

Fejlesztői leírás

Az alábbi fejlesztői leírás a Buliszerviz névre keresztelt alkalmazás hardveres és szoftveres architektúráját hivatott bemutatni.

Tervezési fázis

Az alapvető elképzelésünk egy mágnesszenzoros ajtónyitó/csukó mechanizmusra épült volna. A megvalósításhoz szükséges alkatrészek viszont a jelenleg is tartó készlethiány következtében nem voltak elérhetőek záros határidőn belül. Így született meg az ötlet, hogy egy webalkalmazással működtetett, modern beléptetőrendszert készítsünk, ami egy buliba biztosítja a beléptetést, távolról.

Hardver/szoftver és a felhasználók kapcsolata

Egy ilyen, már akár komplexnek is nevezhető program esetén több kérdést is meg kellett vizsgálni:

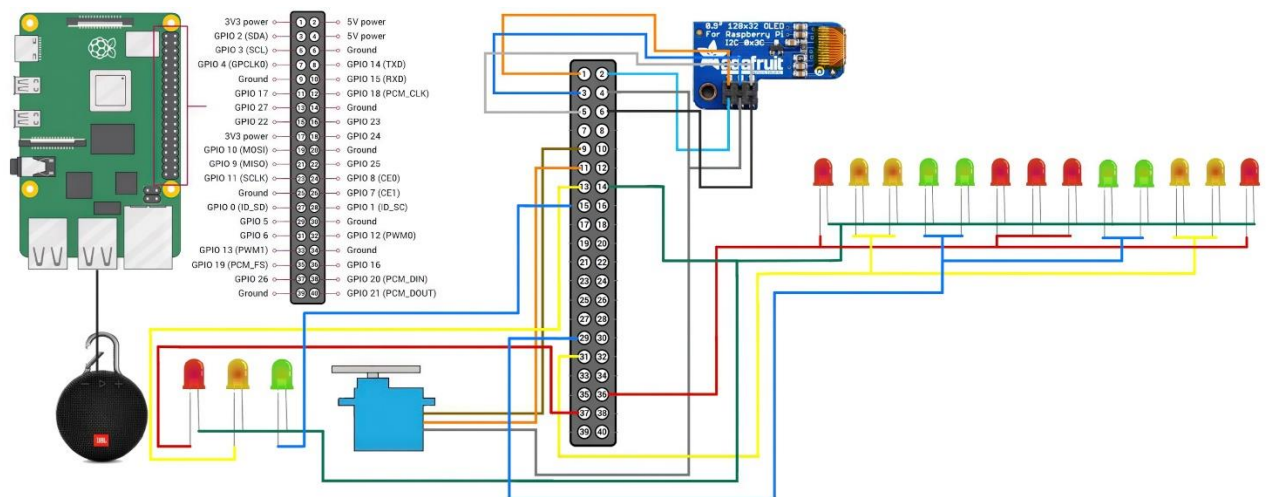
- hogy biztosítsuk megfelelő módon az oda-vissza kommunikációt a felhasználók, adminok és a webalkalmazás között?
- mi a legmegfelelőbb módja annak, hogy a mikrokontrollerhez csatlakoztatott eszközöket vezéreljük?
- milyen webes keretrendszert válasszunk?

Makett elkészítése

Célunk volt, hogy egy megfelelő minőségű makett készítsünk, mely otthont ad a különböző komponenseknek. Kikötés volt, hogy egy modern, letisztult és elegáns megvalósítás mellett döntsünk.

A tervezéstől a végleges változatig több iteráció készült.

Hardver



Hardver elemek

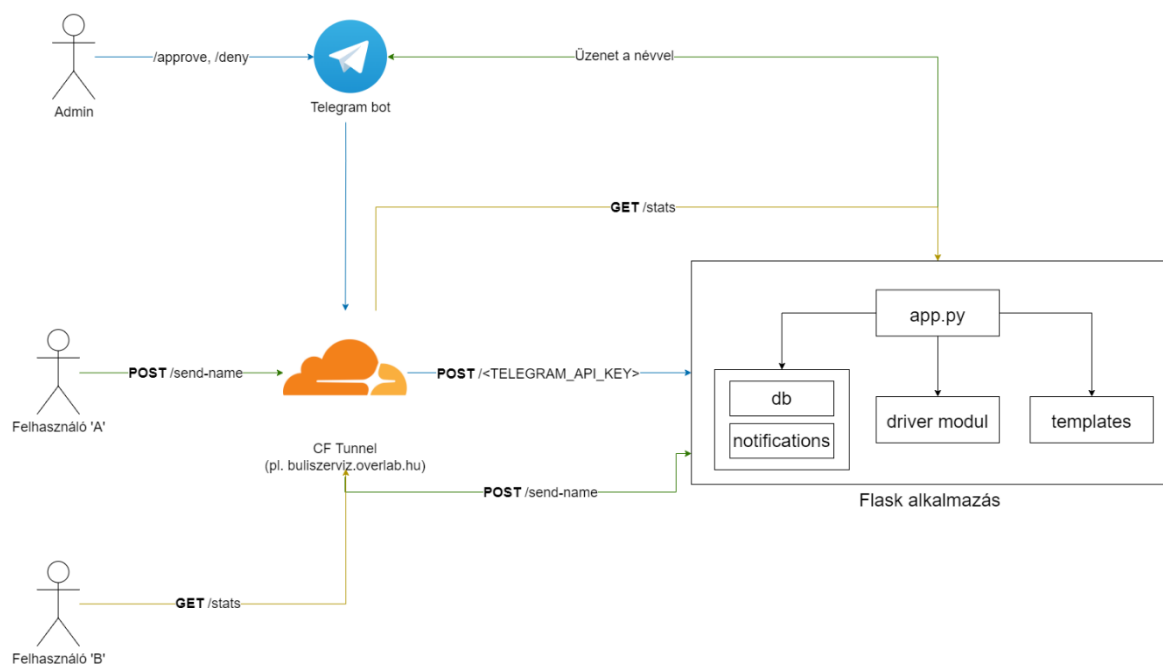
- Raspberry Pi 4 Model B 4GB
- Kingstone 16GB USB 2.0 pendrive
- Adafruit PiOLED - 128x32 Mini OLED
- LED csomag - 6x piros, 6x sárga, 6x zöld, 20x330R
- SG92R A169 Micro Servo - szervomotor
- 3 db - LED műanyag beépítő keret
- 1 db - Breadboard for RPI
- 1 db - JBL Clip 3
- 1 db - USB Type-C to USB-A
- 1 db - USB Micro B to USB-A
- Egyedi, méretre vágott és forrasztott patch kábelek
- Erős kétoldalú ragasztó

Makett előállításához szükséges alapanyagok és eszközök

- Ipari rack faalapanyag
- Körfűrész
- CNC router / flex
- Popszegecselő gép
- Sima szegecselő gép

- Alumínium huzal
- Fúrógép
- Csiszolópapír
- Forrasztópáka
- Lágy forrasztó cin
- Mini zsanér

Szoftver



Alkalmazásunk főmozgatórugója a Pythonban és JavaScriptben írt, a Flask nevű "mikrokeretrendszer" használó fullstack webalkalmazás.

A framework megengedő struktúrája könnyen lehetővé tette, hogy lerövidüljön a hardveres komponensek irányításáért felelős kód (továbbiakban: driver) integrációja. A könyvtár nagy hangsúlyt fektet a szervertoldali renderelésre, így nem annyira különül el a backend a frontentől.

Backend

Az appunk backendje felelős mind a hardveres elemekkel való kapcsolat létesítéséért, mind a webes (REST API) kérések kezeléséért.

A fejlesztési folyamatok egy ideig két külön szálon mozogtak, így egyszerre fejlesztettük a driver kódot és ismerkedtünk a keretrendszerben rejlő lehetőségekkel.

A **driver** modul a naiv megoldásokkal teletűzdelt, proof-of-concept fázisától hosszú út vezetett a Python konvenciókat használó, objektumorientált változatáig. A különböző hardverkomponensek további almoduloként jelennek meg, hisz így a modularizáció révén könnyebben menedzselhető kódbázist kapunk. A megvalósított függvények és metódusok implementációja során próbáltunk arra törekedni, hogy újrafelhasználhatóan írjuk meg őket. Így az igények és elvárások változása esetén se kell jelentős módosításokat eszközölni. Hatalmas terhet vett le a vállunkról, hogy szabványosított segédkönyvtárak lehetővé tették a kommunikációt, mind GPIO, mind I2C oldalon.

A főmodul `__init__.py` nevű fájljába deklaráltuk a **Driver** osztályt, melynek példányosítása után elérhetővé válnak a főbb, hardveroldali kéréseket kezelő metódusok, függvények.

Az **api** modulban találhatóak az alkalmazás azon rétegei, melyek valami külső erőforrás használatát teszik lehetővé.

A **notifications** modul tartalmazza a **TelegramBot** osztályt, mely példányosítva képes az üzenetküldési és -fogadási feladatok ellátására. A konstruktornak egy access tokenre és egy chatazonosítóra van szüksége, utána képes a **python-telegram-bot** könyvtár beépített funkcióira, típusaira támaszkodva kezelni a kommunikációt az alkalmazás és a Telegram csoport felhasználói között. Fontosabb pont volt, hogy támogassa a webhook-alapú (ld. Cloudflare Tunnels) üzenetfogadást, amely egy hatékonyabb módja az érkező válaszok feldolgozásának, mintha bizonyos időközönként kérdeznénk le a Telegram API-ján keresztül. Hasznosnak bizonyult még az ún. "válaszbillentyűzet" létrehozása is, amivel

gombok formájában tudjuk megjeleníteni a különböző Telegram kliensekben az elérhető parancsokat egy kérelem esetén.

A **database** modul neve magáért beszél, ezen egyszerű implementáción keresztül egy absztrakciót biztosítunk az **SQLite** adatbázisunkkal kapcsolatos műveletekhez. Így az endpointoknak már elég meghívni egy-egy függvényt, ha lekérdeznénk az eseményeket vagy esetleg hozzáadnának egy újat. Nagy előnye az SQLite adatbázisnak, hogy könnyen készíthető róla backup, tekintve, hogy egy **.db** kiterjesztésű fájl tartalmazza a benne lévő adatokat. Az osztály konstruktora kezeli azt az eshetőséget is, ha még nem lenne az átadott fájlnevével egy adatbázis sem. Ilyenkor legenerálja egy **schema.sql** fájl alapján a szükséges táblát, a megfelelő adattípusú oszlopokkal együtt.

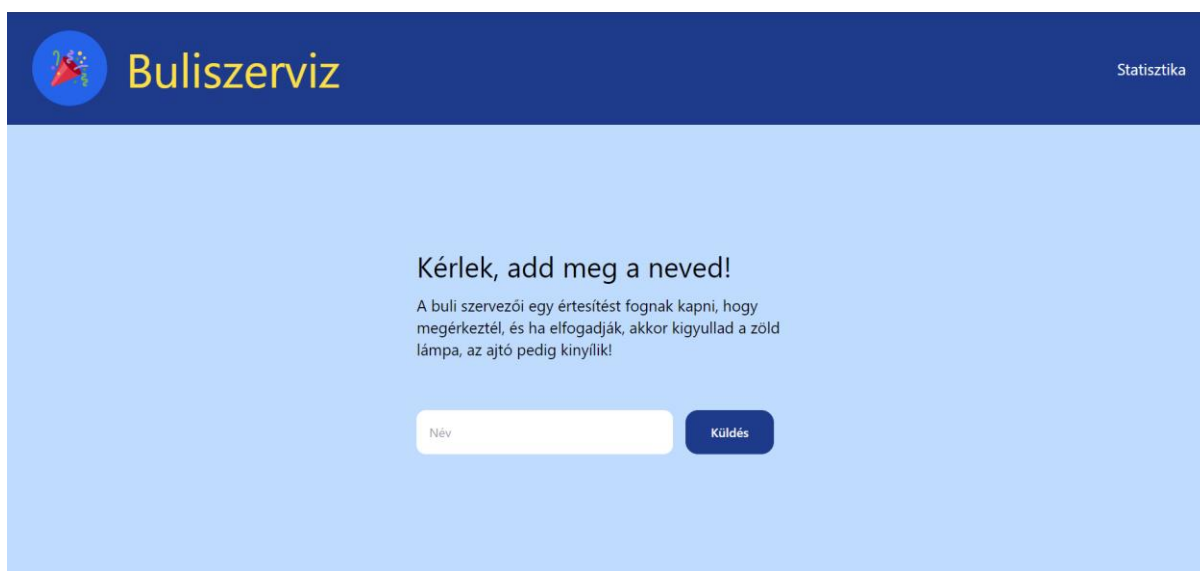
Érdemes még a "főattrakció" előtt megemlíteni a **utils** modulban található **constants.py**-t, ami az **app.py**-ban lévő visszatérő válaszértékeket tartalmazza enumok formájában.

Végül, de nem utolsó sorban: az **app.py** tartalmazza az összes olyan végpontot, ami lehetővé teszi, hogy mind a frontend, mind pedig a Telegram felől egy REST API-n keresztül tudjanak kommunikálni az alkalmazással. Ugyancsak itt kerülnek példányosításra a fentebb említett osztályok, melyek szükséges értékeit egy **.env** nevű fájlban tároljuk, mint szabványmegoldás. Az **app.py**-ban lévő endpointok meghívása esetén a korábban már tárgyalt modulok metódusainak hívását hangolja össze. Ilyen például, hogy fogadott üzenet esetén (ez esetünkben csak az elfogadási vagy az elutasítási parancs) lefuttatja a **driver** modul **on_approval()** vagy **on_denial()** metódusát, majd rögzíti az adatbázisunkban a megfelelő értékekkel az eseményeket. A kliensoldalon megjelenített grafikonhoz szükséges adatok normalizálása is itt, a **/api/stats** endpoint hívása esetén történik meg. Emellett még a Jinja template nyelv segítségével megírt HTML oldalak megjelenítésért is felel a backend.

Frontend

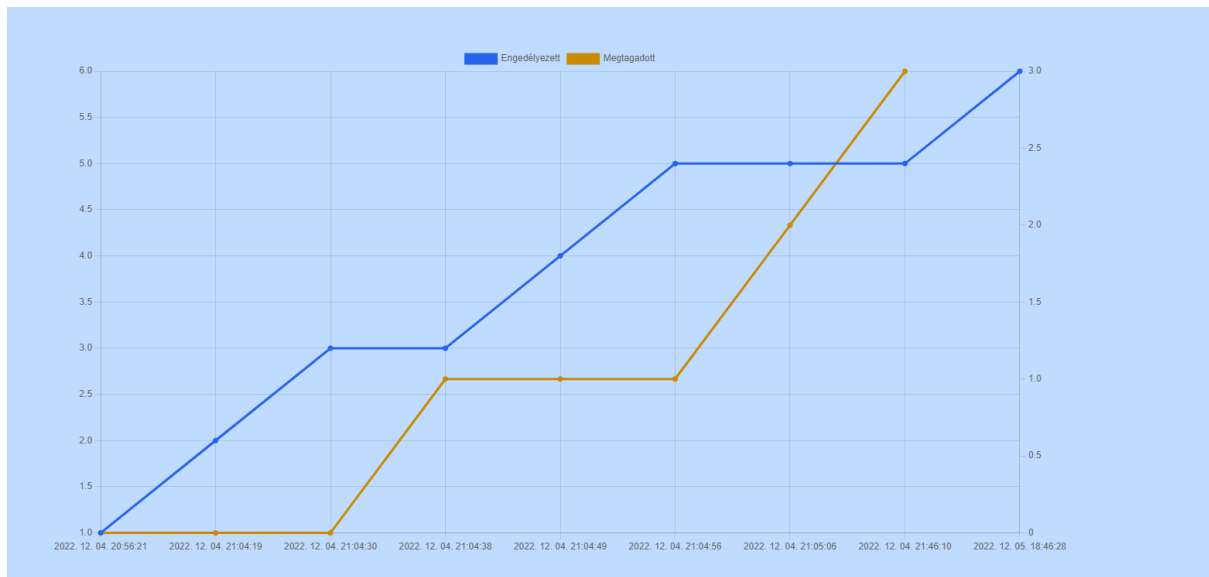
Kliensoldalon két aloldalról beszélhetünk: ez az `index.html (/)`, mely a főoldal, ahonnan a felhasználó be tudja küldeni a nevét, és `stats.html (/stats)`, mely a [Chart.js](#) nevű segédkönyvtár segítségével jeleníti meg a backendből érkező adatokat.

A két HTML oldal egy közös ősről osztozik, amely a `templates` könyvtárban `base.html` található fájl. Ez nyújtja a bázist a két oldal számára, hisz a `head` és a fejléckomponens megegyezik mindkét esetben. A két, igazi oldalon az `extends` parancs segítségével adjuk meg, mint szülő, és a `title` változó megadásával pedig az aktuális aloldal címe is hozzákerül a böngészőlapra megjelenített címhez.



The screenshot shows the Buliszerviz web application. The header is dark blue with a logo on the left and the text 'Buliszerviz' in yellow. On the right, 'Statistika' is written in small white text. The main content area is light blue. In the center, there is a text prompt 'Kérlek, add meg a neved!' followed by a smaller line of text: 'A buli szervezői egy értesítést fognak kapni, hogy megérkeztél, és ha elfogadják, akkor kigyullad a zöld lámpa, az ajtó pedig kinyílik!'. Below this text is a white input field with the placeholder 'Név' and a dark blue button labeled 'Küldés'.

A korábban említett oldalak esetén szükségünk volt az API hívások eléréséhez kliensoldali kódra is, így ES6 szabványokat kiaknázó JavaScript kódot írtunk. A beépített Fetch API-nak köszönhetően egyszerűen tudtunk kommunikálni a backend kódban definiált REST API-val, legyen az egy `GET` vagy egy `POST` kérés. A főoldal esetén `/api/send-name` endpointot használjuk a név beküldésére, mely eredményéről egy, a képernyő alján megjelenő ablakban tájékoztatjuk a felhasználót.



A statisztikai adatokat megjelenítő oldalunk esetén szükség volt a Chart.js segédkönyvtárra, amelyet egy CDN-en (Content Delivery Network) keresztül hivatkozunk be. Az oldal betöltése után, lekérdezzük `/api/stats` végpontot, amely "konyhakész" módon adja vissza a library által igényelt adatokat. Így képesek vagyunk megjeleníteni egy időalapú grafikont, amely megmutatja a kérelmek elbírálásának eredményét, az adott kérelmező nevével együtt.

A CSS esetén a [Tailwind CSS](#) nevű keretrendszerre bízunk magunkat, ami lehetővé teszi, hogy segédosztályok segítségével gyorsan személyre tudjuk szabni komponenseinket. Az effektív használatához szükséges volt az `npm` nevű, JavaScript alkalmazásokhoz használt csomagkezelő, amely az általa generált `package.json` fájlban tárolja az alkalmazás JS-alapú függőségeit. Pozitív hozadéka, hogy a CSS build folyamatához szükséges script definiálása mellett, összegyűjtöttük a teljes alkalmazás futásához szükséges parancsokat is, így egy `npm run <dev|prod>` futtatása esetén az egész rendszer életre kel.

Cloudflare Tunnels

A tervezés során először komoly fejtörést okozott az a követelmény a Telegram részéről, hogy webhook alapú kommunikáció esetén csak egy megfelelő tanúsítvánnyal rendelkező domain felé engedélyezett az adatküldés. Alapesetben ez portnyitással járna, és valamilyen webserver konfigurálásával, ami plusz erőforrást és terhet jelentett volna.

Ezután arra jutottunk, hogy a legegyszerűbb megoldás, ha a Cloudflare nevű cégóriás proxyszolgáltatását használjuk. Így a kommunikáció során a Cloudflare küldi/fogadja, majd továbbítja a be- és kiérkező adatokat. Az adott tunnel a host egy portján futó alkalmazáshoz van kötve. A szóban forgó tunnel a konfiguráció során megadott domain subdomainjeként (pl. <https://ez-az-en-tunnelem.pelda.hu> -> localhost:5000) fog élni, mindaddig, amíg meg nem szakítjuk a **cloudflared** nevű Debian-csomag futását.

Telegram

A Telegram üzenetküldő alkalmazásra azért esett a választás, mert ez találtuk a "legköltséghatékonyabb" módnak arra, hogy a kétoldali kommunikációt megvalósítsuk biztonságos módon. Emellett mindkettőnk számára ismerős volt már a kezelőfelülete, amely ugyancsak egy plusz volt.