Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Révész Gergő László

2022

Szegedi Tudományegyetem

Informatikai Intézet

Általános feladatütemező szolgáltatás

Szakdolgozat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Készítette: |  | Témavezető: |  |
|  | Révész Gergő László |  | Pénzes János |  |
|  | programtervező informatikus szakos hallgató |  | Üzletági igazgató |  |

Szeged

2022

## Feladatkiírás

A szakdolgozat keretein belül a hallgató egy általános feladat ütemező készít, amely segítségével időszakosan végrehajtandó funkciók futtathatók, előre definiált időközönként.

A hallgató szakdolgozatát a GriffSoft Informatikai Zrt.-nél készíti. A cég fő terméke a Forrás ügyviteli rendszer, amelyet főként Önkormányzatok, Költségvetési intézmények és középvállalatok használnak.

Az ügyviteli rendszer használata során többször felmerül az igény, hogy szükség van adott időközönként rendszer- vagy egyéb ügyviteli funkció futtatására. Ilyen lehet egy e-mail küldés vagy akár a NAV számára küldendő Online számla bevallás.

Ennek megoldására készít a hallgató egy központi ütemező alkalmazást, amelybe ezeket az időszakos futást igénylő funkciókat könnyen beilleszthetjük, egyedileg kezelhetjük, paraméterezhetjük.

Az ütemezőben definiált feladatok végrehajtásának figyelésére, egy böngésző alapú alkalmazás is fejlesztésre kerül, ami által egyszerűen nyomon követhetjük az aktív feladatok futási státuszát.

## Tartalmi összefoglaló

* ***A téma megnevezése:***

*Általános feladatütemező szolgáltatás*

* ***A megadott feladat megfogalmazása:***

*A feladat egy olyan alkalmazás készítése, amely dinamikusan, futásidőben tud Windows szolgáltatáshoz funkciókat hozzáadni és törölni, valamint egy felhasználói felület készítése ezen funkciók állapotainak megfigyelésére.*

* ***A megoldási mód:***

*A keretrendszer által nyújtott technológiák segítségével a program futásidőben tud változtatni saját szerkezetén. Ezt kihasználva készült egy újra indítást nem igénylő, de szerkezetileg folyamatosan változó alkalmazás, az ezzel való kommunikációra pedig egy web alapú felhasználói felület került fejlesztésre.*

* ***Alkalmazott eszközök, módszerek:***

*Az alkalmazás a Visual Studio nevű fejlesztői környezetben, C# nyelven készült.*

*A fejlesztés Windows szolgáltatás részéhez .NET, a felhasználói felület részéhez a Blazor keretrendszer nyújtott segítséget. Az alkalmazást több eszközön is tesztelve lett.*

* ***Elért eredmények:***

*Az alkalmazást a Griffsoft Informatikai Zrt használja.*

* ***Kulcsszavak:***

*Service, dll, AppDomain, aszinkron*

## Tartalomjegyzék

[Feladatkiírás 2](#_Toc99844576)

[Tartalmi összefoglaló 3](#_Toc99844577)

[Tartalomjegyzék 4](#_Toc99844578)

[Bevezetés 6](#_Toc99844579)

[1. Használt technológiák, eszközök bemutatása 7](#_Toc99844580)

[1.1. Programozási nyelv 7](#_Toc99844581)

[1.1.1. C# 7](#_Toc99844582)

[1.1.2. Egyéb nyelvek: 7](#_Toc99844583)

[1.2. Keretrendszer 8](#_Toc99844584)

[1.2.1. .NET 8](#_Toc99844585)

[1.3. Fejlesztői környezet 8](#_Toc99844586)

[1.3.1 Visual Studio 9](#_Toc99844587)

[2. A megvalósítás menete 10](#_Toc99844588)

[2.1 Windows Service 10](#_Toc99844589)

[2.1.1. Timer 11](#_Toc99844590)

[2.1.2 Task objektum, async és await kulcszó 12](#_Toc99844591)

[2.1.3 FileSystemWatcher 13](#_Toc99844592)

[2.1.4 Reflection 14](#_Toc99844593)

[2.1.5. AssemblyLoadContext 15](#_Toc99844594)

[2.1.6. NLog 16](#_Toc99844595)

[2.2. Blazor 17](#_Toc99844596)

[2.2.1. Service Controller 17](#_Toc99844597)

[2.2.2. Toastr 18](#_Toc99844598)

[2.2.3. TCP Listener és TCP Client 19](#_Toc99844599)

[5. A kész alkalmazás 21](#_Toc99844600)

[5.1. Funkciók tesztelése 21](#_Toc99844601)

[5.1.1. Indítás funkció 21](#_Toc99844602)

[5.1.2. Betöltés funkció 22](#_Toc99844603)

[5.1.4. Frissítés funkció 23](#_Toc99844604)

[5.1.5. Leállítás funkció 23](#_Toc99844605)

[5.2. További fejlesztési lehetőségek 23](#_Toc99844606)

[Irodalomjegyzék 24](#_Toc99844607)

[Nyilatkozat 25](#_Toc99844608)

## Bevezetés

A Griffsoft Informatikai Zrt. fő terméke a Forrás ügyviteli rendszer, amit Önkormányzatok, Költségvetési intézmények és középvállalatok is használnak adminisztrációs célokra. A Forrás használata közben számos alkalommal felmerül az igény bizonyos funkciók előre meghatározott időközönként történő lefuttatására. Az ilyen funkciók pl. egy e-mail küldés vagy a NAV számára küldendő Online számla bevallás.

Az ellátandó feladatok megnövekedett száma miatt a dolgozók munkáját megkönnyítené, ha ezeket egy számítógépes program végezné el automatikusan. Ezen probléma megoldása miatt esett a választásom egy olyan alkalmazás elkészítésére, ami ennek a feladatoknak eleget tesz.

A szakdolgozatom célja egy olyan Windows operációs rendszeren futó szolgáltatás implementálása, amit mindamellett, hogy menedzseli a szükséges funkciók futtatását, lehetőséget nyújt újabb feladatok hozzáadására és meglévők törlésére egyaránt anélkül, hogy meg kellene szakítani a program futását.

Az alkalmazás egy háttérben futó szolgáltatás, amihez egy Blazor webalkalmazást is készítek a felhasználói interakciók kezelésére.

A kész alkalmazás egy könnyen használható Windows szolgáltatás, aminek továbbfejlesztése kevés programozói tudást igényel. A dinamikus megvalósításnak köszönhetően bármennyi ideig képes futni nagyon kevés erőforrás használatával, és tudja kezelni a funkciók futtatása közben fellépő esetleges hibákat.

//TODO bővíteni

# 1. Használt technológiák, eszközök bemutatása

## 1.1. Programozási nyelv

Egy mai számítógép az emberek számára követhetetlen számolási sebességgel bír, és ez miatt jó ötletnek tűnhet, ha az egyes emberi feladatokat rá tudnánk bízni egy számítógépre. Ezek „agya” egyelőre még nem ért emberi nyelven, de akkor mégis hogyan tudnánk megmondani egy számítógépnek, hogy mit csináljon?

Ennek a problémának a megoldására születtek meg a programozási nyelvek. A programozási nyelvek mindegyike, csak úgy, mint egy emberi nyelv, rendelkezik saját szabályrendszerrel. Egy szabályrendszer lehetőséget ad az emberek számára egységes módon megfogalmazni egy utasítást, vagy annak sorozatát, egy programot. Egy program a programozási nyelv szabályait követve az más, a szabályokat szintén ismerő ember számára is olvasható és érthető.

### C#

Egyes problémák megoldására bizonyos programozási nyelven megírt programok előnyt jelenthetnek, ezért annak megválasztása fontos. A Microsoft által kifejlesztett C# nyelv egy objektumorientált programozási nyelv, aminek alapja a C++ és a Java. C#-ban több platformra is tudunk alkalmazást készíteni, például mobilra és webre. A szakdolgozat 80%-a ebben a nyelvben íródott.

### Egyéb nyelvek:

A JavaScript egy interpretált programozási nyelv. A böngészővel való kommunikációs képessége miatt webprogramozásban ez a leggyakrabban használt nyelv, mivel a böngésző ezt a nyelvet értelmezni tudja, és segítségével például egy weboldal DOM-ja is dinamikusan megváltoztatható.

A HTML (Hipertext Markup Language) egy leíró nyelv, ami előre definiált tag-ek egymásba ágyazásával teszi lehetővé a weboldal vázának felépítését. Frontend-nek a legtöbb webfejlesztési keretrendszer, így a Blazor is HTML-t használ.

A CSS (Cascading Style Sheet) egy stílusleíró nyelv, ami előre definiált kulcs-érték párok megadásával tudja változtatni a strukturált dokumentumok, például a HTML megjelenését.

## 1.2. Keretrendszer

Egy nagy projekt megírása sok profi programozónak is rengeteg időbe telik, azonban léteznek módszerek, amikkel a fejlesztés idő lerövidíthető. Ebben tudnak segítséget nyújtani a keretrendszerek. Ezek általános eszköz tarát biztosítanak, amik segítségével a fejlesztőnek nincs szükség mindent az elejéről felépíteni. Használatukkal a már működő rendszer ki lesz terjesztve munkánkkal így a lényegi részre, a funkciók implementálására fókuszálhatunk, nagyban lerövidítve ezzel a projekt fejlesztésének idejét. Elsajátításukkal javítja, szükségtelen használatuk viszont csak ront a program performanciáját.

### 1.2.1. .NET

A szakdolgozatom elkészítéséhez a .NET keretrendszert használtam. Ez egy ingyenes, nyílt forráskódú szoftver, ami több platformon is képes futtatni a C#-ban megírt forráskódunkat, továbbá rengetek hasznos NuGet csomag használatára is lehetőséget ad, amikkel más munkája egyszerűen használhatóvá válik programunkban.

## 1.3. Fejlesztői környezet

A fejlesztői környezetek olyan programozók számára készült szoftverek, amik a fordítót, a futtatókörnyezetet és szövegszerkesztőt is tartalmaznak annak érdekében, hogy a programozónak a rendelkezésére álljon minden egy helyen. Egy ilyen fejlesztői környezet nagyon sokat tud gyorsítani egy program fejlesztésének idején.

Léteznek azonban olyan fejlesztői környezetek is, amik mindamellett, hogy tudják ezek mindegyikét, még adott programozási nyelvek formai szabályait is „ismerik”, ami miatt még több segítséget tudnak nyújtani. Képesek például a programkód gépelése közben vétett hibákat jelezni a fejlesztő számára, felugró listával segíteni befejezni a nyelv kulcs szavait, és képes velük a fejlesztő a program futása közben belelátni a folyamat memóriájának bizonyos részeibe. Ezeket a fejlesztői környezeteket integrált fejlesztői környezeteknek hívják, és manapság már a leggyakrabban használt programozási nyelvek mindegyikéhez létezik ilyen.

### 1.3.1 Visual Studio

A C# nyelvhez a Visual Studio nevű integrált fejlesztői környezet a legnépszerűbb. A Microsoft által fejlesztett szoftver a programkódunk megírásának segítésén kívül is rengetek funkciót tartalmaz, amik többnyire a kényelmünket szolgálják, és élvezhetőbbé teszik a fejlesztést.

# 2. A megvalósítás menete

## 2.1 Windows Service

Egy hagyományos alkalmazás általában rendelkezik felhasználói felülettel annak érdekében, hogy a felhasználók interakcióba tudjanak lépni az alkalmazással, kommunikálni tudjanak vele és közölni tudják szándékukat a programmal. Jó példa erre egy számítógépes játék, aminek elindulásakor megjelenik az ikonja a tálcán, a bejelentkezett felhasználónak tudomására van hozva, hogy fut-e az alkalmazás, amit elindított, és tud-e játszani vele.

Egy Windows Service alkalmazás ezekből a szempontokból különbözik egy hagyományos alkalmazástól. Ilyen alkalmazásokat általában azért készítünk, hogy bizonyos funkciókat kezeljenek hosszú időn keresztül. Ezeknek a funkcióknak nincs szükségük felhasználói interakcióra, csak futniuk kell, és tenniük a dolgukat, emiatt alapértelmezetten nincs is hozzájuk felület, és a felhasználó sok esetben nem is tud futásukról. Debug-olására csak úgy, mint fejlesztése közben, éles futáskor is van lehetőség egy debugger hozzácsatolásával. Egy Service alkalmazásból egy számítógépen maximum egy példány fut még akkor is, ha felhasználó váltás történik. Mivel az alkalmazás nem egy bejelentkezett felhasználóhoz, hanem a számítógéphez van kötve, így általában boot-olás után már el is indulnak. Szüneteltetésükre és megállításukra is van lehetőség, de csak megfelelő felhasználói jogosultsággal.

A szakdolgozat egy megállás nélkül sok ideig futó, a feladatok futtatásáért felelős alkalmazás, aminek motorja egy háttérben futó Windows Service program. Ilyen alkalmazást a Visual Studio nevű fejlesztői környezetben könnyedén tudunk készíteni. Ehhez egy Worker Service típusú projektet hozunk létre, majd a projekt file-ban beállítjuk, hogy a fordítás során ne dll file-ok, hanem egy futtatható exe kiterjesztésű file készüljön.



Az alapértelmezett Host builder-rel build-eljük az IHost típusú objektumot, majd futtatjuk.



Az így elkészült futtatható file-t a Windows szolgáltatásai közé az sc.exe parancs felparaméterezésével és kiadásával tudjuk beregisztrálni, és futtatni.

### 2.1.1. Timer

A .NET keretrendszerben több lehetőségünk is van időzítő használatára. Ezek mindegyike egy jól, de másképpen implementált osztály. A dokumentációjának alapos tanulmányozásával eldönthetjük, hogy melyikre van szükségünk.

A System.Timers névtérben található a Timer osztály egy szerver alapú, több szálas környezetben működő időzítő használatára ad lehetőséget. Segítségével a Windows időzítőnél is pontosabban tudjuk a kívánt metódusok futtatásának idejét időzíteni. Amikor a megadott millimásodperc letelik, a hozzárendelt eseménykezelő meghívódik, a több szálas környezetnek köszönhetően pedig az óra fut tovább.

A Timer objektum példányosítása, és property-jeinek beállítása után ezt az objektumot használjuk arra, hogy ütemezze a feladatok lefuttatásának idejét.



Az egy másodpercenként lefutó eseménykezelő minden lefutáskor eggyel inkrementálja az eltelt másodpercek követésére használt változó értékét.



Ha a másodpercszámláló és a feladatban definiált időzítő értéke alapján elindítható a feladat, akkor Reflection segítségével lefuttatjuk.



### 2.1.2 Task objektum, async és await kulcszó

A System.Threading.Task névtérben található Task osztály egy műveletet reprezentál, aminek végrehajtása előtt általában aszinkron módon felsorakozik a thread pool-on, majd sorra kerülés után egy szálon lefut. Hasznos lehet, ha olyan műveletet szeretnénk végrehajtani, ami sok időbe telik, várakoztatja és blokkolja a fő szálat. Ilyenkor a fő szál átadja egy másik szabad szálnak a feladatot, ami majd helyette elvégzi azt.

Az await kulcsszó segítségével tudjuk megvárni az előbb említett Task objektum sorra kerülését és lefutását. Hasznos lehet, ha például a felhasználó várakozásának ideje alatt nem a lefagyott, hanem például egy töltő képernyőt szeretnénk megjeleníteni. Az e féle várakozás ezt lehetővé teszi, hiszen a fő szálat tehermentesíti az elvégzendő feladat alól, így annak erőforrása használható marad másra. Az async kulcsszó teszi lehetővé, hogy az ezzel megjelölt metódusok belsejében használható legyen az await kulcszó, ezért az async és await kulcsszavak általában együtt használatosak.

Annak érdekében, hogy a futtatás, a hozzáadás és a törlés események ne zavarják, ne blokkolják egymást, és hogy egy időben több művelet is elvégzésre tudjon kerülni, az alkalmazásban aszinkron programozásra, és Task objektumok használatára van szükség, ezért ahol csak lehet és van is értelme, async Task visszatérési értékkel definiált metódusokat használjunk, amiknek megvárásáról is gondoskodni kell egészen a CallStack aljáig.



### 2.1.3 FileSystemWatcher

A .NET Core System.IO névterében található FileSystemWatcher egy keretrendszer által biztosított osztály, ami képes egy könyvtárban, és annak alkönyvtáraiban észrevenni, ha egy megadott mintára illeszkedő névvel rendelkező file-lal változás történt, és lefuttatni az eseményhez hozzárendelt függvényt.

Ezt az osztályt használhatjuk arra, hogy futás közben file-ok hozzáadásakor vagy törlésekor elindítsuk a megfelelő eseménykezelőket. Az osztály konstruktorába először is injektáljuk az alkalmazás paramétereit, amikkel létrehozzuk az objektumot. Ezek a paraméterek az alkalmazás exe file-jának mappájában található appsettings.json file-ban vannak definiálva, amit a Service felületről való indítása során tudjuk módosítani.



Ez után a hozzáadás és a törlés delegate-ekhez hozzáadjuk az eseménykezelőket, amik a dll-ek betöltését és kitörlését végzik. Ezek aszinkron metódusok, amik void a visszatérési értékkel rendelkeznek. Ez azért fontos, mert ezek is aszinkron módon futnak le csakúgy, mint a Task-ok, viszont az ilyen típusú függvényekből kidobódott kivételeket nem tudjuk elkapni, hiszen nem await-elhető, és Task objektum sincsen, amivel a hiba visszajöhetne a CallStack-en.



### 2.1.4 Reflection

A Reflection egy keretrendszer által nyújtott technológia. Segítségével dinamikusan tudunk assembly-ket és típusokat kezelni, ezekről futás idejű információt szerezni és példányosítani őket. A példányosításkor létrejött objektumnak meghívhatóvá válnak a metódusai, és az adattagjait is elérjük.

Ezt kihasználva tudunk egy teljes mértékben ismeretlen dll-ről strukturális ellenőrzéseket végezni, majd ezek alapján megállapítani, hogy megfelel-e a követelményeinknek, példányosítható és futtatható-e. Az ismeretlen assembly-k betöltését Reflection segítségével tudjuk megvalósítani, amit megelőz néhány lépés. Először ellenőrizzük, hogy létezik-e a példányosítandó osztály, és hogy pontosan egy van belőle.



Ez után megnézzük, hogy implementálja-e a kötelező interface-t.



Ha eddig megfelelt a követelményeknek, megpróbálhatjuk példányosítani és elérni az interface által megkövetelt Run metódust, illetve Timer property-t.



Végül ellenőrizzük, hogy sikerült-e minden, és hogy az ütemező elkezdheti-e futtatni a Run metódust.



### 2.1.5. AssemblyLoadContext

.NET Core-ban megszűnt a lehetőség egy folyamaton belül több AppDomain létrehozására, ami helyett az AssemblyLoadContext-et vezették be.

Ez az osztály a System.Runtime.Loader névtérben található. Használatával assembly-ket tudunk betölteni, amik egy összefüggő kollekcióban, az objektum Assemblies property-jében el lesznek tárolva. Előnye az Assembly osztály statikus Load metódusával szemben, hogy miután elengedtük az összes assembly objektumra mutató referenciát, az Unload metódus meghívásával az AppDomain futás közben is el képes engedni dll file-okat.

Ezt kihasználva nem csak a dinamikus dll betöltést, hanem annak dinamikus elengedését is meg tudjuk valósítani. Ehhez először a FileSystemWatcher által detektált zip file-t kicsomagoljuk, és a keletkezett mappa szerkezetben megkeressük az entry dll-t, aminek a neve meg kell, hogy egyezzen a tömörített mappa kiterjesztés nélküli nevével. Találat után példányosítjuk az AssemblyLoadContext típusú objektumot.



A megtalált file-t az összes referenciájával együtt betöltjük a context-be. Ezt rekurzívan, a referencia dll-ek megkeresésével, tesszük, az alapértelmezett .NET 6 dll referenciákat kihagyva.



A kitörléshez az assembly-k és az azok által használt objektumok referenciáját a null értékre állítjuk azért, hogy a Garbage Collector, ami a nem használt memóriaterületek felszabadításáért felelős, el tudja takarítani az objektumokat, annak érdekében, hogy az Unload metódus el tudja engedni a dll-eket.



### 2.1.6. NLog

Egy alkalmazásnak nagyon fontos részét képezi a naplózás. Segítségével információt tudunk szerezni az alkalmazásunkban futás közben történt eseményekről, hibákról. Ezeket visszamenőlegesen is meg tudjuk tekinteni, ami jól jöhet például egy hiba okának kiderítésére.

Naplózni sok helyre tudunk, például file-ba, adatbázisba, konzolra, és vannak platformok, amik ezt rugalmasan, akár futás közben kezelni tudják. Egy ilyen NuGet csomag például az NLog, ami egy config file-ban konfigurálható, és programban egyszerűen használható.





a naplózási célpont megadása, majd indításkor az IHostBuilder beállítása után injektálással már használható logger.

## 2.2. Blazor

A szakdolgozatom Windows szervíz részéhez készítettem egy webes felületet is annak érdekében, hogy a felhasználók kezelni tudják, és nyomon tudják követni a szervízben futó feladatok státuszát.

Manapság számos keretrendszer létezik annak érdekében, hogy megkönnyítsék egy webalkalmazás elkészítését. Ezek segítségével már nem csak JavaScript nyelven van lehetőségünk a weboldalunk háttér logikáját megírni. Gondoskodnak arról, hogy a JavaScript akár teljes mértékben való elkerülése mellett is hibátlanul tudjanak működni weboldalaink.

A Blazor egy olyan webes keretrendszer, amivel C#, HTML és CSS segítségével felépíthető egy webalkalmazás. Futtatására két féle lehetőség áll a rendelkezésünkre. Az egyik lehetőség a Blazor Server. Ezt használva a kliens SignalR kapcsolaton keresztül kommunikál a szerverrel, a C# kód a szerver gépen fut, ami a megfelelő DOM változtatásokat visszaküldi a kliensnek.

A másik lehetőség a Blazor WebAssembly. Ezzel a technológiával a C# kód a .NET Runtime-mal együtt letöltődik a felhasználó böngészőjébe, és az alkalmazás teljes mértékben ott fut, amit a saját számítógépe futtat. Ebben az esetben a C# program korlátozva van a böngésző képességeire, nem fér hozzá a Windows szolgáltatások listájához, hiszen nem is biztos, hogy a böngésző Windows-on fut.

Mivel a felület elsődleges célja, hogy elérje és bizonyos mértékben kezelje a Windows Service alkalmazást, a WebAssembly korlátozásai miatt a Blazor Server technológiát választottam, amivel a szerver gépen futó Windows Service alkalmazás elérhetővé válik.

### 2.2.1. Service Controller

A .NET System.ServiceProcess névtérben található ServiceController osztály a Windows operációs rendszeren található Windows szolgáltatások kezelésére szolgál. Segítségével kapcsolódni tudunk a Service-ekhez, elindítani, szüneteltetni és leállítani is tudjuk azokat.

Használatához szükség van a System.ServiceProcess.ServiceController NuGet csomagra. Mivel ebben az esetben egy konkrét, a szerver számítógépen található Windows Service-t szeretnék kezelni, példányosításához elég egy paraméter, a szolgáltatás neve, ami alapján kikeresi a szolgáltatások listából a megfelelőt.



Ezután kezelhetővé válik a szolgáltatás, és a Start, Stop és Pause metódusokat használva elindíthatjuk a kívánt műveletet.



Még a fejlesztői környezet sem figyelmeztet rá, hogy az utóbbi műveletek hibát dobhatnak, ha nem sikerül elindítani az alkalmazást, mert például nem szerepel a listában.

### 2.2.2. Toastr

A felhasználói interakció hatására a háttérben általában eseményhez hozzárendelt metódusok, függvények futnak le, amik megpróbálják elvégezni a kívánt műveleteket. Jól jöhet, ha az eredmény sikerességét a felhasználó tudatára hozzuk. Ezt sokféleképpen tehetjük meg.

Ennek egy egyszerű módját, egy Alerter könyvtár használatát választottam, a Toastr-t. Egy CSS és egy script file weboldalra való betöltése után rendelkezésünkre fognak állni JavaScript függvények, amik lefuttatása figyelemfelkeltő előugró dialógusokat eredményeznek.

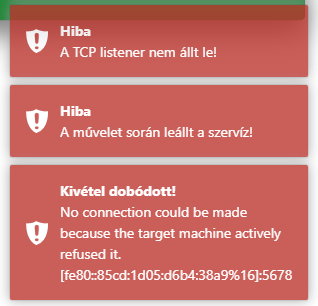
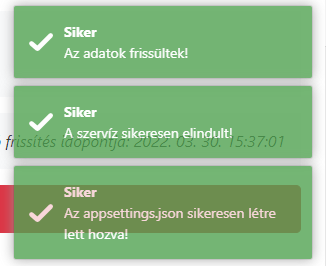


A Blazor a C# kód mellett nem zárja ki a JavaScript kód futtatásának lehetőségét sem. Az IJSRuntime kiterjesztésével és injektálásával futás közben C# kódból futtatni tudunk saját JavaScript kódot is, amit a keretrendszer gond nélkül kezel.









### 2.2.3. TCP Listener és TCP Client

A .NET System.Net.Sockets névterében található a TCP Listener és TCP Client osztályok segítségével TCP kapcsolaton keresztül tud kommunikálni egymással a listener és a kliens.

A listener egyféle szerverként viselkedik. Egy beállított IP címen és porton várakozik a bejövő TCP kapcsolatra, a kapcsolatból kiolvassa az üzenetet, ugyan ezen a kapcsolaton keresztül válaszol rá, majd bontja a kapcsolatot. A kapcsolatot a kliens indítja. Csatlakozik arra az IP címre és portra, amelyiken a listener hallgat, az üzenetet egy bájt tömbbe csomagolva átküldi a TCP kapcsolaton keresztül, majd megvárja a választ.

Ezt az osztályt használtam arra, hogy a webes felület le tudja kérdezni a Service-ben futó feladatok státuszát. Ehhez először a Service alkalmazásban elindul a listener, és egy ciklusban várakozik a kliensekre.



A kliens csatlakozik a listener-hez, és a kapcsolaton keresztül átküldi az üzenetet egy bájt tömbben.



Ezután a listener lekéri kapcsolatot és kiszedi belőle a bájt tömböt, ami az üzenetet tartalmazza, majd ennek feldolgozása után visszaküldi a választ, és lezárja a kapcsolatot.



Végül a kliens megkapja a választ, és lezárja a kapcsolatot.



# 

# 5. A kész alkalmazás

## 5.1. Funkciók tesztelése

Egy alkalmazás tesztelésének fázisa nagyon fontos szerepet játszik a fejlesztés során. Ilyenkor végig próbáljuk az alkalmazásban megvalósított funkciókat, és megkeressük majd kijavítjuk a fejlesztés során vétett esetleges hibákat, amik kezeletlen eseteket idéznének elő, ha egy felhasználó bukkanna rájuk.

Egy Windows szolgáltatás futásának nyomon követéséhez konzol és alapértelmezett felhasználói felület nem áll rendelkezésünkre, így a naplózás tűnhet a legegyszerűbb megoldásnak. A korábban már említett NLog használatával kiírattathatjuk az metódusok futásának státuszát. Mivel az alkalmazás futtatás indításának belépési pontja egy try – catch-ágba van rakva, ezért biztosak lehetünk benne, hogy az elkapható, de kezeletlen hibák az alkalmazás leállása előtt el lesznek kapva, és naplózva lesznek.

### 5.1.1. Indítás funkció

A webes felület indításakor a következő kép fogad minket.

//TODO KÉP

A felhasználói felületnek két feladata van az indítás gomb megnyomásával. Az egyik, hogy a szolgáltatás exe kiterjesztésű file-jának mappájába elkészítsen egy json kiterjesztésű file-t, ami a form-ban megadott paramétereket tartalmazza. Ezt a file-t fogja induláskor a szerviz feldolgozni, majd a megfelelő kulcs értékeit felhasználni paraméterként.

//TODO KÉP A FORM PARAMÉTEREIRŐL

//TODO KÉP A JSON SIKERESSÉGÉRŐL: TARTALMÁRÓL

A másik feladata, hogy indítsa el a szolgáltatást. A szolgáltatás állapotát a Windows operációs rendszer Szolgáltatások alkalmazásában megtekinthetjük.

//TODO KÉP ARRÓL, HOGY FUT

A szolgáltatás sikeres elindulása esetén a felület átvált a futási képernyőre, illetve a TCP Listener indulása is naplózásra kerül.

//TODO MÁSIK SCREEN

//TODO LISTENER ELINDULT

### 5.1.2. Betöltés funkció

A betöltés gomb megnyomása hatására egy file választó ablak ugrik elő, ami a betöltendő zip kiterjesztésű file-okat várja kiválasztásra.

//TODO FILECHOOSER KÉP

A kiválasztás után a betöltés folyamata részletes naplózásra kerül. Sikeres betöltés esetén a hibaüzenet nélküli adatokat, valamint a futtatáshoz szükséges mappákat láthatjuk.

//TODO SIKERES LOG

//TODO KÉP ZIP KICSOMAGOLVA ÉS BEMÁSOLVA

Sikertelen betöltés esetén a részletes hibák naplózásra kerülnek.

//TODO SIKERTELEN LOG

5.1.3. Futtatás funkció

Sikeres betöltés után a betöltött projekt megkezdi futását. A sikeres futtatás például egy olyan projekttel tesztelhetjük, ami 4 másodpercenként beleírja egy szöveges file-ba, hogy sikeresen lefutott.

//TODO KÉP A SIKERES RUN-RÓL

Ha a futás közben egy hiba jön vissza, az el lesz kapva és naplózva lesz.

//TODO KÉP A HIBÁS RUN-RÓL

### 5.1.4. Frissítés funkció

A sikeresen betöltött projektek felületen való megjelenítéséhez a frissítés gomb megnyomására van szükség. Ilyenkor TCP kapcsolaton keresztül megérkezik a service-ben aktuálisan futó projektek nevei, amik kikerülnek a képernyőre.

//TODO KÉP A FUTÓ PROJEKTEKRŐL A FELÜLETEN

A gomb megnyomására az utolsó frissítés időpontjának ideje is aktualizálódik, így a felhasználó tudhatja, hogy az éppen megjelenített adat mennyire aktuális.

//TODO UTOLSÓ FRISSITÉS IDŐPONTJA

### 5.1.5. Leállítás funkció

A felületen található leállítás gomb csak úgy, mint az indításé, 2 feladatot végez el. Az egyik, hogy a TCP Listener leállítását kérvényezi egy TCP kapcsolaton keresztül. A Listener sikeres leállítása naplózásra kerül.

//TODO LISTENER LEALLAS KEP

A gomb másik feladata, hogy állítsa le a szolgáltatást, aminek eredményéről ezúttal is a Szolgáltatások alkalmazásban győződhetünk meg.

//TODO KEP ARROL H LEALLT

Sikeres leállítás esetén a felület visszavált az indítási képernyőre, és a folyamat megismételhető.

## 5.2. További fejlesztési lehetőségek

//TODO interface implementálásával barmit betölthetünk és futtathatunk

# Irodalomjegyzék

# Nyilatkozat

Alulírott Révész Gergő László, Programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, Programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2022. január 17.

aláírás