**Vizsgaremek**

***AUMS***

**Access and User Management System**

Magyar Máté

Táborosi István Balázs

Lator Bence

2023.

Szegedi Vasvári Pál technikum

**Tartalomjegyzék**

[A szoftver célja 3](#_Toc134391223)

[Szerver 3](#_Toc134391224)

[Domain (Cloudflare) 7](#_Toc134391225)

[GitHub (Action) 8](#_Toc134391226)

[Hardware 9](#_Toc134391227)

[Adatbázis 10](#_Toc134391228)

[Általános leírás 10](#_Toc134391229)

[User tábla 11](#_Toc134391230)

[Schedule tábla 11](#_Toc134391231)

[Card tábla 11](#_Toc134391232)

[User\_card tábla 11](#_Toc134391233)

[Role tábla 12](#_Toc134391234)

[User\_role tábla 12](#_Toc134391235)

[Unknown\_card tábla 12](#_Toc134391236)

[EK diagram 12](#_Toc134391237)

[Backend 13](#_Toc134391238)

[Általános leírás 13](#_Toc134391239)

[Modellek 13](#_Toc134391240)

[Model Inicializálás 13](#_Toc134391241)

[Model 14](#_Toc134391242)

[Kontrollerek 14](#_Toc134391243)

[Szolgáltatások 14](#_Toc134391244)

[Segédfüggvények 15](#_Toc134391245)

[Konfiguráció, környezeti változók és futtatás 15](#_Toc134391246)

[Tesztek 16](#_Toc134391247)

[Tesztek inicializálása 17](#_Toc134391248)

[Beállítások és problémák kiküszöbölése 17](#_Toc134391249)

[Tesztelés 18](#_Toc134391250)

[Frontend 19](#_Toc134391251)

[Alkalmazott szoftver eszközök 19](#_Toc134391252)

[Nyelvhasználat 19](#_Toc134391253)

[Megvalósítást segítő keretrendszerek 19](#_Toc134391254)

[Bejelentkező felület 20](#_Toc134391255)

[Home 20](#_Toc134391256)

[Register 21](#_Toc134391257)

[Cards 21](#_Toc134391258)

[A folyamat az alábbi módon zajlik: 21](#_Toc134391259)

[Schedule 22](#_Toc134391260)

[User Management 22](#_Toc134391261)

[Log 23](#_Toc134391262)

[Desktop 23](#_Toc134391263)

[Felhasznált források 24](#_Toc134391264)

# A szoftver célja

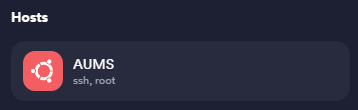
A kártyás beléptető rendszer célja, hogy lehetővé tegye a szervezetek számára, hogy hatékonyan kezeljék a dolgozók belépését és kilépését munkahelyi területekre.

A rendszer tartalmaz egy beléptető felületet, ahol minden felhasználónak lehetősége van belépni a saját fiókjába és ellenőrizheti a kártyájával való belépésének és kilépésének időpontjait, saját adatait. Két adminisztrációs felületet, amelyet csak admin jogosultsággal rendelkező felhasználók érhetnek el, ami lehetővé teszi a kártyák és a hozzájuk tartozó személyek kezelését. Egy regisztrációs ablakot, ahol egy menedzser vagy admin rangú személy létre tud hozni új felhasználót. Valamint egy beléptető terminált, amely lehetővé teszi a dolgozók számára a munkahelyi területekre történő belépést és kilépést.

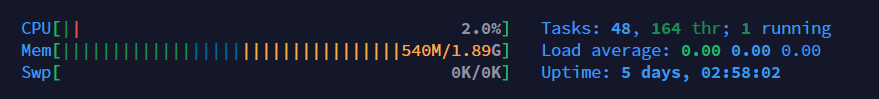
A rendszer célja, hogy biztonságosan és hatékonyan kezelje a munkahelyi belépések és kilépések folyamatát, és így megóvja a szervezet értékeinek és információinak biztonságát a jogosulatlan belépéstől és az esetleges biztonsági fenyegetésektől.

# Szerver

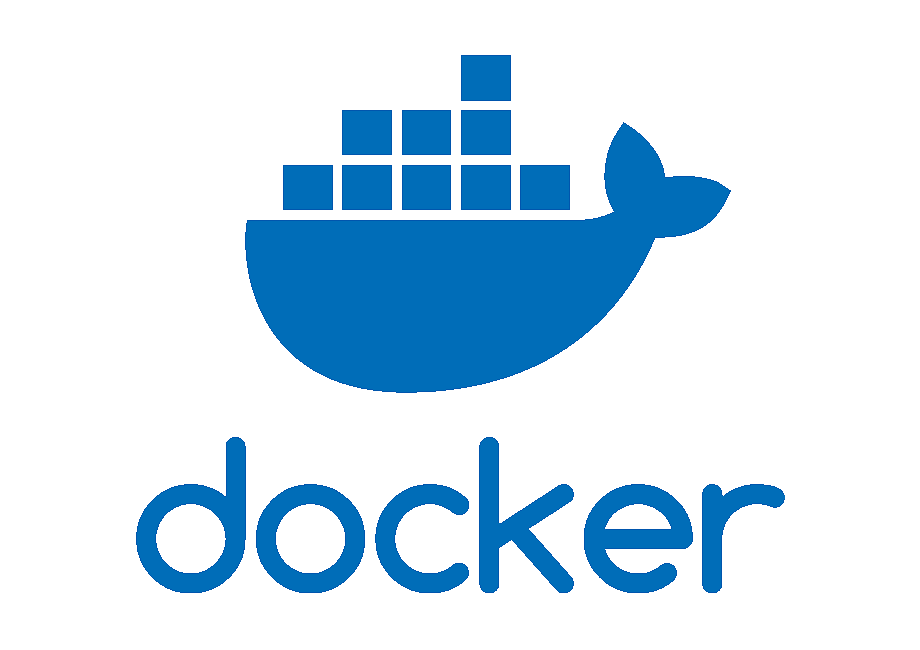
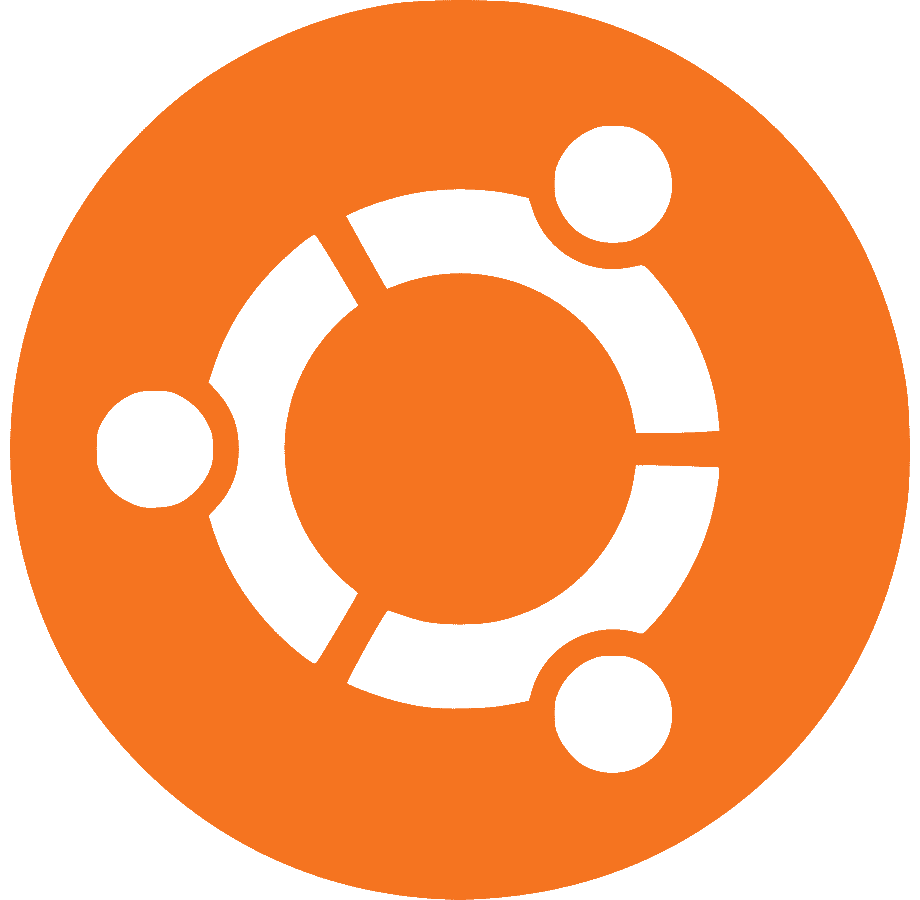
Az éles rendszer megfelelő működéséhez egy fizikai szerveren futtatjuk az állományokat. Ennek megvalósításához több felvezető lépést alakítottunk. A cél az volt, hogy egy 0-24 elérhető szolgáltatást nyújtson a projektünk, így a személyes otthoni futtatás nem volt opció számunkra. Természetesen teszt környezet továbbra is futtatható a helyi gépeken, viszont az éles termék egy távoli elérésű eszközön fut.



Ezen hardware eszközt egy külső cég szolgáltatja (hetzner). A specifikációt tekintve egy 2 magos AMD processzor, 2 GB memória és 40 GB SSD tárhely áll rendelkezésre. Az elérése pedig SSH (secure shell) kapcsolaton keresztül történik a megfelelő felhasználónév-jelszó páros megadásával. A hardware kiválasztása után, szerver szempontból a következő nehéz terület a rajta felhasznált rendszer kérdése volt, hiszen több szempontnak is tökéletesen meg kellett felelnie, ilyen például az alacsony sebezhetőség, jól skálázhatóság vagy könnyű kezelhetőség.

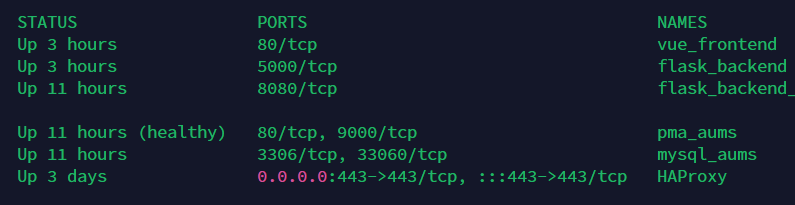


Elsőként az operációs rendszer kiválasztása történt meg, úgy gondoltuk az ubuntu 20.04 server (CLI) változata tökéletes megoldás lehet számunkra hiszen már sok hozzá tartozó lehetőséget ismertünk és a felhasználható opciók tárháza is igen nagy. A rendszer működőképes állapotba hozatala után (telepítés, frissítés) a fő technológia választás következett. Itt a docker mellett tettük le a voksunk.



A docker lényegében elképzelhető úgy, mint egy, a fő eszközön folyamatosan futtatott rendszer, amely lehetővé teszi azt, hogy különböző szolgáltatások egymástól függetlenül működjenek elkülönített “konténerekben”. Jelen esetben ez azt jelenti, hogy például a frontend és a backend alkalmazásunk is egy bizonyos docker container-en belül fut, aminek egymástól függetlenül megvan a saját tárhelye, beállításai is környezete. Ez továbbá lehetőséget ad arra, hogy ha egy újraindítás szükséges a backend-re akkor az külön megtehető anélkül, hogy a teljes rendszer leállna. Továbbá sok hasznos lehetőséget nyújt számunkra, hiszen egyszerűen rendszerezve elérhetőek a futtatott részek, és a hozzá tartozó alap és beállítások is könnyen cserélhetőek.

A futtatáshoz a docker compose opcióját használjuk, amely megmondja, hogy egy bizonyos struktúrájú szöveges fájl (yml) megadásával meghatározhatóak a futtatandó szolgáltatás specifikációi. És könnyedén indíthatóak, leállíthatóak, újraindíthatóak vagy törölhetőek/módosíthatóak ezzel (docker-compose up, down, restart, ps, rm stb).

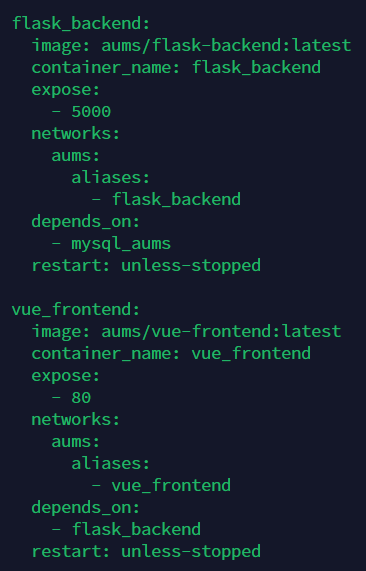


A használt compose file felépítése:

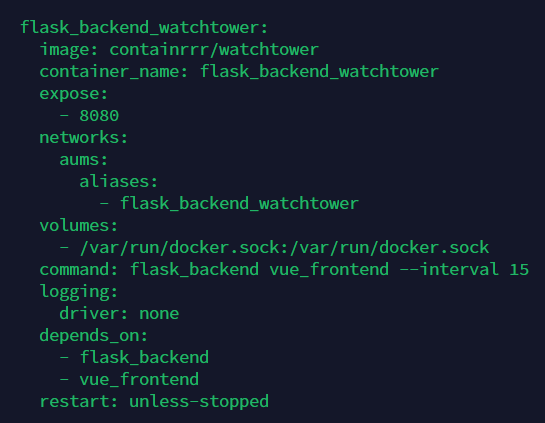
* version: A compose fájl verziója
  + Esetünkben ez mindenhol “3.5”
* services: A szolgáltatások ezen belül helyezkednek
  + A projekten futtatott szolgáltatások:
    - MySQL
    - HaProxy
    - Watchtower
    - PhpMyAdmin
    - A saját backend szolgáltatásunk
    - A saját frontend szolgáltatásunk
* image: A futtatandó szolgáltatás “képfájlja” (lásd lentebb)
  + Röviden maga a teljes program egy fájlba tömörítve
* environment: A futtatási környezeti változók meghatározása
* volumes: Az adott konténer tárhely meghatározása, ha szükséges
* container\_name: A docker konténer neve, a jól felismerhetőség érdekében
* expose/ports: A felhasznált portok megadása
  + Ports megadásával nyitott portként fut tovább
  + Expose esetében a felhasznált port csak a konténeren belül érhető el
* networks: Belső hálózat meghatározása
  + A biztonság érdekében a szolgáltatások az “aums” belső hálózaton futnak
* restart: Újraindítási vagy hiba esetén felmerülő indítási szekvencia meghatározása

A compose fájl további beállításokat is tartalmaz, ami nem került említésre, ilyen például A “depends\_on” amely kapcsolatot biztosít két külön konténerben futó szolgáltatás között, az egyik csak úgy működőképes ha már a másik elindult előtte. Illetve egy másik szolgáltatás is futtatás alá került saját compose file felhasználásával (haproxy), ezekre később esik említés.

A compose fájlban megadott image lényegében egy olyan fájl ami magába foglalja a teljes projektet. Egy ilyen fájl build segítségével hozható létre, ilyet mint a frontend, mint a backend szolgáltatásunk számára elkészítettünk amely segítségére egy Dockerfile került felhasználásra. Ebben lényegében meghatározható minden, amely az adott rész telepítéséhez és futtatásához szükséges lehet. Mint például használatos technológia (js, node, python), a munkakönyvtár vagy akár a program telepítendő szükségletei. Ezeket megadva a Dockerfile buildelhető állapotba kerül, amely végül létrehozza a docker image fájlt amely magába tömörítve tartalmazza a szolgáltatást és a hozzá tartozó szükséges telepítendő és felhasználandó körülményeket. Az image-et felhasználva a compose fájl egy sorát lecserélve akár több verziót is kipróbálhatunk az adott szolgáltatásból, és egyszerű csere/bővítési és módosítási lehetőséget biztosít. A compose környezeti változó opciója pedig hasznos lehet részletesebb beállítások megadására, ilyen például a mysql esetében a root felhasználó neve, vagy a phpmyadmin esetében a host adatai, a mi esetünkben pl. a host adat a fentebb létrehozott mysql szolgáltatás, és a service név megadásával egyből tudja, hogy a szerveren található mysql konténer szükséges a megfelelő működéséhez.



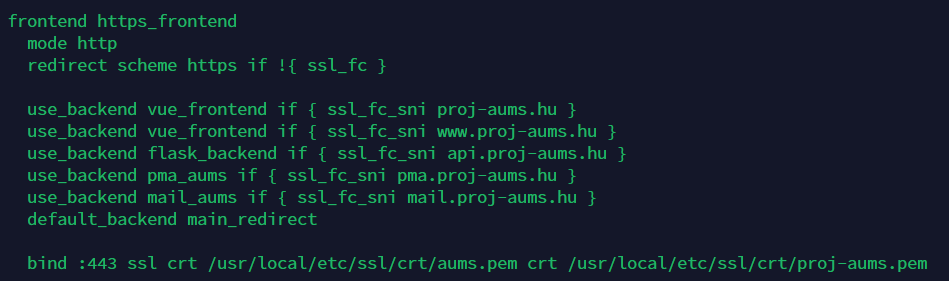
A felépített szolgáltatásokok közé tartozik a watchtower is, amely hasznos eleme az automatizálásának. Úgy gondoltuk, hogy a szerverre kézzel való felmásolás már elavult folyamat, így a GitHub action segítségével (lásd GitHub pont) az újonnan létrehozott verziókat a szerveren található watchtower folyamatosan figyeli, és amennyiben talál egy frissebb verziót egyből letölti azt és felülírja a szerveren helyileg futó verziót. A docker hubot választottuk a frissítések tárhelyének, erről a GitHub pontban részletesebb leírás található.



A minél nagyobb biztonság érdekében egyetlen portot sem hagytunk nyitottan, mivel meghatároztuk, hogy minden szerveren futó éles szolgáltatásunk egy saját “aums” hálózaton belül legyen, így csak ezen a hálózaton belül érhetik el egymást. Természetesen szükségesek voltak biztonságos külső elérési pontok kialakítása is, hiszen valahogyan el kell érni a frontend vagy phpmyadmin felületet. Erre egy specifikusan beállított proxy került felhasználásra.

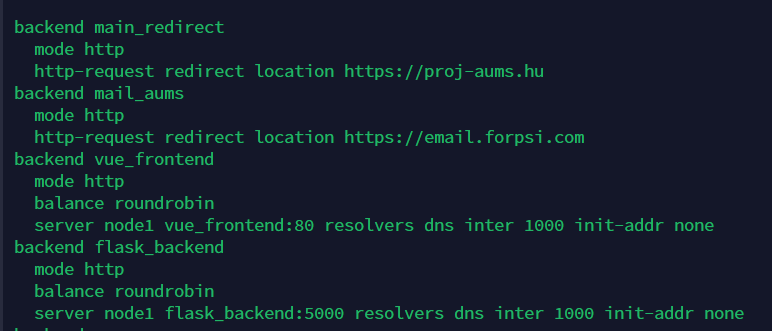


A felhasznált proxy (HaProxy) feladata, hogy a kommunikáljon a domain-re érkezett kérdésekkel és a megfelelő szerveren található szolgáltatásra irányítsa ezeket. Fontos szerepet vállal ebben a Cloudflare is, ami lentebb szintén bemutatásra kerül. A proxy egy config fájlal rendelkezik, amelyben megadható az alap frontend kezelési platform. Ebben meghatározhatóak, hogy bizonyos kérések esetén milyen irányba továbbítsa ezt. Először is egy, a cloudflare által generált tanúsítványt használ, amely lehetővé teszi, hogy biztonságos (https) kapcsolat kerüljön létrehozásra és felhasználásra a szükséges platformok esetében.



Példa a kérés fogadásra, és továbbítására: use\_backend vue\_frontend if { ssl\_fc\_sni proj-aums.hu }. A példában látható, hogy amennyiben kérés érkezne a proj-aums.hu domain-re, akkor irányítsa azt át a vue\_frontend részre, amely az alábbi módon kerül megadásra:

backend vue\_frontend  
 mode http  
 balance roundrobin  
 server node1 vue\_frontend:80 resolvers dns inter 1000 init-addr none



Az vue\_frontendre irányítás megadásakor meghívja a 80-as porton futó szolgáltatást, amely a docker-compose fájlban vue\_frontend néven került meghatározásra. Szóval összesítésben, egy bizonyos kérés érkezésekor a proxy megkeresi a kéréshez társított szolgáltatást és azt adja vissza a felhasználó számára. Néhány esetben a kérés ismeretlen lehet, erre egy alapértelmezett beállítás került meghatározásra, amely az ismeretlen kéréseket különösebb tevékenység elvégzése nélkül csak a főoldalra irányítja.



Továbbá a proxy egy fontos eleme, hogy egy további biztonsági lépés ékelhető a szolgáltatás és a felhasználó közé, jelen esetben az úgy történik, hogy például, ha egy felhasználó megpróbálja betölteni az éles rendszer phpmyadmin felületét (pma.proj-aums.hu) akkor egy felugró ablak jelenik meg felhasználónév-jelszó párost várva, amelynél feltétlen szükséges a megfelelő adatok megadása vagy a kérés elutasításra kerül. Ezen beállítás is a haproxy konfiguráció hasznos része további egyéb, kisebb lépésekkel együtt.

# Domain (Cloudflare)

Már a projekt előkészítésénél tudtuk, hogy a nyilvános elérés legjobb módszere egy publikus domain elérhetőségével lenne teljes. Ezért megvásároltuk a “proj-aums.hu” címet, amely megadásával jelenleg a frontend szolgáltatás betöltése történik (és a további aldomain részeken az egyéb szolgáltatásaink). Bizonyos szempontból egy domain üzemeltetési folyamata fontos része a teljes képnek, illetve precíz beállítás és konfiguráció szükséges a megfelelő működés elérése érdekében.



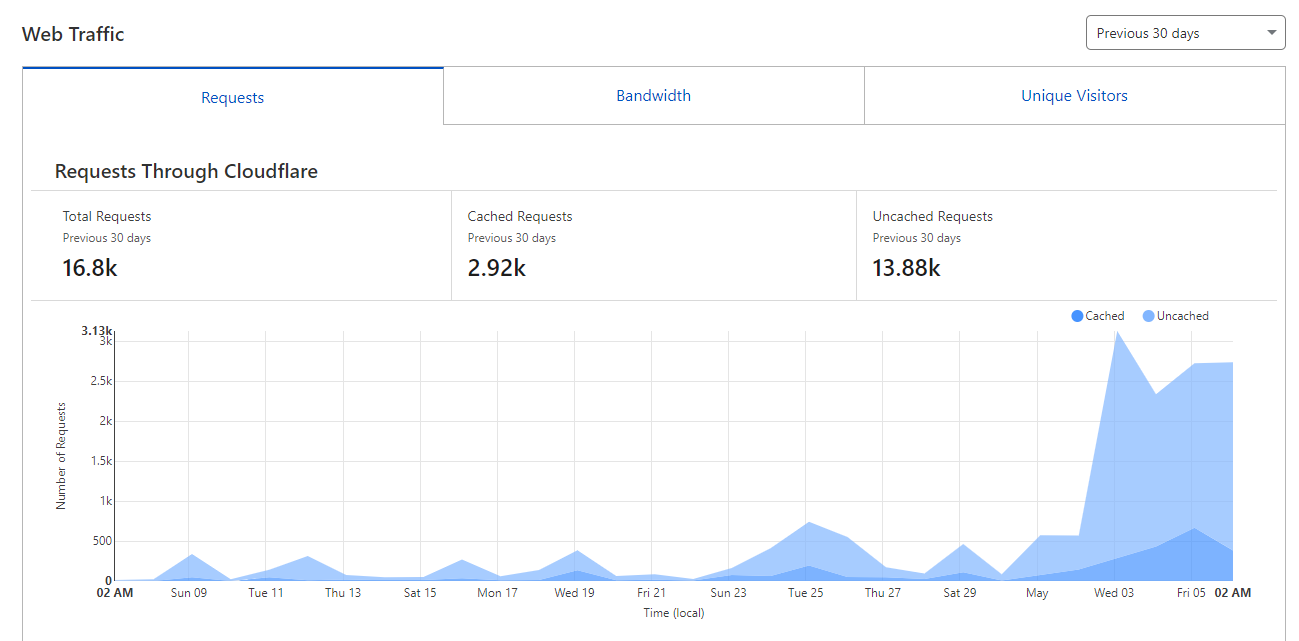
A projekt részeként úgy döntöttünk, hogy felhasználjuk a Cloudflare által nyújtott szolgáltatásokat, a cloudflare lényegében a saját szolgáltatását nyújtva lehetőséget ad arra, hogy a megvásárolt címet rajtuk keresztül irányítsuk. Természetesen rengeteg védelmi és optimalizációs lehetőség válik elérhetővé a hozzáadás után, ezért döntöttünk mi is mellette.

A hozzáadás két rövid lépésben zajlott, először is a regisztrátor oldalán megváltoztatásra került a domainhez csatolt alap dns szerver a cloudflare dns-re. Majd egy rövid ellenőrzés után már használhatóvá vált minden lehetőség.



A főoldalon kiemelendő beállítási lehetőség a támadás alatti mód, amely támadás esetén bekapcsolható (automatikus védelem és védelmi szint is állítható), a teljes körű támadási védelem még a domain betöltése előtt egy bot megfigyelésre és captcha ellenőrzésre kényszeríti a felhasználót, ezzel is csökkentve a szerverre érkező terhet. A fejlesztői mód pedig kikapcsolja a cachelést, ezzel biztosítva, hogy minden beállítás és új megjelenés/funkció a feltöltés után egyből megjelenjen az oldalon.

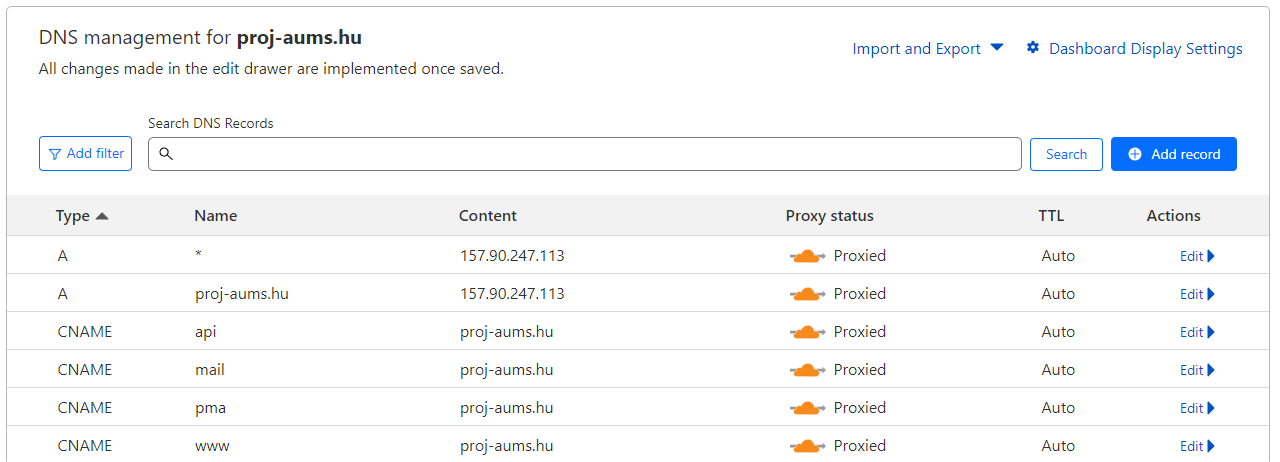
Az analitika és jelentések fül részletes információt biztosít az oldal forgalmáról, mint például az összes lekérdezés száma, forgalom nagysága és egyedi látogatók száma, illetve országok szerinti csoportosítása. Ezen a fülön továbbá minden statisztika napi/heti/havi felbontásra váltható. Valamint tartalmaz teljesítmény mérési részleteket és támadás védelmi kimutatást is.



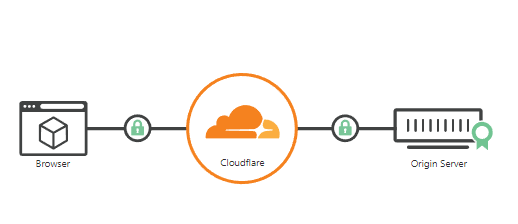


Az oldalon megtalálható egyik, ha nem a legfontosabb menüpont a DNS rekordok kezelése. Itt kapcsolható össze a domain a szerverrel. Egy A típusú rekord létrehozása volt szükséges, amelynél a name a “proj-aums.hu” címre, a content pedig a szerver IP címére mutat.

A cloudflare és a proxy közös munkájával a lehető legnagyobb biztonságot sikerült elérnünk, hiszen visszafejtés esetén is az esetleges támadó csak a cloudflare publikus IP címéhez jut el, míg a szerver címe mindvégig titkos marad. A DNS beállításoknál hoztuk létre a szükséges al-címeket, ilyen például az api.proj-aums.hu amely CNAME rekordként került létrehozásra és erre a címre való kérés érkezésekor a cloudflare továbbítja a proxy felé a kérést, a proxy pedig felismerve ezt feldolgozza, továbbítja és visszaadja a backend API bizonyos lekérdezését és válaszát.



Az SSL/TLS menüpont alatt is hasznos beállítások kerültek megadásra, ilyen a full-strict mód, amely megadja, hogy teljeskörű titkosítás megy végbe a kliens és a szerver között a cloudflare beágyazással. Itt került generálásra az SSL tanúsítvány is amely a biztonságos kapcsolatot biztosítja. Továbbá több kisebb beállítás is aktív, mint például az automatikus https átirányítás (http esetén a https verzióra irányítja a felhasználót) vagy a minimum TLS verzió megadása.



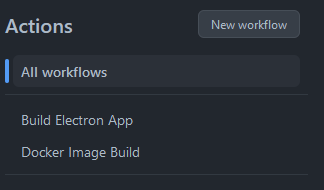
A biztonsági fülön létrehozásra kerültek alapvető szabályok, amelyek a bot, DDoS támadások elleni védelmet erősíti, valamint itt állítható az alapértelmezett védelmi szint is.

Továbbá sok hasznos beállítást is alkalmaztunk több menüpontból, ilyen például az always online, amely esetleges hiba felmerülésénél, ha a szerveren nem fut a szolgáltatást akkor a cloudflare a saját mentéséből elérhető verzióban tartja. Továbbá többféle teljesítmény és hatékonyság növeléséért felelős beállítását is aktiváltuk. A teljesítmény tesztelését pedig az általuk nyújtott eszköz segítségével mértük, amely teljes képet mutatott a saját szolgáltatásunk felületi minőségéről.

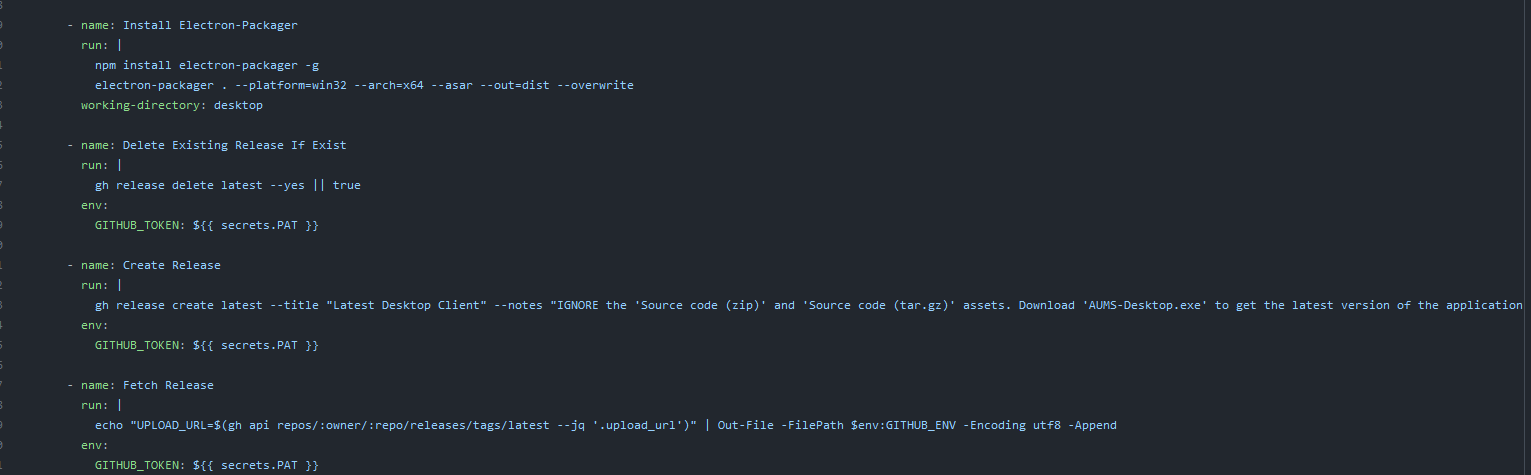
# GitHub (Action)

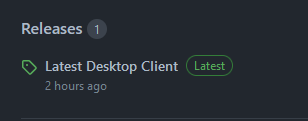
A csoportos munkát lényegesen segítette a github rendszere, hiszen a létrehozott projekt repository tartalmazta a legújabb, illetve a régebbi verziókat, így hiba esetén bármikor ellenőrzésre kerülhettek a korábbi lépések. A különböző ágak létrehozása pedig lehetőséget adott arra, hogy külön dolgozhassunk, majd az elkészített munkát a main (fő) ágban egyesítsük. Az egyesítések pull request formájában történtek, az új frissítések ellenőrzésére pedig kijelöltük a csapat más tagjait, hogy felülvizsgálják az újdonságokat és amennyiben megfelelőnek látták azt, elfogadták a kérelmet és frissűlt a főág az új kiegészítésekkel. Egészében a verziókezelés sokat segített a kivitelezésben, és törekedtünk arra, hogy az általa nyújtott legtöbb lehetőséget kihasználjuk.

A GitHub rendelkezik egy szolgáltatással, amely actions (akciók, tevékenységek) névre hallgat, és számunkra ez egy igen fontos szerepet vállalt a teljes rendszerben. Két actiont hoztunk létre, amelyek feladata az projekt automatizáció (CI/CD), folyamatos integráció és éles rendszerbe való helyezés, valamint a legfrissebb asztali telepítőkészlet létrehozása.

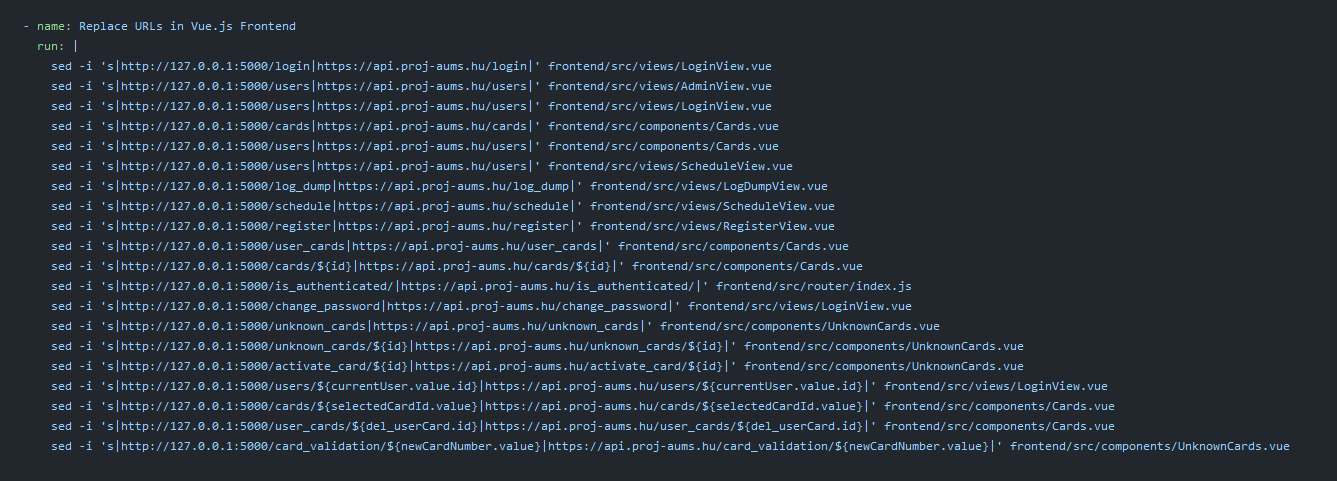


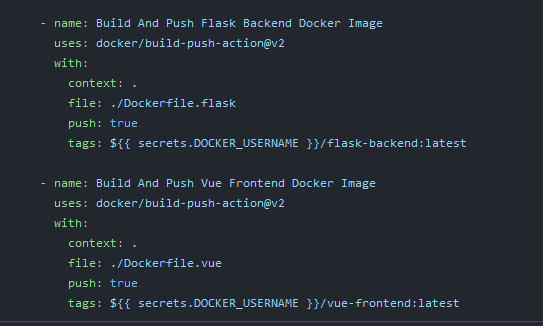
Az első action a desktop alkalmazásért felelős, a main ágba való feltöltéskor automatikusan lefut és mivel node.js került felhasználásra az asztali alkalmazás készítésekor, így az action egy virtuális, a projekt repository-tól független futási környezetbe feltelepíti a megadott node verziót. Ezután ezt felhasználva feltelepíti a desktop alkalmazás függőségeit és elkezdi a projekt futtatható állománnyá való alakítását az általa létrehozott környezetben. Úgy döntöttünk ennek a megjelenítése release formájában történik meg, ami azt jelenti, hogy az action lefutása után a projekt github repository oldalán a release fülre kerül a legújabb desktop frissítés. A felhasználó innen letöltheti a létrehozott exe állományt, amely lényegében az asztali telepítőkészlet. A futtatás és rövid telepítés után pedig már használható is az alkalmazás.





A második action a fontosabb, ez már egy komplexebb munkát végez. Szintén a main ágba való feltöltéskor indul el a futása, és a desktop action-el ellentétben, amely windows környezetet alakít ki, ez ubuntu rendszer alatt fut. Ennél az első lépés az, hogy a projekt beállításánál megadtuk a repository titkos értékeit. Ezek lényegében olyan környezeti változók, amit az action felhasználhat a futása során, viszont titkosak és nem kerülnek ki. Az action első lépéseként lefutó folyamat már egyből felhasználja ezen környezeti változók két tagját, amivel a docker hub-ra próbál bejelentkezni. A docker hub egy olyan oldal, amely docker image-ek tárolására alkalmas, számunkra ez azért fontos mert az action végén létrehozott frontend és backend image fájlok ide kerülnek feltöltésre, és a szerveren lévő watchtower képes arra, hogy ellenőrizze az itt fent megtalálható fájlokat és újabb verzió esetén letöltse azokat és frissítse az éles rendszert. Az action további lépésében létrehozza a virtuális környezetet, ellenőrzi a fájlok épségét és hibákat keres. Amennyiben hiba mentes minden állomány elkezdi a bizonyos részek cseréjét, hiszen a localhost futtatás során 127.0.0.1 ip címet használtunk a backend eléréséhez, az action pedig lecseréli ezeket a sorokat az “api.proj-aums.hu” megfelelő endpoint részeire. Az éles (production) rendszer számára továbbá szükséges több környezeti változó, ezeket is betölti az action a repository secret részekből és végül elvégzi a képfájlok létrehozását és feltöltését a docker hub-ra.





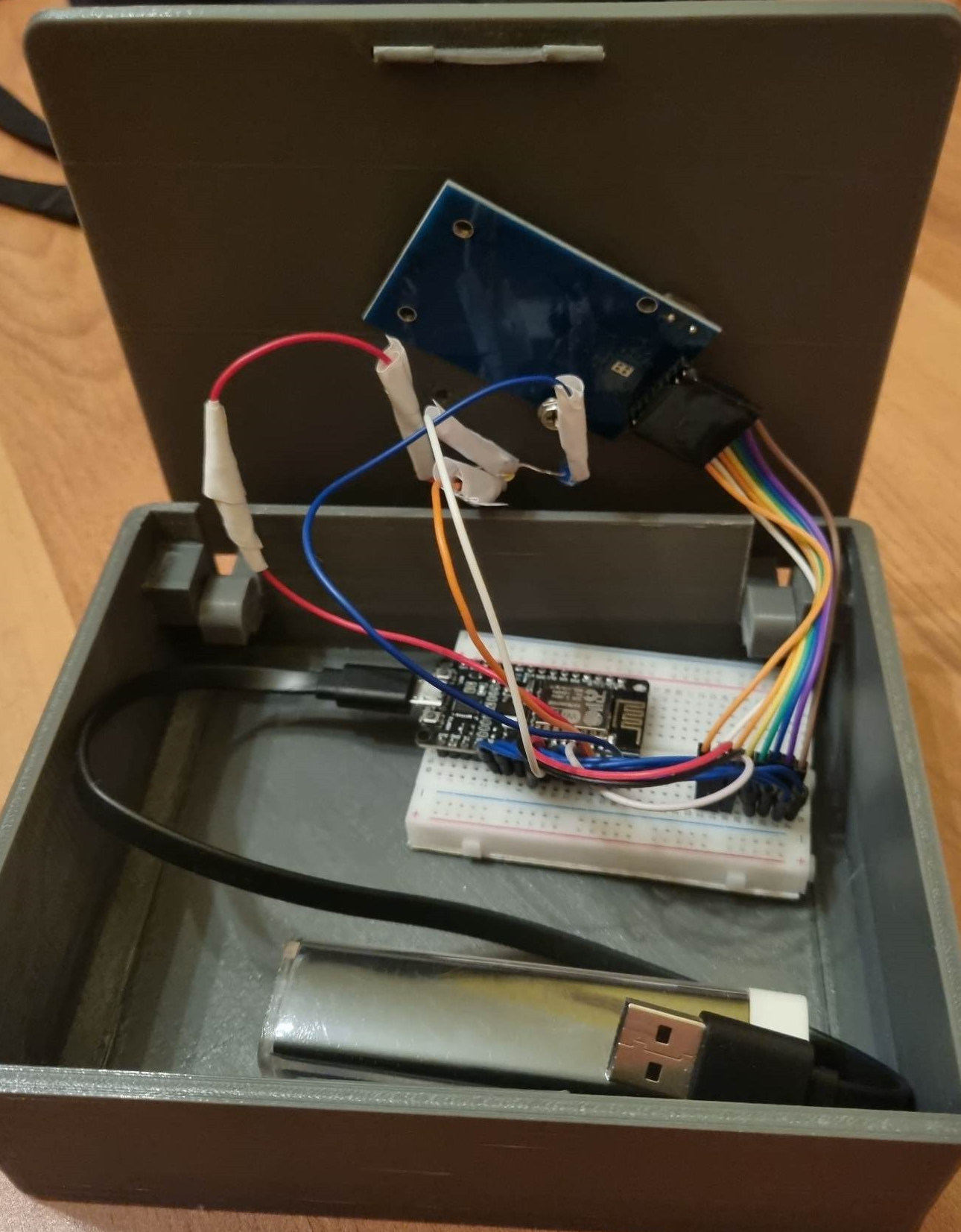
# Hardware

A beléptető rendszer megfelelő szimulációja érdekében készítettünk egy fizikai eszközt is, amely az éles rendszer többi elemével együtt működve tökéletesen bemutatja a belépés és felhasználó kezelés bizonyos részeit.

Az egész egy megtervezett, 3D nyomtatott dobozban helyezkedik el. A doboz tetejére került három státuszjelző LED (piros, sárga, kék), valamint a belülről a doboz közepére lett csavarozva a kártyák olvasására használt modul. A beépített LED-ek közül a piros a főként a hibás és nem megengedett belépést jelzi, a sárga egy státuszjelzés, hogy esemény történt, a kék pedig a sikeres kérést, kilépést/belépést jelzi.

Az olvasó egy RFID modul, a belépéshez használható kártyák vagy tag-ek pedig RFID chippel ellátott kártyák és tag-ek. Az egészet egybentartó komponens pedig egy NodeMCU, ez lényegében teljesen olyan, mint egy átlagos arduino eszköz, viszont a leglényegesebb különbség (ami miatt választottuk), hogy a wifi modul beépítetten érkezik az eszközzel. Az energiaellátásért pedig egy külső akkumulátor felelős.





Az beléptető doboz bekapcsolása után a sárga LED villogással jelzi, hogy várja az internet kapcsolatot, ez a kódban meghatározásra került és specifikusan a doboznak létrehozott hostot keres. Amennyiben sikeres a kapcsolat. már várja is a kártyák/tag-ek érintését.



Az érintés után a kártya/tag egyedi titkos azonosítóját az olvasó beolvassa, majd az erre kialakított backend végpontra küldi, ahol feldolgozásra és értelmezésre kerül. Végül a végpont válasza alapján dönti el, hogy milyen LED-et mutasson a felhasználó számára, ezzel átengedve vagy elutasítva a kérését.

# Adatbázis

## Általános leírás

Az adatbázis **ORM** (Objektum-relációs) leképezéssel készül el, amit a Python back end hajt végre. Az ORM keretrendszerek konceptuális absztrakciót nyújtanak az adatbázis rekordok és az objektumorientált nyelvekben található objektumok közötti leképezéshez. Az ORM segítségével az objektumok közvetlenül az adatbázis rekordokhoz vannak leképezve. Például, ha frissíteni szeretnénk egy felhasználó nevét az adatbázisban, elegendő egy egyszerű metódushívást végrehajtani, mint például: **user:updateName(‘Peter’)** (*Chen és mtsai*, 2016).

Azért választottuk ezt a leképezési technikát, mert ennek segítségével sokkal **dinamikusabban** tudjuk létrehozni és módosítani az adatbázisunkat. Továbbá, mivel az adatbázis-lekérdezések automatikusan generáltak és paraméterezettek, ezért az alkalmazásunk **védettebb** az **SQL injekciós támadások** ellen. Viszont fontos megjegyezni, hogy **teljes körű védelmet nem nyújt** ez a keretrendszer, ezért további **biztonsági intézkedéseket** szükséges végrehajtani.

Az ORM segítségével generált adatbázisunk nyolc táblából áll: **user, schedule, user\_card, card, user\_role, role és uknown\_card**.

## User tábla

A user tábla tárolja az összes felhasználó személyes adatait.

Mezők:

* id: int, elsődleges kulcs
* first\_name: varchar(100)
* last\_name: varchar(100)
* birth\_date: date
* phone\_number: varchar(50)
* address: varchar(100)
* company\_email: varchar(50), egyedi
* personal\_email: varchar(50), egyedi
* username: varchar(50), egyedi
* password: varchar(255)
* access\_token: varchar(255), egyedi, lehet NULL

## Schedule tábla

A schedule tábla tárolja a felhasználók be- és kilépési idejét.

Mezők:

* id: int, elsődleges kulcs
* user\_id int, idegen kulcs (sok az egyhez kapcsolat)
* enter\_date: datetime, lehet null
* leave\_date: datetime, lehet null

## Card tábla

A card tábla tárolja a hitelesített kártyák adatait.

Mezők:

* id: int, elsődleges kulcs
* card\_number: varchar(100), egyedi

## User\_card tábla

A user\_card tábla kapcsolja össze a user és a card táblákat.

Mezők:

* id: int, elsődleges kulcs
* user\_id: int, idegen kulcs, egyedi (egy az egyhez kapcsolat)
* card\_id: int, idegen kulcs, egyedi (egy az egyhez kapcsolat)

## Role tábla

A role tábla tárolja felhasználóknak kiosztható szerepköröket.

Mezők:

* id: int, elsődleges kulcs
* name: varchar(50), egyedi
* level: int, egyedi

## User\_role tábla

A user\_role tábla kapcsolja össze a user és a role táblákat.

Mezők:

* id: int, elsődleges kulcs
* user\_id: int, idegen kulcs, egyedi (egy az egyhez kapcsolat)
* roler\_id: int, idegen kulcs, egyedi (egy az egyhez kapcsolat)

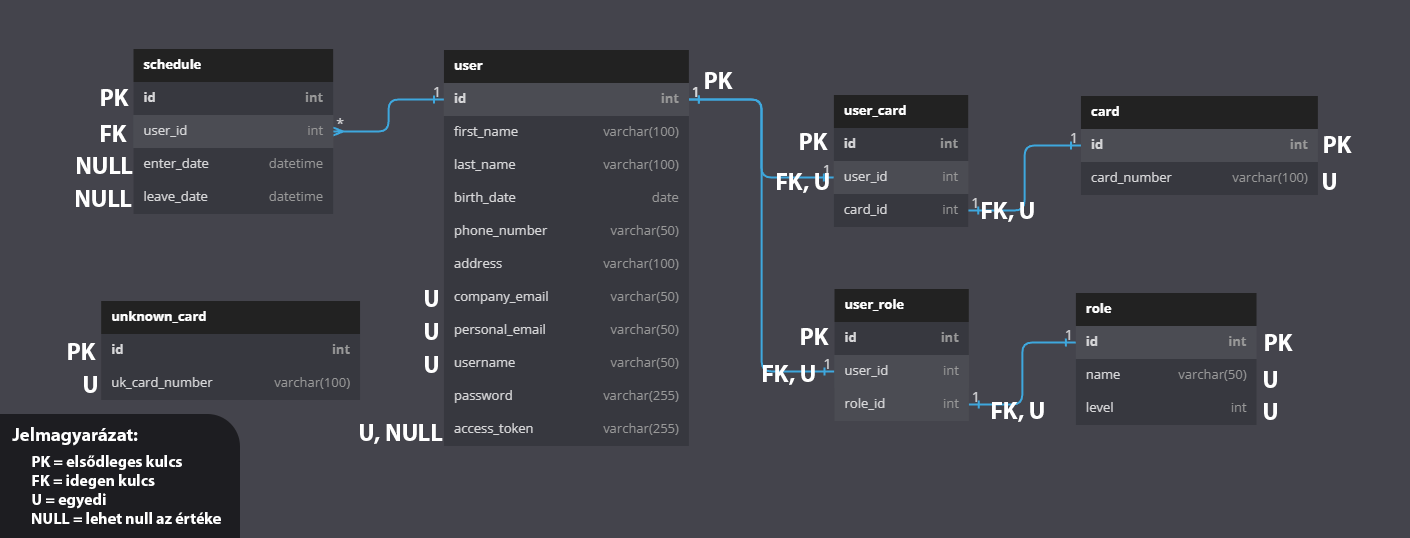
## Unknown\_card tábla

Az unknown\_card tábla biztonsági célból tárolja a még nem hitelesített kártyákat. Az adminisztrátor szintű felhasználók ezeket a kártyákat hitelesíthetik és áthelyezhetik a cards táblába.

Mezők:

* id: int, elsődleges kulcs
* uk\_card\_number: varchar(100), egyedi

## EK diagram



# Backend

## Általános leírás

A szerver-oldali komponensünknek a Python nyelvet választottuk. A **Flask** keretrendszer segítségével építettük ki a **REST API** rendszerünket. A felépítés tagolt, model-controller-service struktúrát követ. Az adatbázis leképezés az **SQLAlchemy** eszközkészlet segítségével valósul meg. Az egyszerűbb strukturált felépítés minden esetben az adatbázis elkészítésével indul. A models mappában hozzuk létre a táblákat, kapcsolótáblákat objektumorientált kód felépítéssel, illetve az inicializálást.

Ezek után a controllers mappában elkészítjük a modellekhez tartozó kontrollereket. Egy controller állhat akár egy sorból is, ami olyankor egy importálást jelent a models mappából, vagy hasonlóképp, mint a modelleknél, **OOP** struktúrát alkalmazva használhatunk különböző osztályokat (amik később a végpontok elkészítésében sokat fognak segíteni), amikben a kívánt kéréseinket tudjuk meghatározni (**GET, POST, PATCH, DELETE**). Továbbá a Python rendelkezik úgynevezett dekorátorokkal, amik segítségével meghatározhatjuk például, hogy bizonyos kérések végrehajtásához / végpontokra való navigáláshoz szükséges azonosítás access token segítségével, vagy csak egy bizonyos felhasználói szint felett érhető el. A kontrollerek metódus hívásokért és azok vissza adásáért felelnek. A metódusokat a services rétegből hívjuk meg és a regisztráció, bejelentkezés stb. végrehajtásáért a services réteg metódusai a felelősek.

A services rétegben található az összes adatbázis-lekérdezést végrehajtó kód, a válaszüzenet visszaadás státusszal, üzenettel és státusz kóddal pl.: **return \_response("success", "Card has been added to the database", 201)**. Ebben a rétegben valósítjuk meg a felhasználó validálást, a jelszó titkosítást, email küldést a regisztrált felhasználó e mail címére, továbbá minden egyéb validációs feladatot mielőtt elküldené az adatbázisba a kapott adatokat az adott metódusunk.

## Modellek

Itt készítjük el az adatbázis létrehozásához szükséges modelleket, OOP struktúrával, ami alapján az SQLAlchemy eszközkészlet leképezi az adatbázist

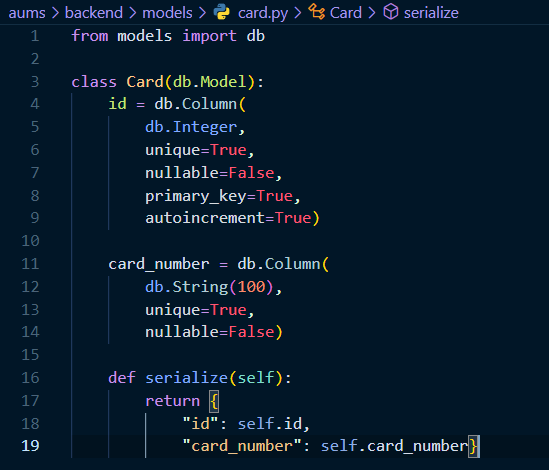
### Model Inicializálás

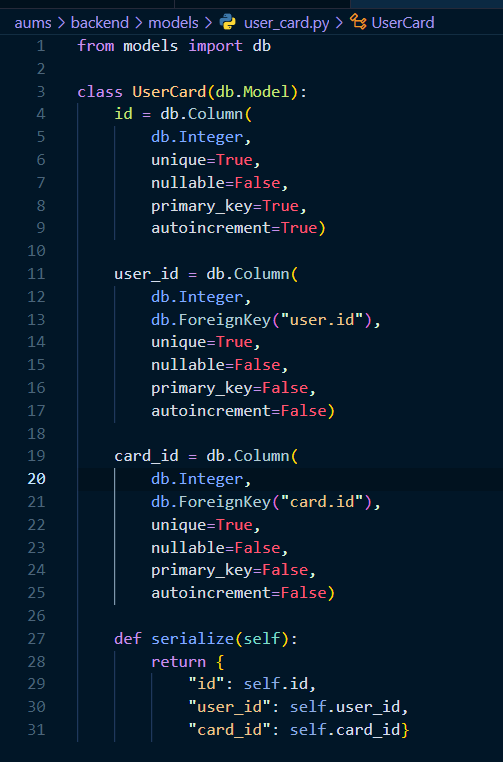
A model mappában található \_\_init\_\_.py-al inicializáljuk az adatbázis kezelését, és létrehozunk egy db nevezetű objektumot, amit később a modelljeinkben fogunk használni

### Model

Amint az alábbiakban látható, importáljuk az előzőekben elkészített db objektumot, majd az objektum különböző tulajdonságaival létrehozunk egy modellt és végül bizonyos modelleknél hozzáadunk egy serialize függvényt ami JSON formátumban visszaadja nekünk az adott modellt amikor a service rétegben ezt meghívjuk (pl.: **return [item.serialize() for item in items]**).

Néhány model:





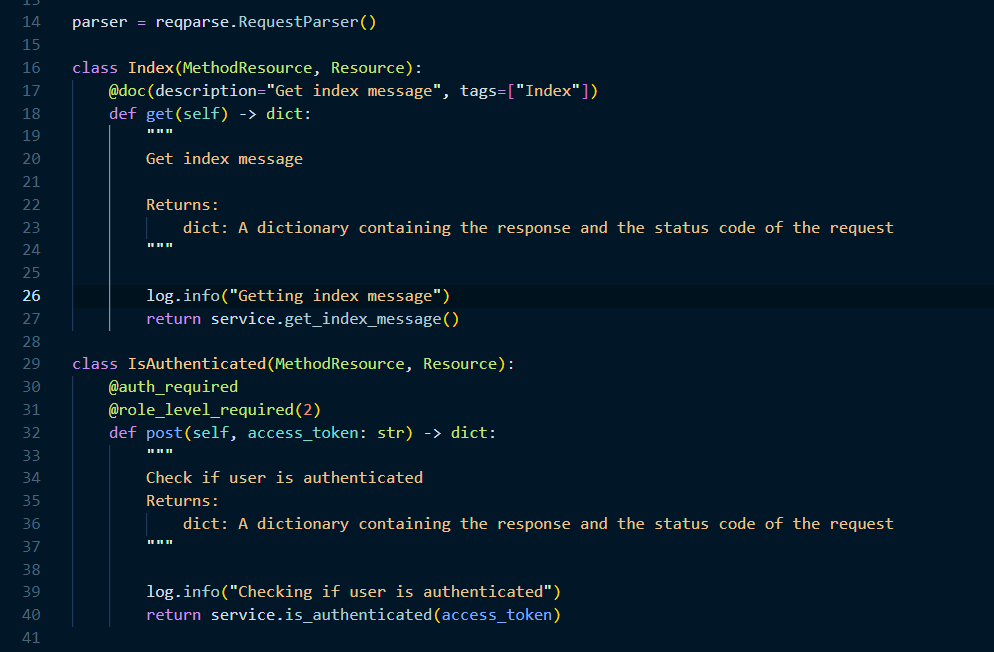
## Kontrollerek

Mindegyik modellhez készíteni kell egy kontrollert, hogy az SQLAlchemy tudja, hogy melyik táblákat kell létrehoznia az adatbázisban. Továbbá a végpontokra küldött kérések elfogadott típusait is itt tudjuk meghatározni. Bizonyos kérésekhez argumentumokat kell hozzárendelni, ilyen esetekben használjuk a RequestParser() metódust a flask\_restful csomagból. a parser segítségével több argumentumot is tudunk fogadni és elemezni. Meghatározhatjuk az argumentumok nevét és típusát is.

A kontrollerekben dekorátorokat használtunk a Swagger leírás létrehozásához és végpontok csoportosításához. Ezen felül a dekorátorokkal meghatároztuk azt, hogy néhány végpont elérése egy bizonyos szerepköri szint alatt nem lehetséges.

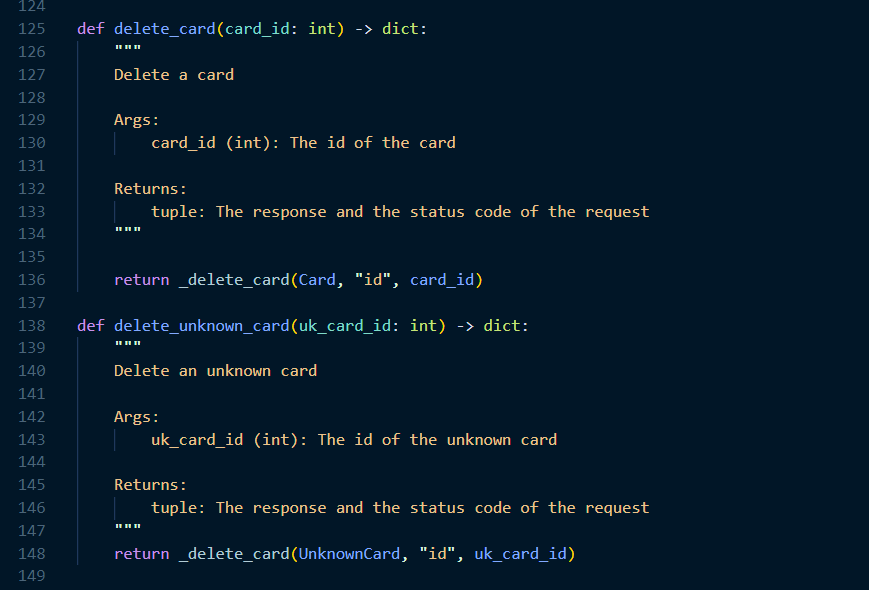
Továbbá az index vagy login végpontok kivételével mindegyik végponthoz szükséges az azonosítás, ami access tokennel történik.

Végül a kontrollerek a beérkező kéréstől függően meghívják a service rétegben lévő metódusokat.



## Szolgáltatások

Itt hajtjuk végre a beérkező kérésekkel kapcsolatos feladatokat, amiket a kontrollerek továbbítanak, ellenőrizzük a kapott adatokat, kommunikálunk az adatbázissal és küldjük a választ a kliens oldalnak. A különböző model - kontrollerekhez tartozó feladatokat elkülönítettük a service rétegben is (card\_service, user\_service, index\_service, stb.) a jobb átláthatóság szempontjából. A service rétegben továbbá létrehoztunk segéd metódusokat is az újrafelhasználhatóság miatt. Mivel ezek a segéd metódusok generikusok, ezért több service rétegbeli függvény is meghívhatja őket, ezért a kód nagysága is csökkenni fog



## Segédfüggvények

Ebben a mappában találhatók a kisebb feladatokat ellátó metódusok. Ilyen például a log készítése, email küldése a felhasználónak, a back end leállítása hiba esetén és a környezeti változók lekérése az eszközről.

Az email küldés egy komplexebb folyamat az alap kisegítő eszközökkel szemben, hiszen itt a backend a “proj-aums.hu” címet felhasználva továbbít üzenet a felhasználó privát email fiókjába. Ezt úgy érjük el, hogy a “proj-aums.hu” domain mellé létrehozásra került egy saját SMTP szerver, és a megfelelő port, felhasználónév-jelszó páros megadásával már képes a postmaster nevében üzenetet küldeni a megadott üzenettel, feltételekkel. Továbbá a “mail.proj-aums.hu” megadásával elérhető a privát levelező kliens is.

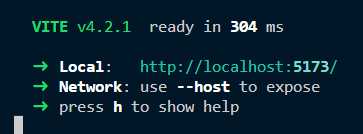
## Konfiguráció, környezeti változók és futtatás

Az applikáció helyi gépen történő elindításához használjuk a VsCode terminálját. Az applikációt futtatni a következőképpen tudjuk:

1. a VsCode felső eszköztárán kattintsunk a terminal menüre, majd New Terminal (vagy **Ctrl+Shift+ö** kombináció)
2. az alul megjelenő ablak jobb oldalán kettő powershell gombot kell látnunk, amennyiben csak egy van, nyissunk még egy terminált a plusz jellel a powershell felett (vagy **Ctrl+Shift+ö** kombináció)
3. Az egyik powershellben navigáljunk a backend mappába (**cd backend**), majd adjuk ki a következő parancsot: **py app.py** (ez elindítja a backend applikációt a **localhost:5000**-es porton)
   1. sikeres futtatásnál az alábbi üzenet fogad minket:



1. Nyissuk meg a másik powershellt, majd navigáljunk a frontend mappába (**cd frontend**) ezután adjuk ki a következő parancsot: **npm run dev**
   1. sikeres futtatásnál az alábbi üzenet fogad minket:



1. Amennyiben a backend indítás és a frontend indítás is sikeres volt, nyissuk meg a frontend powershellben található localhost linket **Ctrl+bal kattintás**sal
2. A megjelenő oldalra az alábbi adatokkal tudunk belépni:
   1. **felhasználónév: admin.admin**
   2. **jelszó: admin**
3. Sikeres belépés után tudjuk tesztelni a különböző oldalakat, funkciókat. Az adminnak alapértelmezetten nincs belépés/kilépés ideje a beosztásban, ezért nem lehet kiválasztani a Schedule fülön, továbbá alapértelmezetten kártyák sem léteznek az adatbázisban, de tesztelés szempontból elérhető egy **add card** gomb (A kártyákat a hardvereszközhöz érintve lehet hozzáadni).

Annak érdekében, hogy az alkalmazásunk egymástól függetlenül működjön a szerverünkön, a helyi gépeken és az egység teszteket se befolyásolják a különböző beállítások, létrehoztunk egy konfigurációs fájlt, amiben elkülönítettük a szerver, a helyi gép és a tesztek adatbázis útvonalát és egyéb beállításaikat.



A környezeti változóknál (.env fájl) állítjuk be a levelezéshez szükséges portot, jelszót, szervert, felhasználónevet és a szerveren futó adatbázis útvonalát.

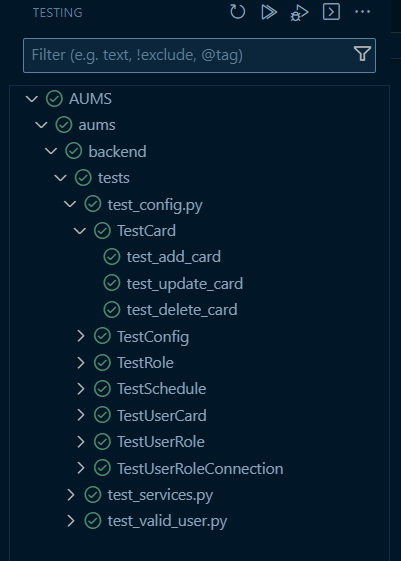
A create.py fájl felel a back end applikáció és API objektumok elkészítéséért. Az ebben található függvény (amely létrehozza az appot) egy konfigurációs objektumot vár paraméternek, amit az előbb említett konfigurációs fájlban találunk. Továbbá itt engedélyezni kell az eredetközi erőforrás megosztást (CORS), hogy a beérkező kéréseket ne dobja vissza a backend CORS hibákkal. Majd ezután a log fájlba írunk, hogy sikeresen elkészült az app és végül a elkészítjük a Swagger dokumentációt az applikációnkhoz.

Végül az app.py-ban állítjuk be hogy melyik konfigurációt fogjuk használni a futtatásnál és beállítjuk a levelezést. Ez után létrehozzuk az összes szükséges végpontot (pl.: /login). Ezek után a Swagger-be is regisztráljuk a végpontokat, amit a swagger magának automatikusan elkészít. Majd jön az adatbázis elkészítése. Úgy oldottuk meg ezt a részt, hogy amennyiben az adatbázis még nem létezik, az applikáció létrehozza azt és feltölti alapértelmezett felhasználókkal, titkosított jelszóval, létrehozza a szerepköröket, majd minden szükséges kapcsolatot. A két alapértelmezett felhasználó az Admin és a Hardware, amik szükségesek a projekt működéséhez. Ezután (illetve, ha már létezik az adatbázis) a log fájlba belekerül, hogy elindult az applikáció és a választott konfiguráción elindul az app.

## Tesztek

Az egység teszteket az úgynevezett Setup/Teardown eljárással készítettük el és az SQLite adatbázist a memóriában tárolja el. Ezek által az adatok gyorsabban elérhetők és nem okoznak esetleges konfliktusokat a helyi és a szerveren létrehozott adatbázissal. A Setup/Teardown stílus segítségével minden teszt futtatása előtt létrejön az adatbázis, ezután lefut a teszt, majd minden teszt lefutása után törlődik az adatbázis. Ezt az eljárást azért választottuk, mert gyors, izolálható a többi adatbázisunkból és egyszerű a kezelése.

Az átláthatóbb tesztesetekhez létrehoztam osztályokat, amiket olyan csoportokba sorolhatunk, mint például kártyák tesztjei (TestCard osztály), konfigurációs teszt (TestConfig) stb. Az osztályok miatt, a VsCode a megfelelő csoportokra bontotta a teszteket, így sokkal átláthatóbb és egyszerűbb a tájékozódás a Test fülön.



### Tesztek inicializálása

Hasonlóképp, mint a modelleknél láthattuk, itt is kell létrehozni egy inicializáló fájlt, hogy az egységtesztek felfedezhetők legyenek. Jelen esetben egy teljesen üres fájlt kellett létrehozni \_\_init\_\_.py néven. Az egység tesztekhez a pytest könyvtárat használjuk.

### Beállítások és problémák kiküszöbölése

Az egység teszteket a Visual Studio Code-ban a bal oldalon található Testing fülön lehet elérni.

Kezdetben előfordulhatnak hibák, mint például a Flask nincs telepítve, pytest nincs telepítve stb. ebben az esetben egy **új terminált** kell nyitnunk VsCode-ban, el navigálni az alkalmazás gyökérkönyvtárába **(cd mappanév**), ezután, hogy megbizonyosodjunk arról, hogy valóban a gyökérkönyvtárban vagyunk, adjuk ki a következő parancsot: **ls**. Ez kilistázza a jelenlegi könyvtárban található mappákat, fájlokat és ha látjuk a requirements.txt nevű fájlt, akkor a terminálban kiadhatjuk a következő parancsot: **pip install -r requirements.txt** (ez feltelepíti az összes szükséges könyvtárat az eszközünkre).

További beállításra volt szükség ahhoz, hogy a VsCode megtalálja az útvonalat a tesztekhez, ezért létrehoztam a pytest.ini fájlt a gyökérkönyvtárban (Ezzel nekünk nem kell foglalkoznunk)

A Konfiguráció, környezeti változók és futtatás pontban említett konfigurációs fájl segítségével ki kellett küszöbölni olyan problémákat a sikeres tesztekhez, amik a helyi gépeken futtatott beállításokban és a szerver beállításokban nem okoztak problémát. Ezekhez tartozik például az email küldés a személyes email címekre a generált bejelentkezési adatokkal, illetve a levél küldés kiiktatása miatt a bejelentkezés lehetetlenné vált a teszkörnyezetben. Az első hibát a SEND\_EMAILS változó segítségével küszöböltem ki, a bejelentkezéshez viszont ezen kívül kellett még módosítást végezni a user\_service fájlban. A sikeres tesztek elvégzéséhez további paramétereket kellett létrehozni a felhasználó regisztrálásánál:

***def register\_new\_user(args: dict, password: Optional[str] = None, return\_password: bool = False) -> dict:***

***if return\_password:***

***return {***

***"status": "success",***

***"message": "User successfully registered" }, 201, password***

***return {***

***"status": "success",***

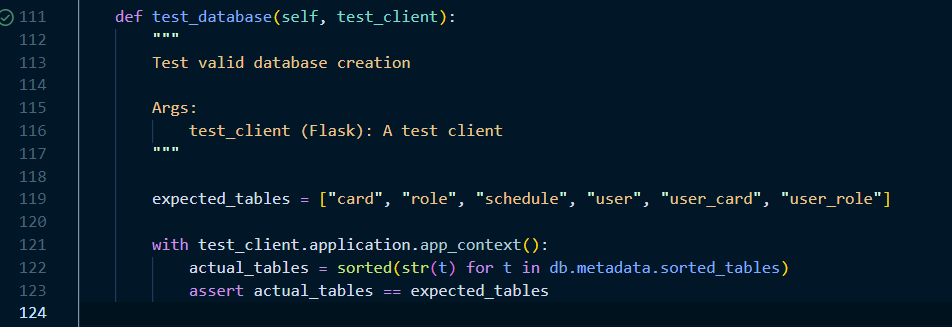
***"message": "User successfully registered" }, 201***

A return\_password paramétert a regisztrációs teszt metódus adja át ennek a rétegnek, ez által az esetleges adat kiszivárgás és a teszt bejelentkezéshez szükséges jelszó hiánya is egyaránt ki lett küszöbölve.

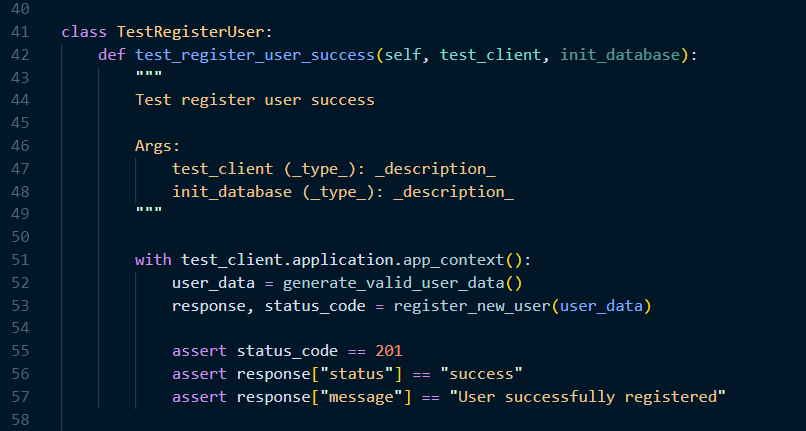
### Tesztelés

Több mint 25 egységteszt készült el, a helyes adatbázis konfigurációs teszttől egészen a felhasználó belépés tesztig. Ezek közül csak néhány fontosabbat említenék meg.

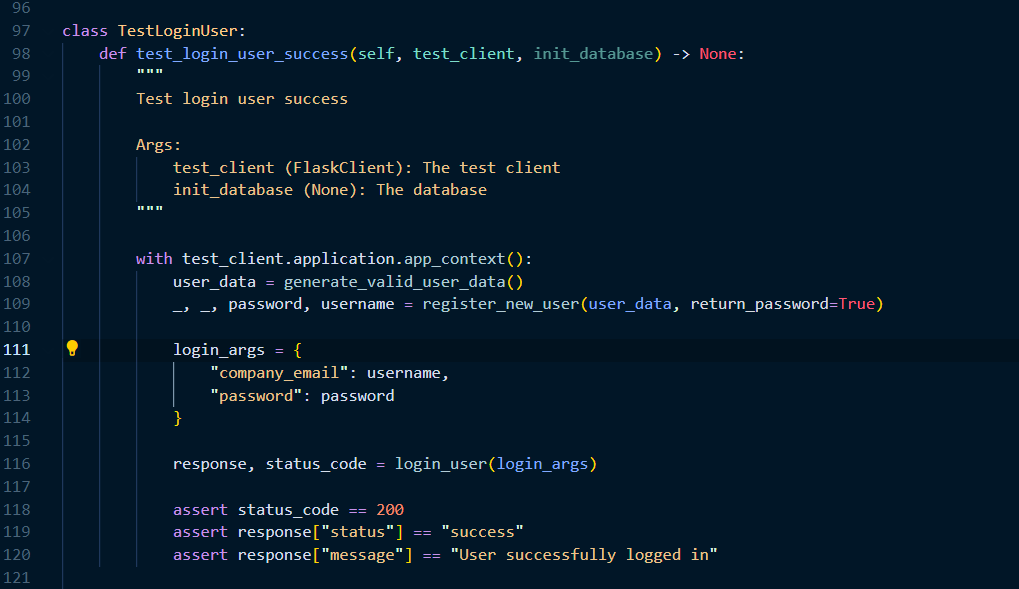
A test\_database egységtesztnél megvizsgáljuk, hogy a helyes táblák jönnek-e létre (az oszlopokról feltételezzük, hogy helyesek).



A test\_register\_user\_success tesztnél egy előre elkészített felhasználó adatait küldjük el a services rétegnek, majd a kérelmünkre érkező választ hasonlítjuk össze az elvárt eredményekkel.



A test\_login\_user\_success tesztnél ismét a felhasználó regisztrálásával kezdünk, viszont itt átadjuk a return\_password paramétert is, mivel a fent említett Beállítások és problémák kiküszöbölése menüpontban az email küldést kikapcsoltuk a teszt beállításoknál. Mivel nekünk a regisztráció után visszakapott felhasználónév és a jelszó a fontos, ezért ezeket eltároljuk egy változóban, a másik két visszakapott értéket (válasz, státuszkód) eldobható változóknak adjuk át (\_, \_,). Ezután egy változónak adjuk a bejelentkezési adatokat, és a teszt meghívja a service rétegbeli bejelentkezési függvényt. Ennek a függvénynek a válaszát és a státusz kódját hasonlítjuk össze az elvárt értékekkel.



Ezeken az egységteszteken kívül további tesztek azt vizsgálják, hogy például hozzá lehet e adni egy kártyát, vagy szerepkört az adatbázishoz, azokat lehet e módosítani, illetve törölni. A regisztrációnál és a bejelentkezési teszteseteknél is vannak következetesen hibás adatokkal történő tesztelés is.

# Frontend

## Alkalmazott szoftver eszközök

* Visual Studio Code
* Google Chrome

A REST API elkészítéséhez a Python programozási nyelvet használtuk, amiről a Back End pontban több információ található.

A kliens oldali komponensnek a Vue.js keretrendszert választottuk. A keretrendszer segítségével a fejlesztés hatékony volt, mivel a HMR (Hot Module Replacement) funkció segítségével azonnal láttuk a fejlesztett oldalakon a változást. A keretrendszerhez további csomagokat is használtunk, mint például az axios (A kérések küldésére/válaszok fogadására a back end-től), bootstrap (a reszponzív megjelenéshez). Továbbá a Vue.js 3-ba épített Composition API használatával rendezettebb kód, egyszerűbb kezelés és átláthatóságot biztosítottunk a front end részen.

## Nyelvhasználat

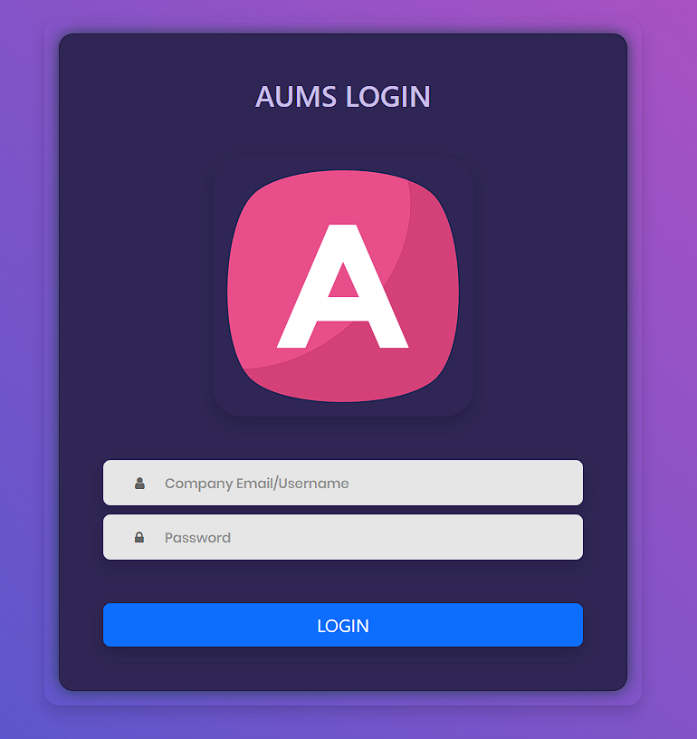
* HTML
* Javascript
* CSS

## Megvalósítást segítő keretrendszerek

* VueJs
* Bootstrap

## Bejelentkező felület

Egy olyan dizájn létrehozása volt a cél, amely megállja a helyét a mai letisztult kinézetet uralta világban, viszont nem a megszokott, fekete-fehér-szürke színek dominálásával, hanem egy kis élénk színösszeállítással, amely kevésbé használatos manapság.



A weboldal megnyitása után ez a felület fogad minket, ahol kettő input mező és egy, a bejelentkezést végrehajtó gomb található.

Ahhoz, hogy egy zöld animáció működjön az input mezőkbe kattintás után, átlátszóvá, illetve szürkévé kellett módosítani a css állományban.

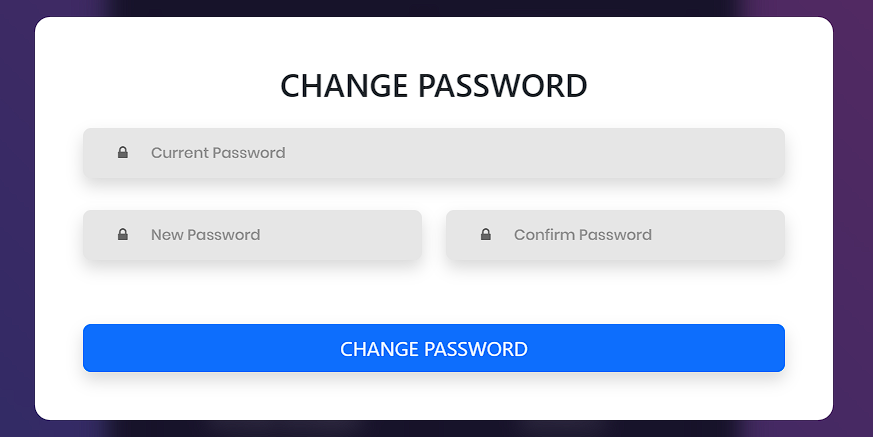
Az animáció egy kis zöldes felvillanás, amely elhalványul a mezők körül, illetve a kis icon-ok is színt váltanak és balra csúsznak.

A weboldalon összes oldalán egységesítve van ez a lila-rózsaszín átmenetes színösszeállítás, amit az alábbi módon állítottunk be:

Az bejelentkezés először egy validációval kezdődik, ahol megnézzük, hogy ha emaillel próbál a felhasználó bejelentkezni, akkor az csakis a cég által automatikusan generált emaillel lehessen vagy a generált felhasználónévvel. Validáció után egy ***axios*** Post küldés segítségével küldjük el a bejelentkező személy által megadott adatokat a futó backend felé, ami azonnal lekéri az adatokat az adatbázisból. Az adatbázisban ***hash***-elve vannak a jelszavak eltárolva, ezért backend részen ***bcrypt*** segítségével ellenőrizzük, hogy a weboldalról küldött, illetve az adatbázisban szereplő kódolt jelszó megegyezik-e. Ha minden stimmel egy tokent (access token) küld vissza, amit innentől kezdve minden egyes axios kérés és/vagy küldésnél továbbítani kell. A későbbiekben ki lesz fejtve, hogy erre miért van szükség.

Ha a kettő adat közül valamelyik hibás volt, akkor azt egy felugró ablak jelezni fogja nekünk, viszont, ha minden sikeres akkor a felület továbbít minket a „Home” oldalra, ahol az adott személy az adatait éri el és ellenőrizheti és a jelszavát módosíthatja.

## Home

Bejelentkezés után a “Home” felület fogad minket. Itt az adott felhasználó adatai látszódnak, melyeket ellenőrizhet és elírás vagy probléma esetén értesítheti az illetékeseket, hogy javítsák az elírást és/vagy problémát. Ezen kívül lehetőség van új jelszó beállítására, ami első bejelentkezés után javasolt is, mivel minden felhasználó létrehozásakor email-ben küldünk értesítőt az ideiglenes jelszóról. Az új jelszó beállításához meg kell adni a régi (vagyis az előző) jelszót, illetve kétszer az újat. Természetesen, ha a megadott új jelszó és annak megerősítése nem egyezik, akkor azt a felület jelzi.

## Register

Ebben a menüpontban lehetőségünk van hozzáadni egy új felhasználót. A felhasználónak a valós adatait kell megadni, különösképpen a személyes email címet, mivel erre a címre fogja küldeni a szerver a belépéshez szükséges adatokat. A regisztrálást csak 5-ös szintű felhasználó végezheti el, illetve csak neki jelenik meg a kliens oldali menüpont.

## Cards

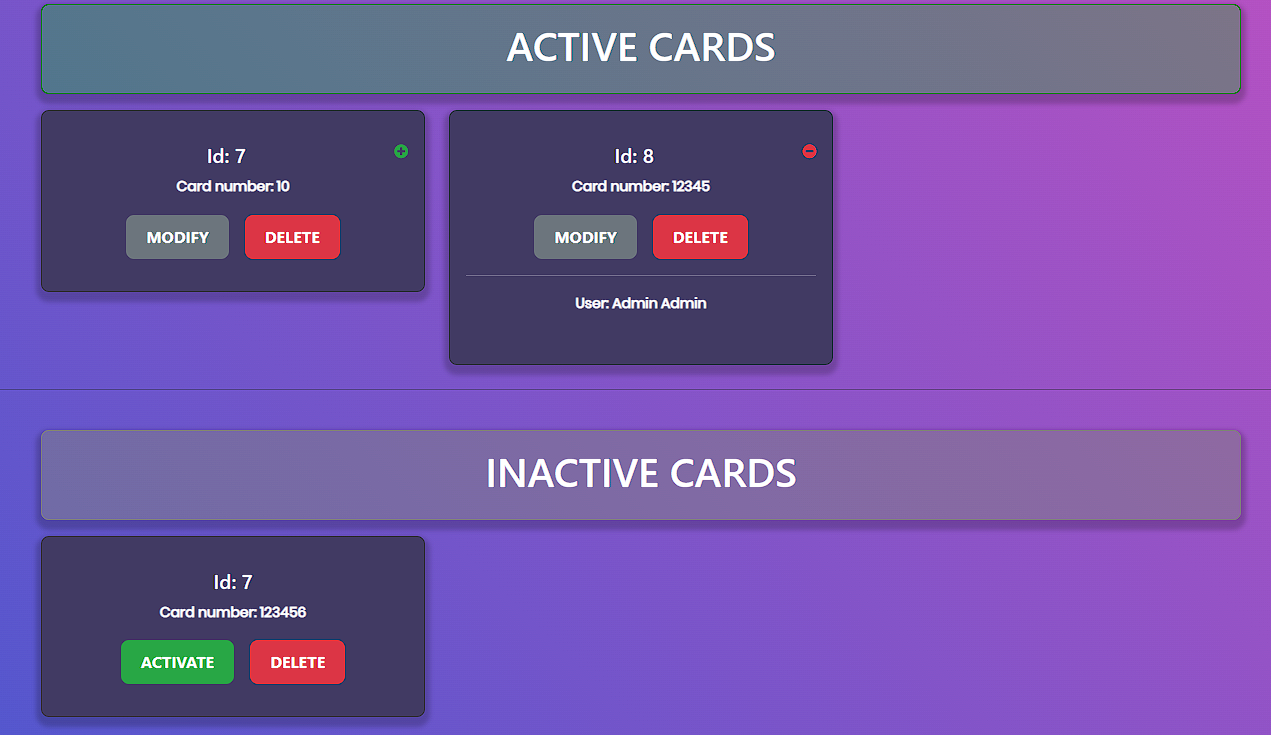
Ezt a felületet csak admin joggal rendelkező személyek érhetik el. Itt lehet az összes kártyát kezelni:

* aktiválni
* törölni
* módosítani
* személyt hozzárendelni
* személyt lekapcsolni

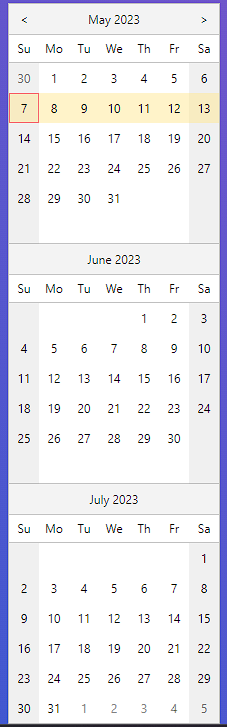
## A folyamat az alábbi módon zajlik:

1. Egy kártya első hozzáérintése után bekerül az “Ismeretlen kártyák” fül alá. Ilyenkor a beléptető rendszert imitáló doboz LED jelzésekkel jelzi, hogy ez a kártya ismeretlen.
2. Ezután lehetőségünk van aktiválni vagy törölni az adott kártyát.
3. Aktiválás után, ha újra hozzáérintjük a kártyát akkor egy másik LED jelzést ad a készülék, ami azt jelenti, hogy már aktív a kártya, viszont még nincs senkihez hozzárendelve.
4. Most lehetőségünk van a kártyának az adatait módosítani, személyt hozzárendelni. Személy hozzárendelése a kártya jobb felső sarkában lévő “+” -jellel lehetséges. Ha a kapcsolatot el szeretnénk távolítani, akkor egy foglalt kártyán a “+” jel egy “-” jellé válik, amit erre a funkcióra lehet használni.
5. Egy már aktivált, foglalt kártyát, ha hozzáérintünk ismét a beléptető dobozhoz egy újabb LED jelzés fogad minket, ami a “szabad a belépés” -t jelenti.

Kártyát csak akkor lehet törölni, ha az nem foglalt. Ha személy hozzá van rendelve a kártyához, akkor a felület jelzi, hogy először távolítsd el a kapcsolatot a fentebb említett módon. Mivel minden kártyának az azonosítója egyéni, ezért nem kell a duplikációtól félni sem.



## Schedule

Itt találhatóak az összes felhasználónak a be- és kilépései egy táblázatba szedve. Jelenleg minden felhasználónak van lehetősége mindenkinek a munkaidejét látni, amit egy <select> mezőből lehet kiválasztani. Összesen kettő táblázat van a felületen, az egyik a be- és kilépéseket jeleníti meg napra lebontva, a másik pedig egy naptár, ahol ki tudjunk választani a hónap, év melyik hetét jelenítse meg a főtáblán. A naptár eltüntethető egy kis fehér nyilacskával, hogy a főtábla kitölthesse a teljes felületet, ezzel megkönnyítve mobilon is a navigálást.

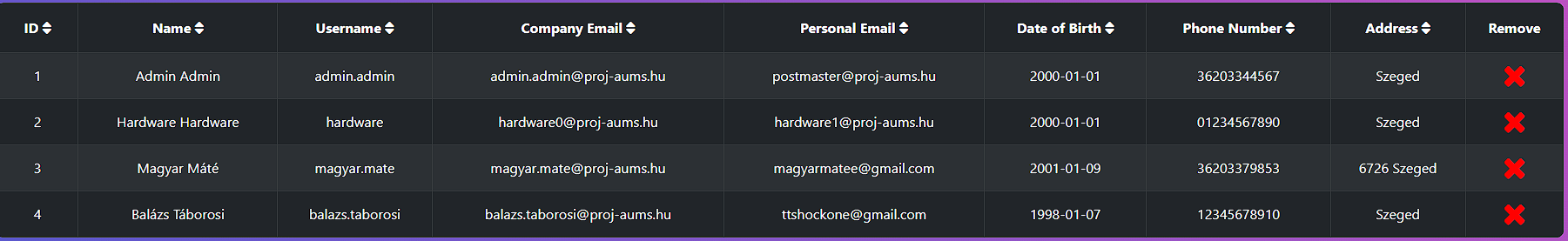
A főtáblát és melléktáblát egy DayPilot-Lite nevű eseménynaptár segítségével hoztuk létre. A melléktábla egy “DayPilotNavigator”, a főtábla pedig “DayPilotCalendar” (*npm*, 2023).

A főtáblán úgy nevezett “event” -ekként, azaz eseményenként jelentjük meg a felhasználók munkaidejét. Ezeket axios kéréssel kérjük le, amely egy JSON adathalmazt ad vissza a dátum kezdeti és vég (ha van) értékével. Minden ilyen esemény kattintható. Kattintás után egy modális ablak jelenik meg ami tartalmazza a pontos belépésnek, illetve kilépésnek (ha van) az idejét, és a dátumát. Erre azért is volt szükség, mert míg számítógépen keresztül megnyitva minden nap szépen látszódik, ez mobil már kevésbé. Annak érdekében, hogy ezt ki tudjuk küszöbölni, létrehoztuk az események interaktivitását, azért, hogy pontosan látható legyen minden lényeges adat az adott eseményről (*npm*, 2023).

A kártya első odaérintése a terminálhoz számít belépésnek, amit azonnal jelez is nekünk a felület, a második a kilépést. Ezzel a módszerrel mindenkinek lehetősége van akár napi vagy óránkénti többszörös be- és kilépésre, amelyeket listázva megtekinthetünk.

A schedule modul továbbfejlesztéséhez jó kiindulópont lehet az, ha csak az 5-ös szintű felhasználónak, azaz adminnak lenne lehetősége az összes kártyás be- és kilépési adat elérésére.

## User Management

Ebben a menüpontban érhető el az összes felhasználó, melyek törölhetőek, módosíthatóak. A megjelenítés egy táblázat formájában van, amelynek bármely oszlopára lehet rendezni az oszlop címére kattintva. A táblázat nagyrésze Bootstrap segítségével lett létrehozva. A felület azért készült, hogy az adminok minél egyszerűbben tudják a felhasználók adatait módosítani, vagy esetlegesen felhasználót törölni.

A sorok szerkesztéséhez a táblázat bármely sorába lehet kattintani, ami előhoz egy modális ablakot a jelenlegi adatokkal, amit azonnal lehet módosítani.

## Log

A back end-ben létrehozott log\_dump végpontot ezen az oldalon tudjuk lekérni, mindezt azért készítettük, hogy az esetleges hibákat, figyelmeztetéseket, illetve kéréseket jobban átlássuk. Kezdetben a logból nehéz volt kiszűrni, hogy mik is történtek, mivel egy összefűzött string-ként viselkedik, amiben többféle hasznos információ tárolódik minden kérésnél és folyamat végrehajtás után: időpont, szint (info, warning, error) és az üzenet, hogy mi is történt. A jobb átláthatóság miatt ezeket egy reszponzív táblázatba helyeztem és a különböző szinteket más-más sor színekkel jelöltem meg a táblázatban.

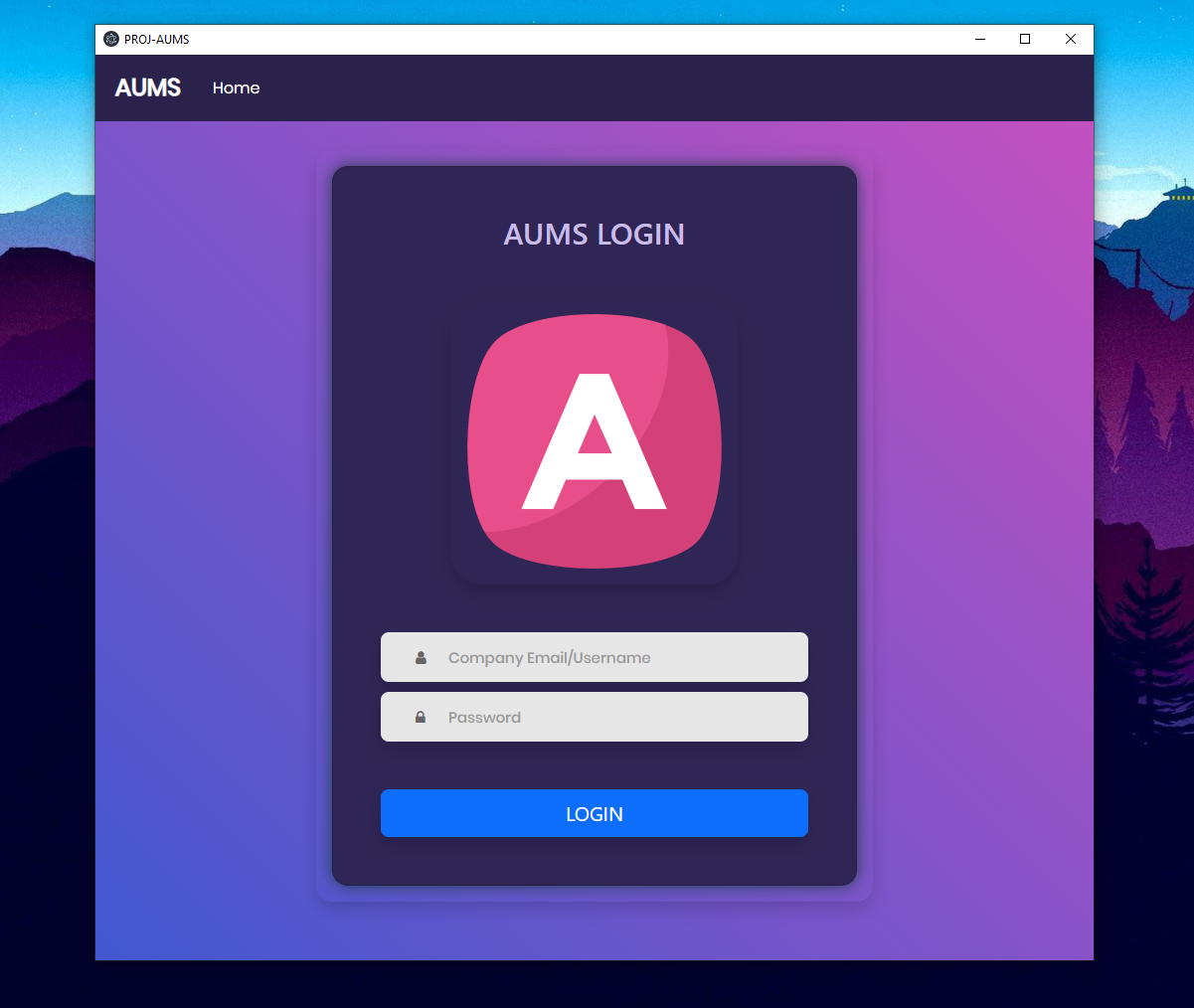
Azonban, mivel előfordulhat az, hogy a log több ezer soros lesz idővel, ezért átalakítottuk, hogy rendezhető legyen a lekért adat a táblázatban. Az adatokat lehet rendezni növekvő és csökkenő sorrendben az adat szintje és ideje alapján egyaránt.

Ennek az oldalnak a létrehozásával jelentősen csökkent a hibajavítás időigénye.

# Desktop

Az asztali alkalmazásunkhoz az Electron.js keretrendszert használtuk. Ez a keretrendszer egyszerű, nyílt forráskódú szoftver. A mi esetünkben, egy böngésző ablakba tölti be a weboldalunkat, amivel ezután ugyanúgy interaktálhatunk, mintha egy böngészőben nyitottuk volna meg.

Ez az asztali alkalmazás egy javascript fájlból, a hozzátartozó csomagokból és az AUMS ikont tartalmazó mappából épül fel.



# Felhasznált források

Chen, T. H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2016). Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, *42*(12), 1148-1161. Letöltés: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7451264>, 2023. 05. 02.

npm (2023): DayPilot Lite for Vue. Letöltés: <https://www.npmjs.com/package/@daypilot/daypilot-lite-vue>, 2023. 05. 02.