elhelyeztem egy gombot, amelyre kattintva a kliensnek lehetőséget adtam arra, hogy megváltoztassa jelszavát.

A második kártyán helyeztem el a kliens nevét, elérhetőségeit, egy rövid leírást saját magáról és a dátumot amikor csatlakozott a rendszerhez. Ennek a kártyának a láblécében is elhelyeztem egy gombot. Erre a gombra kattintva lehetőséget adtam a kliensnek arra, hogy módosítsa az előbb felsorolt adatokat, a csatlakozási dátum kivételével. A későbbiekben a vendég felületen ezek az adatok megtekinthetők lesznek a kliens által közzétett hirdetéseken keresztül.

Az alábbi képen látható, hogyan is néz ki a felület, ahol az előzőleg felsorolt adatokat szerkeszteni lehet.

Profile / Edit Profile

**Profile**

Name

Email

Address

Description

I'm an admin.

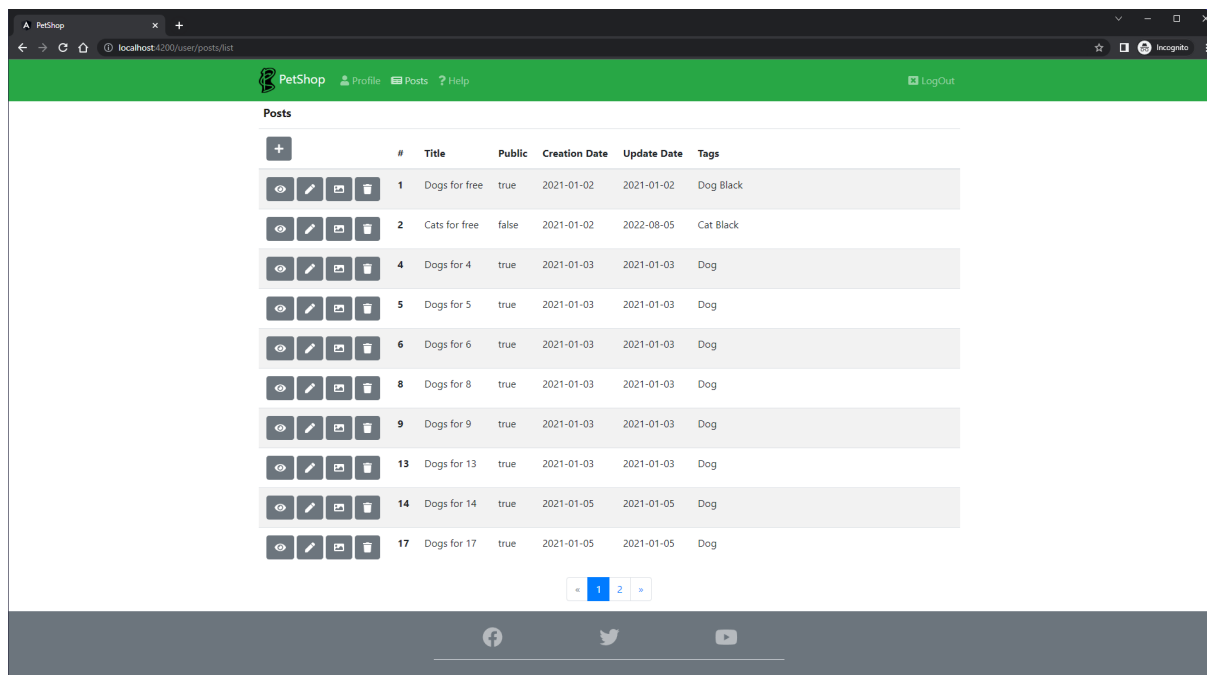
Save changes

Reset changes

*14. ábra - Egy profil szerkesztése a felhasználói felületen*

A harmadik kártyán helyeztem el a kliens profiljának képét. Ennek a kártyának a láblécében két gombot helyeztem el. Az első segítségével egy új profilképet tölthet fel a kliens, esetleg egy létezőt lecserélhet egy újra. A második gomb segítségével pedig eltávolíthatja az aktuális profilképét, így a vendég felületen megtalálható hirdetésein sem lehet majd a profilképét megtalálni.

A Router Outlet a posts menüpont kiválasztása esetén betölti a posts modult. Ez is egy összetettebb modul.

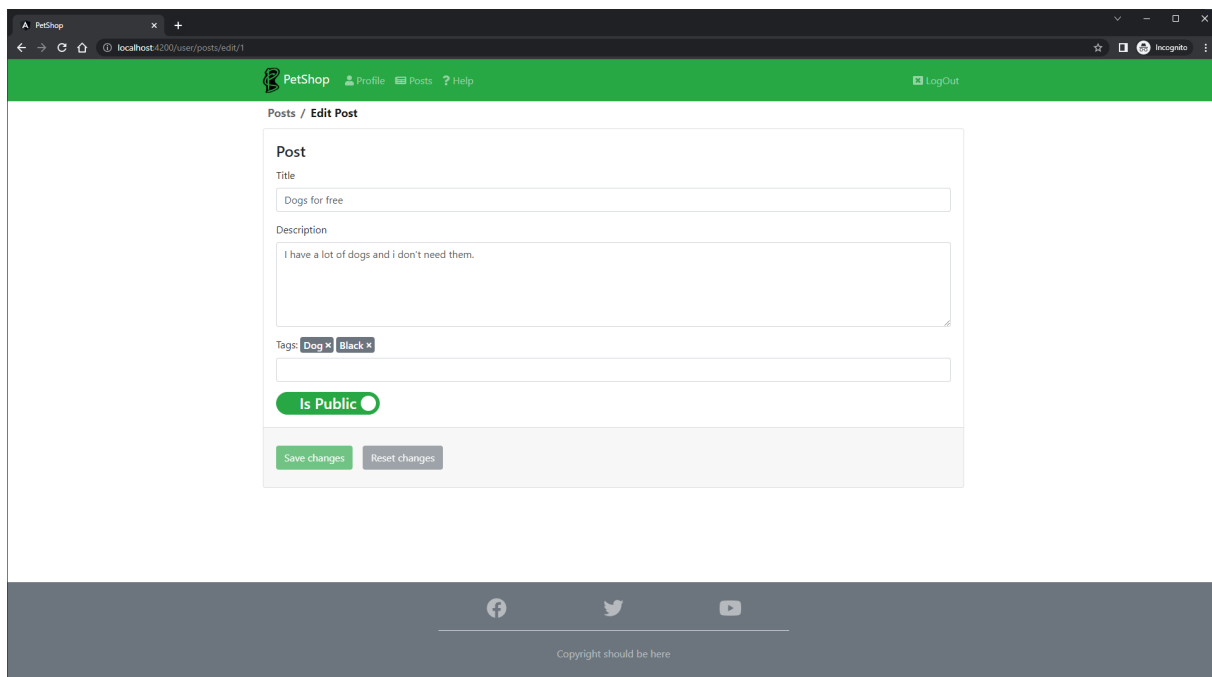


15. ábra - A posts oldal felhasználói felületen

A posts oldalon van lehetősége a kliensnek menedzselni a saját felhasználói fiókjához tartozó hirdetéseket. Itt egy táblázatot helyeztem el, amelyben a kliens keresgélhet a hirdetési között. A táblázat fejlécében elhelyeztem egy gombot, amely segítségével a kliens létrehozhat egy új hirdetést. A táblázat minden egyes sora egy-egy hirdetést reprezentál. Összesen hat oszlopa van ennek a táblázatnak és ebből öt sor a hozzá rendelt hirdetések adatait jelenít meg. A táblázat első oszlopában pedig gombokat helyeztem el.

A legelső gomb segítségével a kliens megnézheti, hogyan is mutatna a hirdetés a vendég felületen egy érdeklődő számára. A második gomb segítségével nyújtottam lehetőséget a hirdetés szerkesztésre. A harmadik gomb segítségével lehet manipulálni a hirdetéshez tartozó képet. A negyedik gomb segítségével pedig véglegesen el lehet távolítani egy hirdetést.

Amint már említettem, a hirdetés adatait szerkeszteni lehet. Az ehhez tartozó oldal kísértetiesen hasonlít ahhoz az oldalhoz, ahol egy új hirdetést lehet készíteni. Emiatt a következőkben csak a hirdetés szerkesztésénél látható oldalt szeretném bemutatni.



*16. ábra - Egy hirdetés szerkesztése a felhasználói felületen*

A hirdetés szerkesztése alatt egy formulár tárul a kliens szemei elé. Ez a formulár négy adat beviteli mezőből áll, illetve a láblécében még két gombot helyeztem el. Az első mezőben a hirdetés címe található meg, amíg a második mezőben a leírást lehet szerkeszteni. Ezek teljesen hagyományos szöveg beviteli mezők. A harmadik adat beviteli mezőben lehet megadni a hirdetéshez tartozó címkéket, ezek alapján lehet majd a vendég felületen célzottan keresgélni a hirdetések között. A negyedik adatbeviteli mező pedig egy úgy mondott kapcsoló. Ennek segítségével döntheti el a kliens, hogy a hirdetés publikus legyen vagy sem. Abban az esetben, hogyha a hirdetés nem publikus, akkor a vendég felületen nem fog az megjelenni. Az utolsó két elem pedig a két gomb, ezek közül az egyik segítségével el lehet menteni a végrehajtott változásokat, amíg a másik segítségével pedig vissza lehet állítani a formulárt az eredeti állapotába.

#### **4.2.5 Az adminisztrátor felületének bemutatása**

Ebben a fejezetben be szeretném mutatni a frontend applikáció Adminisztrátori felületét. Ezt a felületet a AdminLayoutComponent nevű komponensre építettem fel.

Hasonlóan a GuestLayoutComponent és a UserLayoutComponent komponensek felépítéséhez, az AdminLayoutComponent komponenst is három fő alkotórészből állítottam össze: egy fejlécből, egy láblécből és a RoutingOutlet nevű komponensből.



A lábléc teljes mértékben megegyezik a korábban már bemutatott lábléccel.

A fejléc is nagyon hasonlít a korábban bemutatott fejlécekre. A leginkább szembetűnő különbség a fejléc színe. Míg a vendég felület fejléce kék és a felhasználói felület fejléce zöld volt, az admin felület fejlécének színéül a szürkét választottam.



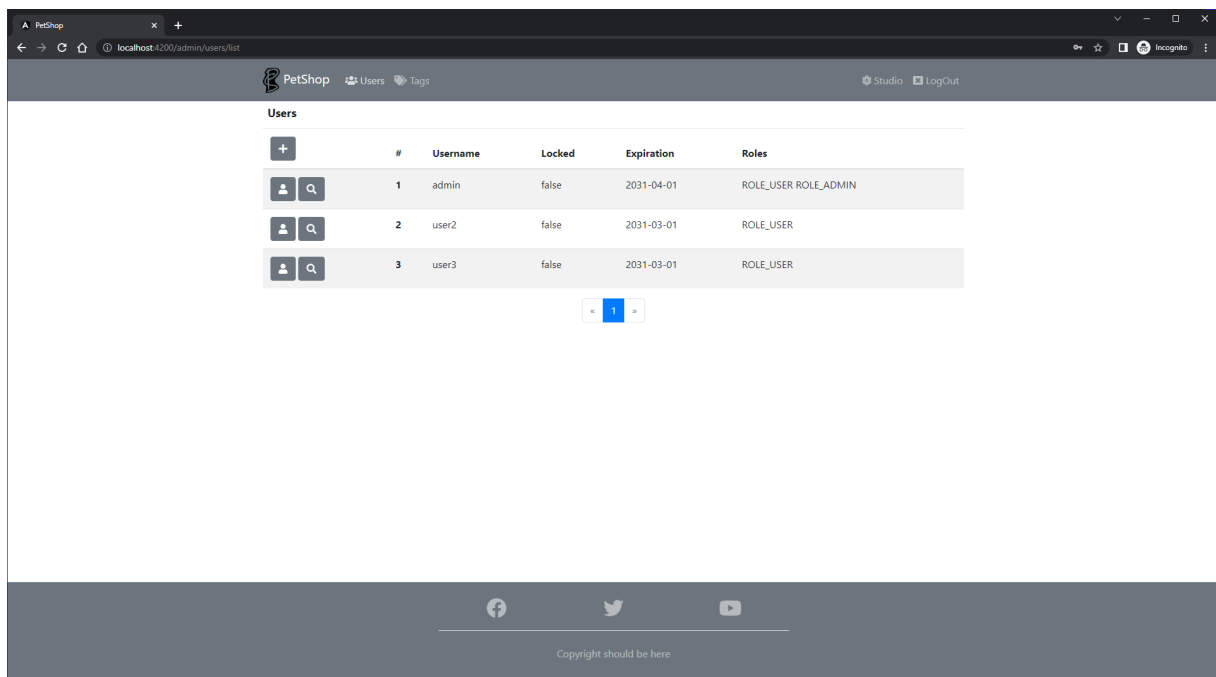
#### *17. ábra - Az adminisztrátori felület fejléce*

A fejléc jobb oldalán két menüpontot helyeztem el. Az első a LogOut menüpont, avagy kijelentkezés. Ez ugyan úgy működik mint a vendég, illetve a felhasználói felület fejlécében megtalálható LogOut menüpont. A második a Studio menüpont. Ez ugyancsak úgy működik mint a vendég felület fejlécében megtalálható Studio menüpont.

A fejléc bal oldalán három menüpontot helyeztem el. Az első ezek közül az applikáció logója, ez ugyan úgy működik mint a korábbi fejezetekben bemutatott logók. Ezt két navigációs menüpont követi, ezek a Users és a Tags menüpontok.

A Users menüpontra kattintva az applikáció a admin/users útra navigál. Amíg a Tags menüpontra kattintva az applikáció a admin/tags útra navigál.

Abban az esetben ha a felhasználó kiválsztja a Users menüpontot, akkor a Router Outlet betölti a users modult. Ha a felhasználó a Tags menüpontot választja ki, akkor a Router Outlet a tags modult tölti be.



18. ábra - A users oldal az adminisztrátori felületen

A users oldalon nyílik lehetősége az adminisztrátornak menedzselni a felhasználói fiókokat és azok hirdetéseit. Ide egy táblázatot helyeztem el, ennek a táblázatnak sorai egy-egy felhasználó adatait jelenítik meg.

Abban az esetben, hogyha az adminisztrátor az egyik felhasználó fiókjának adatait meg szeretné változtatni, ezt megteheti. Annyit kell tennie csupán, hogy a táblázatban a felhasználó adatait tartalmazó sor elején található ceruza ikonra kattint. Ez megnyitja a vendég felületen már bemutatott profile oldalt. A két oldal között kevés az eltérés. Az egyik különbség az, hogy míg egy adminisztrátor megváltoztathatja egy fiók lejáratát, ezt a felhasználói felületen nem lehet megtenni. A második fontos különbség az, hogy az adminisztrátor megváltoztathatja bármely fiók szerepköreit, másképpen mondva jogosultságait.

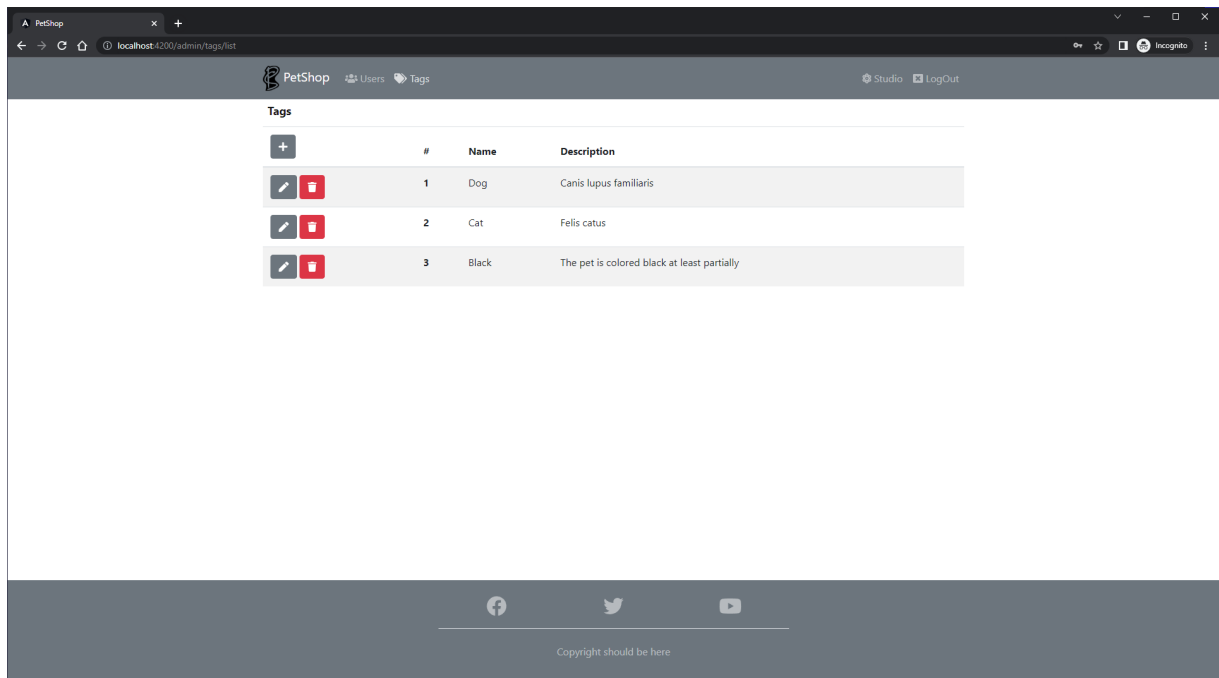
Abban az esetben, hogyha az adminisztrátor az egyik felhasználó hirdetéseiben szeretne változtatásokat tenni. Akkor azt megteheti a táblázatban a felhasználó adatait tartalmazó sor második gombjával. Az ezáltal megnyitott oldal kísértetiesen hasonlít a vendég felületen már bemutatott post oldalhoz. Annyiban van az eltérés a két oldal között, hogy az adminisztrátori felületen található táblázat nem tartalmazza a törlés gombot.

A táblázat címsorában elhelyeztem még egy gombot, erre a gombra kattintva adtam lehetőséget az adminisztrátornak arra, hogy egy újabb felhasználói fiókot hozzon létre. Ennek a felületét az alábbi képen lehet megtekinteni.

The screenshot shows the 'PetShop' admin interface. On the left, there's a sidebar with 'Users' and 'Tags' sections. The 'Users' section has a table with columns '#', 'Username', and 'Password'. The table contains three rows: 1 admin, 2 user2, and 3 user3. A modal window titled 'Account Details' is open in the center. It has fields for 'Username' (with placeholder 'username'), 'Password' (with placeholder 'password'), and 'Confirm Password' (with placeholder 'confirm password'). Below these is a 'Roles:' field and an 'Expiration Date' field (with value '2022-02-27'). There's also a toggle switch for 'Is Locked' and a green 'Save changes' button at the bottom right.

*19. ábra - Egy felhasználói fiók létrehozása az adminisztrátori felületen*

Abban az esetben, hogyha a kliens a Tags menüpontot választotta, a tags oldal tárul a szeme elé. A tags oldalon nyílik lehetősége az adminisztrátornak menedzselni a címkéket. Ezeket a hirdetésekhez lehet hozzákapcsolni és ezek alapján lehetséges a vendég felületen célzottan keresgélni a hirdetések között. Ide is egy táblázatot helyeztem el, ennek a táblázatnak sorai egy egy cimke adatait jelenítik meg.



*20. ábra - A tags oldal az adminisztrátori felületen*

Hasonlóan a többi táblázathoz, itt is megtalálható a fejlécben egy gomb. Erre a gombra kattintva nyílik lehetőség az adminisztrátornak arra, hogy egy új címkét tudjon létrehozni.

A lista soraiban található gombok segítségével pedig megváltoztathatja az adott sorban található címke adatait, illetve törölheti az adott címkét.

## 5 Összefoglaló

Szakedolgozatom célja az volt, hogy dokumentáljam egy modern webapplikáció elkészítését.

Először is kiválasztottam egy témát, vagyis egy problémát, amelyre egy webapplikáció megoldást tud kínálni. Ez a probléma nem más, mint a kisállattenyésztők és a házi kedvencekre vágyó emberek közötti kapcsolat megteremtése.

Másodsorban megterveztem az applikációt. Eldöntöttem, hogy a backendet Java nyelvben készítem el és azon belül pedig a Spring keretrendszert fogom alkalmazni, illetve azt is, hogy a frontendet az Angular keretrendszerben valósítom meg. Ezután az applikáció architektúráját szögeztem le és végül pedig megterveztem az adatbázis sémáját.

A következő lépés a megvalósítás volt. Létrehoztam a backendet és mikor az már használható állapotban volt elkezdtem a munkát a frontenden is.

Az applikáció fejlesztés során előszeretettel alkalmaztam a git verziókezelő rendszert. Abban az esetben, hogyha valamit nagyon elrontottam, ennek segítségével könnyedén vissza tudtam térni egy-egy korábbi verzióhoz.

A fejlesztés során sokszor történt meg, hogy egy-egy problémára a megoldást csak sokszori próbálkozás után találtam meg. Így minden egyes hiba után egy kicsikét gyarapodott a tudásom és a tapasztalatom a témakörben. A szakedolgozat témájának kiválasztása idején szinte semmit sem tudtam arról, hogy hogyan is lehet egy modern webapplikációt elkészíteni. Viszont a dolgozat befejezésével biztosan állíthatom, hogy elsajátítottam azt a szakmai háttér tudást, aminek a segítségével ismételten létre tudok hozni egy hasonló modern web applikációt, vagy akár egy mások által készített applikációt is tovább tudjak fejleszteni.

Úgy gondolom, hogy a dolgozatomban kitűzött célokat sikeresen teljesítettem. Hiszen végső soron sikerült egy jól strukturált és működő web applikációt létrehoznom.

Ugyan az elkészült applikáció nem került produkciós környezetbe, de ez nem is volt része a célkitűzésnek. Abban az esetben, hogyha az applikációt a publikum elé szeretném tární, meg kellene még ejtenem rajta néhány változtatást. Ezek közül talán a legfontosabb az

lenne, hogy az adatbázis kezelő rendszert le kellene cserélni egy robosztusabb, különálló rendszerre mint például a MySQL vagy a PostgreSQL.

Természetesen az applikáció közel sem tökéletes. Jelenlegi formájában eléggé limitált, így rengeteg féle képpen lehetne azt továbbfejleszteni.

Egy pár továbbfejlesztési ötlet:

- Egy üzenet küldő funkció, hogy a hirdető és a vásárló fel tudják venni a kapcsolatot az applikáción belül.
- Külön hirdetőfal elveszett házikedvencek számára.
- Kibővíteni a kereső funkciót hogy nem csak címkék hanem dátum, cím és leírás után is keressen.

Abban az esetben, hogyha a jövőben produkciós környezetbe kerül az applikáció, akkor remélem, hogy tényleg a rendeltetésének megfelelően lesz használva és hogy sok embernek lesz segítségére abban, hogy rátaláljanak a vágyott házi kedvencünkre.

## 6 Irodalomjegyzék

- [1] Git, [Online]. Available: <https://git-scm.com/doc>. [Hozzáférés dátuma: 26 szeptember 2022].
- [2] GitHub, [Online]. Available: <https://docs.github.com/en>. [Hozzáférés dátuma: 26 szeptember 2022].
- [3] IntelliJ IDEA, [Online]. Available: <https://www.jetbrains.com/help/idea/getting-started.html>. [Hozzáférés dátuma: 26 szeptember 2022].
- [4] Visual Studio Code, [Online]. Available: <https://code.visualstudio.com/docs>. [Hozzáférés dátuma: 26 szeptember 2022].
- [5] Java, [Online]. Available: <https://docs.oracle.com/en/java>. [Hozzáférés dátuma: 26 szeptember 2022].
- [6] Maven, [Online]. Available: <https://maven.apache.org/guides/index.html>. [Hozzáférés dátuma: 26 szeptember 2022].
- [7] Spring Boot Reference Documentation, [Online]. Available: <https://docs.spring.io/spring-boot/docs/2.5.4/reference/html/>. [Hozzáférés dátuma: 18 szeptember 2022].
- [8] H2, [Online]. Available: <https://www.h2database.com/html/main.html>. [Hozzáférés dátuma: 26 szeptember 2022].
- [9] Swagger UI, [Online]. Available: <https://swagger.io/docs/open-source-tools/swagger-ui/>. [Hozzáférés dátuma: 26 szeptember 2022].
- [10] HTML, [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTML>. [Hozzáférés dátuma: 26 szeptember 2022].

- [11] CSS, [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/CSS>.  
[Hozzáférés dátuma: 26 szeptember 2022].
- [12] TypeScript, [Online]. Available: <https://www.typescriptlang.org/docs/>. [Hozzáférés dátuma: 26 szeptember 2022].
- [13] Node.js, [Online]. Available: <https://nodejs.org/en/docs/>. [Hozzáférés dátuma: 26 szeptember 2022].
- [14] Angular, [Online]. Available: <https://angular.io/docs>. [Hozzáférés dátuma: 14 szeptember 2022].
- [15] Bootstrap, [Online]. Available: <https://getbootstrap.com/docs/4.6/>. [Hozzáférés dátuma: 14 szeptember 2022].
- [16] Angular powered Bootstrap, [Online]. Available:  
<https://ng-bootstrap.github.io/releases/10.x/>. [Hozzáférés dátuma: 14 szeptember 2022].
- [17] Font Awesome, [Online]. Available: <https://fontawesome.com/v4>. [Hozzáférés dátuma: 26 szeptember 2022].
- [18] Stackoverflow, John Au-Yeung és Ryan Donovan, „Best practices for REST API design” [Online]. Available:  
<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/>.  
[Hozzáférés dátuma: 14 szeptember 2022].
- [19] OWASP, „REST Security Cheat Sheet” [Online]. Available:  
[https://cheatsheetseries.owasp.org/cheatsheets/REST\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html).  
[Hozzáférés dátuma: 14 szeptember 2022].
- [20] Baeldung, „Entity To DTO Conversion for a Spring REST API” [Online].  
Available:  
<https://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application>.  
[Hozzáférés dátuma: 20 szeptember 2022].