Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Révész Gergő László

2022

Szegedi Tudományegyetem

Informatikai Intézet

Általános feladatütemező szolgáltatás

Szakdolgozat

Készítette:
Révész Gergő László
programtervező
informatikus szakos
hallgató

Témavezető:
Pénzes János
üzletági igazgató

Szeged

2022

Feladatkiírás

A szakdolgozat keretein belül a hallgató egy általános feladat ütemezőt készít, amely segítségével időszakosan végrehajtandó feladatok 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 futtatást igénylő feladatokat könnyen beilleszthetjük, egyedileg kezelhetjük, paraméterezhetjük.

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

Tartalmi összefoglaló

- ***A téma megnevezése:***

Általános feladatütemező szolgáltatás

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

A feladat egy olyan alkalmazás fejlesztése, amely dinamikusan, futásidőben tud Windows szolgáltatáshoz feladatokat hozzáadni és törölni, valamint egy felhasználói felület készítése ezen feladatok állapotának 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 felület került fejlesztésre.

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

Az alkalmazás a Visual Studio nevű fejlesztői környezetben, C# nyelven készült.
A fejlesztés Windows szolgáltatás részéhez a .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:***

dinamikus, dll, feladat, aszinkron

Tartalomjegyzék

Bevezetés

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

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

A szakdolgozat célja egy olyan Windows operációs rendszeren futó szolgáltatás implementálása, amit mindamellett, hogy menedzseli a szükséges feladatok futtatását, lehetőséget nyújt újak hozzáadására és meglévők törlésére egyaránt anélkül, hogy meg kellene szakítani a program futását. Az alkalmazás egy háttérben futó Windows szolgáltatás, amihez egy Blazor webalkalmazás is készül 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 minimális erőforrás használatával, és kezelni tudja a feladatok futtatása közben fellépő esetleges hibákat.

1. Használt technológiák, eszközök bemutatása

1.1. Programozási nyelv

Egy mai számítógép az emberek számára követhetetlen számolási sebességgel bír, ami miatt jó ötletnek tűnhet, ha az egyes emberi feladatokat rábíznánk a számítógépekre. 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. A szabályrendszer lehetőséget ad 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.

1.1.1. 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 ezen a nyelven íródott.

1.1.2. Egyéb nyelvek

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 JavaScript egy interpretált programozási nyelv. A böngészővel és a weboldallal való kommunikációs képessége miatt webprogramozásban ez a leggyakrabban használt nyelv. Használatával a HTML DOM-ja is dinamikusan megváltoztatható.

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 fejlesztési idő lerövidíthető. Ebben tudnak segítséget nyújtani a keretrendszerek. Ezek általános eszköz tarát biztosítanak, amik segítségével a fejlesztőnek nincs szüksége 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 performanciáján.

1.2.1. .NET

A szakdolgozat elkészítéséhez a .NET keretrendszer [1] került használatra. Ez egy ingyenes, nyílt forráskódú szoftver, amelyet a Microsoft már több mint 20 éve fejleszt. A keretrendszer .NET 6-os verziója 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 fordítót, futtatókörnyezetet és szövegszerkesztőt is tartalmaznak annak érdekében, hogy a programozónak a rendelkezésére álljon minden egy helyen. Egy ilyen fejlesztői környezet egy program fejlesztésének idején nagyon sokat tud gyorsítani.

Léteznek olyan fejlesztői környezetek is, amik mindemellett adott programozási nyelvek formai szabályait is „ismerik”, ami miatt még több segítséget nyújtanak. 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 nevezzük, és manapság már a leggyakrabban használt programozási nyelvek mindegyikéhez létezik ilyen.

1.3.1. Visual Studio

A C# nyelvhez a Visual Studio [2] 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, könnyedebbé 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 indulása és futása a felhasználó tudomására van hozva.

Egy Windows Service alkalmazás [3] 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 futásukról a felhasználó sok esetben nem is tud. 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, feladatok kezeléséért felelős alkalmazás, aminek motorja egy háttérben futó Windows szolgáltatás. Ilyen alkalmazást a Visual Studio nevű fejlesztői környezetben is tudunk készíteni. Ehhez egy Worker Service típusú projektet hozunk létre, aminek a projekt fájlában beállítjuk, hogy a fordítás során ne dll fájlok, hanem egy futtatható exe kiterjesztésű fájl készüljön.

```
...
<PropertyGroup>
      <OutputType>exe</OutputType>
      <PublishSingleFile>true</PublishSingleFile>
      ...
```

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

```
...
IHost host = Host.CreateDefaultBuilder(args)
...
.ConfigureServices((context, services) =>
{
      services.AddHostedService<Motor>();
      ...
.Build();
await host.RunAsync();
```

Az így elkészült futtatható fájlt a Windows szolgáltatásai közé az „*sc.exe create*„ 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 dokumentációik alapos tanulmányozásával eldönthetjük, hogy programunkban melyikre van szükségünk.

A System.Timers névtérben található Timer osztály [4] 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 eseménykezelők meghívódnak, a több szálas környezetnek köszönhetően az óra pedig fut tovább.

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

```
...
_timer = new System.Timers.Timer();
_timer.Interval = 1000;
...
timer.Elapsed += (sender, e) =>
      ...
      _handler.RunDlls();
...
```

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

```
...
IterationCounter++;
...
```

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

```
private bool CanStartRun() =>
      (Handler.IterationCounter - _startedAt) % _timer == 0;
...

if (!_isLoaded || !CanStartRun() || _isCurrentlyRunning) return;
...
await (Task)_runMethod?.Invoke(_instance, Array.Empty<object>());
...
```

2.1.2. Task objektum, async és await kulcszó

A System.Threading.Task névtérben található Task osztály [5] egy műveletet reprezentál, ami végrehajtódása előtt általában aszinkron módon felsorakozik a Thread Pool-ra, majd sorra kerülés után lefut egy szálon. 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 helyette elvégzi azt.

Az await kulcsszó [6] segítségével tudjuk megvárni az előbb említett Task objektum sorra kerülését és lefutását. Hasznos lehet, ha például a felhasználó várakoztatásának ideje alatt nem a lefagyott, hanem egy töltő képernyőt jelenítünk meg. 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 [6] teszi lehetővé, hogy az ezzel megjelölt metódusok törzsében használható legyen az await kulcsszó, 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, ahol csak lehetséges aszinkron programozásra, és Task objektumok használatára van szükség, ezért általában async Task visszatérési értékkel definiált metódusokat használunk, amiknek megvárásáról is gondoskodunk.

```
...
await ContextContainer.LoadAssemblyWithReferences(zipPath,
      _parameters.MaxCopyTimeInMiliSec);


...
public async Task<bool> LoadAssemblyWithReferences(string zipPath,
      int maxCopyTimeInMiliSec)
{
      ...
      string? rootDirPath = await ZipExtracter.ExtractZip(zipPath,
            maxCopyTimeInMiliSec);
      ...
```

2.1.3. FileSystemWatcher

A .NET Core System.IO névterében található FileSystemWatcher [7] egy keretrendszer által biztosított osztály, ami képes egy könyvtárban, és annak alkönyvtáraiban figyelni azt, ha egy megadott mintára illeszkedő névvel rendelkező fájllal valami változás történt, és lefuttatni az eseményhez hozzárendelt eseménykezelőt.

Ezt az osztályt használhatjuk arra, hogy futás közben fájlok hozzáadásakor vagy törlésekor elindítsuk a megfelelő metódusokat. Az osztály konstruktorába először is injektáljuk az alkalmazás paramétereit, amikkel létrehozzuk az objektumot. Ezek a paraméterek az alkalmazás exe fájlának mappájában található appsettings.json fájlban vannak definiálva, amiket az alkalmazás felületéről való indítása során tudunk módosítani.

```
public Handler(IContextContainer contextContainer,
	IOptions<ParametersModel> parameters)
{
	...
	_fileSystemWatcher = new FileSystemWatcher(_monitoringPath,
		_parameters.Pattern)
	{
		EnableRaisingEvents = true,
	};
	...
```

Ezután a hozzáadás és a törlés delegate-ekhez hozzáadjuk az eseménykezelőket, amik a dll-ek betöltéséért és kitörléséért felelősek.

```
...
_fileSystemWatcher.Created += (sender, file) =>
	OnFileAdd(file.FullPath);
_fileSystemWatcher.Deleted += (sender, file) =>
	OnFileDelete(file.FullPath);
...
```

Ezek aszinkron metódusok, amik void a visszatérési értékkel rendelkeznek. Ez azért fontos, mert ezek is csakúgy, mint a Task műveletek aszinkron módon futnak le, viszont az ilyen típusú függvényekből kidobódott kivételeket nem tudjuk elkapni, hiszen nem await-elhetők, ráadásul Task objektum sincsen, amin keresztül vissza tudna jönni a hiba a CallStack-en.

2.1.4. Reflection

A System.Reflection névtérben található Reflection [8] 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 elérjük az adattagjait és a metódusai is meghívhatóvá válnak.

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. Ezt megelőzi néhány ellenőrző lépés.

Először megbizonyosodunk róla, hogy létezik-e a példányosítandó osztály, és hogy pontosan egy van-e belőle.

```
...
return assembly.ExportedTypes.Single();
...
```

Ezután megnézzük, hogy implementálja-e a kötelező interface-t.

```
...
if (exportedClass.GetInterface(Constants.I_WORKER_TASK) is null)
...
```

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

```
...
var instance = assembly.CreateInstance(exportedClass.FullName);
var runMethod = exportedClass.GetMethod("Run");
var timerPropety = exportedClass.GetProperty("Timer");
var timer = (uint?)timerPropety?.GetValue(instance);
...
```

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

```
private bool CheckIfPropertiesAreNull(object? instance,
      MethodInfo? runMethod, uint? timer) =>
            instance is null || runMethod is null
            || timer is null || timer == 0;
...
```

2.1.5. AssemblyLoadContext

.NET Core-ban megszűnt a lehetőség több AppDomain létrehozására egy folyamaton belül. Ez helyett bevezették az AssemblyLoadContext-et.

Az AssemblyLoadContext osztály [9] 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 képes futás közben elengedni dll fájlokat.

Ezt kihasználva dll-ek dinamikus betöltését és azok dinamikus elengedését is meg tudjuk valósítani. Ehhez először kicsomagoljuk a FileSystemWatcher által detektált zip kiterjesztésű fájlt, 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.

```
_assemblyLoadContext = new AssemblyLoadContext(string.Empty,
      isCollectible: true);
```

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

```
...
Assembly? assembly = _assemblyLoadContext?
      .LoadFromAssemblyPath(filePathToLoad);
...
foreach (var referencedAssembly in assembly.GetReferencedAssemblies())
{
      ...
      string? referencedAssemblyPath = DllLifter?
      .CopyFileToRunnerDir(rootDirName, referencedAssembly.Name);
      ...
      if (LoadAssembliesWithRecursion(rootDirName,
      referencedAssemblyPath) is null)
      {
            return null;
      }
      ...
```

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 a nem használt assembly objektumokat annak érdekében, hogy az Unload metódus el tudja engedni a dll-eket.

```
...
RunableInstance?.UnleashReferences();
_assemblyLoadContext?.Unload();
...
FreeMemoryOfContext(rootDirPath);
...
```

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 a hibák okának kiderítésekor.

Naplózni fájlba, adatbázisba és konzolra is tudunk. Vannak platformok, amik a naplózás konfigurálását rugalmasan, futás közben is kezelni tudják. Egy ilyen naplózó NuGet csomag például az NLog [10], amit egy config fájl beállítása után egyszerűen használhatunk a programunkon belül.

```
...
<targets>
	<target xsi:type="File"
	name="allfile"
		fileName="c:\temp\nlog-all-${shortdate}.log"
		...
	<target xsi:type="File"
		name="ownFile-web"
		fileName="c:\temp\nlog-own-${shortdate}.log"
		...
	<target xsi:type="File"
		name="rotatelog"
		fileName="c:\temp\nlog-rotate-${shortdate}.log"
		...
</targets>
...
```

Az IHostBuilder log-olásának konfigurálása után injektálás segítségével használhatjuk a logger-t.

```
...
.ConfigureLogging(logging =>
{
	logging.ClearProviders();
	logging.SetMinimumLevel(Microsoft.Extensions.Logging
		.LogLevel.Trace);
})
.UseNLog()
...
```

```
...
if (!success)
{
	_logger.LogError("Nem sikerült betölteni a(z) {zipPath} feladatot.",
		zipPath);
	...
	return false;
}
```

2.2. Blazor

A szakdolgozat Windows Service részéhez fejlesztésre került egy webes felület is annak érdekében, hogy a felhasználók kezelni tudják, és nyomon tudják követni a szolgáltatásban 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 teljes mértékben felépíthető egy webalkalmazás. Futtatására két féle lehetőség áll a rendelkezésünkre. Az egyik lehetőség a Blazor Server, ami szerver oldali render-elést biztosít. Ezt használva a kliens SignalR kapcsolaton keresztül kommunikál a szerverrel, és 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 egészben ott fut, amit a saját számítógépe működtet. Ebben az esetben a C# program korlátozva van a böngésző képességeire, például nem fér hozzá az operációs rendszer szolgáltatások listájához, hiszen a böngészőben való futás korlátozza ezt.

A web alkalmazás elsődleges célja az, hogy elérje és bizonyos mértékben kezelje a Windows szolgáltatást. Ahhoz, hogy erre képesek legyen, a WebAssembly korlátozásai miatt a Blazor Server technológiát kell választanunk.

2.2.1. Service Controller

A .NET System.ServiceProcess névtérben található ServiceController osztály [11] a Windows operációs rendszeren található Windows szolgáltatások kezelésére szolgál. Segítségével kapcsolódni tudunk a szolgáltatásokhoz, 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 szolgáltatást szeretnénk kezelni, példányosításához elég egy paramétert megadnunk, a szolgáltatás nevét. Ez alapján a program ki tudja keresni a szolgáltatások listájából a megfelelőt.

```
...
private ServiceController? _serviceController;
...
_serviceController = new(Constants.SERVICE_NAME);
...
```

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

```
_serviceController?.Start();
_serviceController?.WaitForStatus(ServiceControllerStatus.Running,
      TimeSpan.FromSeconds(_secondsToWaitForService));
...
```

Habár a fejlesztői környezet nem figyelmeztet rá, ezek műveletek hibát dobhatnak, ha nem sikerül elindítani a szolgáltatást, mert például az nem szerepel a szolgáltatások listájában.

2.2.2. Toastr

A felhasználói interakció hatására a háttérben általában eseményhez hozzárendelt metódusok, függvények futnak le, amik elvégzik az implementált programot. Az eredmény sikerességét sokféleképpen a felhasználó tudomására tudjuk hozni.

Ennek egyszerű módját egy Alerter könyvtár használata, a Toastr [12]. A HTML fájlba egy CSS és egy script fájl betöltése után rendelkezésünkre fognak állni a Toastr könyvtárban definiált JavaScript függvények, amik lefuttatása figyelemfelkeltő előugró dialógusokat eredményeznek.

```
...
<link href="https://cdnjs.cloudflare.com/ajax/libs/toastr.js
/latest/css/toastr.min.css" rel="stylesheet" />
<script src="https://cdnjs.cloudflare.com/ajax/libs/toastr.js
/latest/js/toastr.min.js" crossorigin="anonymous"></script>
...
```

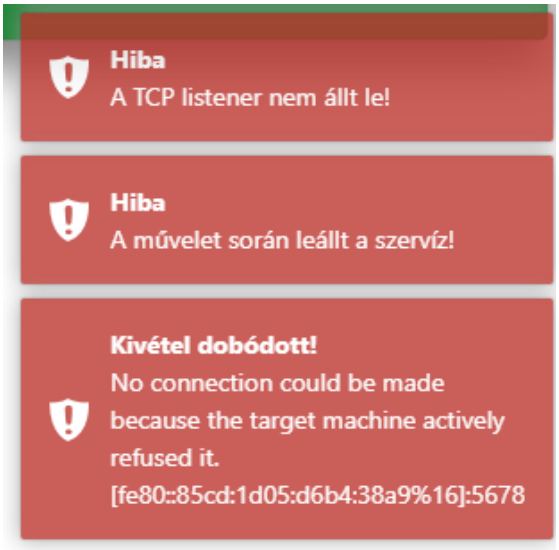
A Blazor a C# kód mellett nem zárja ki a JavaScript kód futtatásának lehetőségét sem. Az IJSRuntime kiterjesztésével és injektálásával futás közben C# kódból futtatni tudunk saját JavaScript kódot is, amit a keretrendszer gond nélkül kezel.

```
public static async ValueTask SuccessAlert(this IJSRuntime JSRuntime,
      string title, string message)
{
      await JSRuntime.InvokeVoidAsync("ShowToastr", "success",
            title, message);
}

[Inject]
private IJSRuntime JSRuntime { get; set; }

public async Task SuccessAlerter(string message) =>
      await JSRuntime.SuccessAlert("Siker", message);
```

2.1. ábra: Alerter a szolgáltatás sikeres elindításáról



2.2. ábra: Alerter a szolgáltatás sikertelen elindításáról

2.2.3. TCP Listener és TCP Client

A .NET System.Net.Sockets névterében található a TCP Listener [13] és TCP Client [14] osztályok segítségével TCP kapcsolaton keresztül tud kommunikálni egymással a listener és a kliens.

A listener egyféle szerverként viselkedik. Egy beállított IP címen és port-on 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 port-ra, amelyiken a listener hallgat, az üzenetet egy bájt tömbbe csomagolva átküldi a TCP kapcsolaton keresztül, majd megvárja a választ.

Ezt az osztályt használni tudjuk arra is, hogy a webes felület le tudja kérdezni a szolgáltatásban futó feladatok státuszát.

A Service alkalmazásban elindul a listener, és egy ciklusban várakozik a beérkező kliens kapcsolatokra.

```
public async void StartListening(Func<RequestMessage, Task<string>> GetJsonData)
{
	...
	TcpListener listener = new(_ipAddress, _port);
	listener.Start();
	...
	while (!shouldStop)
	{
		...
		TcpClient client = await listener.AcceptTcpClientAsync();
		...
```

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

```
...
TcpClient client = new(_hostName, _port);
NetworkStream stream = client.GetStream();
await ListenerHelper.WriteJsonToStream(stream, requestMessage.ToString());
...
```

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

```
...
NetworkStream stream = client.GetStream();
string data = await ListenerHelper.GetStringFromStream(stream);
...
string responseJson = shouldStop ? _responseWhenListenerStops
	: await GetJsonData(requestMessage);
await ListenerHelper.WriteJsonToStream(stream, responseJson);
stream.Close();
client.Close();
...
```

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

```
...
string response = await ListenerHelper.GetStringFromStream(stream);
stream.Close();
client.Close();
return JsonHelper.Deserialize<Tout>(response);
...
```

3. A kész alkalmazás

3.1. Funkciók tesztelése

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

Egy Windows szolgáltatás futásának nyomon követéséhez konzol és alapértelmezett felhasználói felület nem áll rendelkezésünkre, így a naplózás tűnik a legegyszerűbb megoldásnak. A korábban már említett NLog használatával kiírattathatjuk a metódusok futásának státuszát. Mivel az alkalmazás indításának belépési pontját egy try – catch-ágba tettük, biztosak lehetünk benne, hogy az elkapható, de kezeletlen hibákat is naplózni tudjuk az alkalmazás leállása előtt.

3.1.1. Indítás funkció

A webes felület indításakor az indítási felület fogad minket.

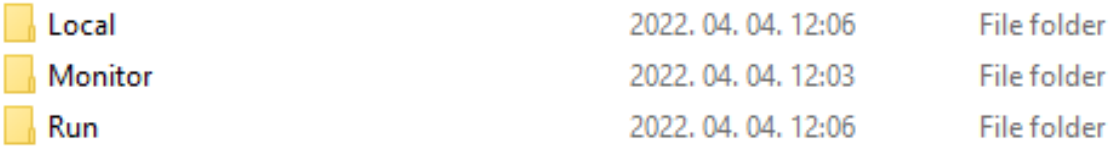
3.1. ábra: Blazor alkalmazás indítási felülete

A felhasználói felületnek három feladata van az indítás gomb megnyomásával. Az egyik, hogy a szolgáltatás exe kiterjesztésű fájlának mappájába elkészítsen egy json kiterjesztésű

fájlt, ami a form-ban megadott paramétereket is tartalmazza. Induláskor ezt a fájlt fogja a szolgáltatás feldolgozni, majd a megfelelő kulcs értékeit felhasználni paraméterként.

3.2. ábra: Paraméterek az appsettings.json fájlban

A második feladata, hogy előkészítse a szolgáltatás futásához szükséges mappákat.

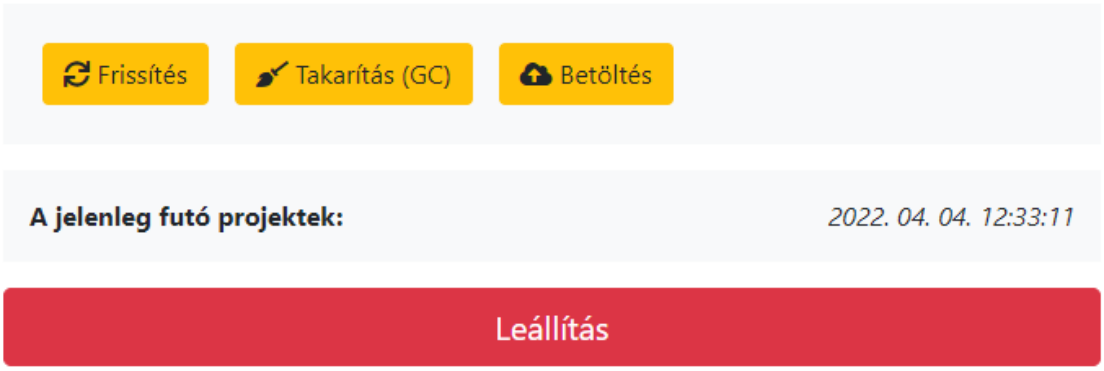


3.3. ábra: A szolgáltatás futásához szükséges mappák

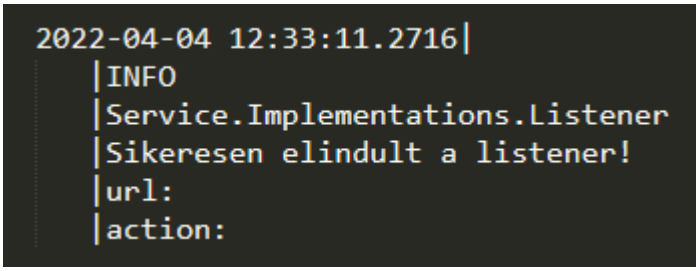
A harmadik feladata pedig az, hogy indítsa el a szolgáltatást. A szolgáltatás állapotát a Windows operációs rendszer Szolgáltatások alkalmazásában tekinthetjük meg.

3.4. ábra: A szolgáltatás futását jelző panel

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



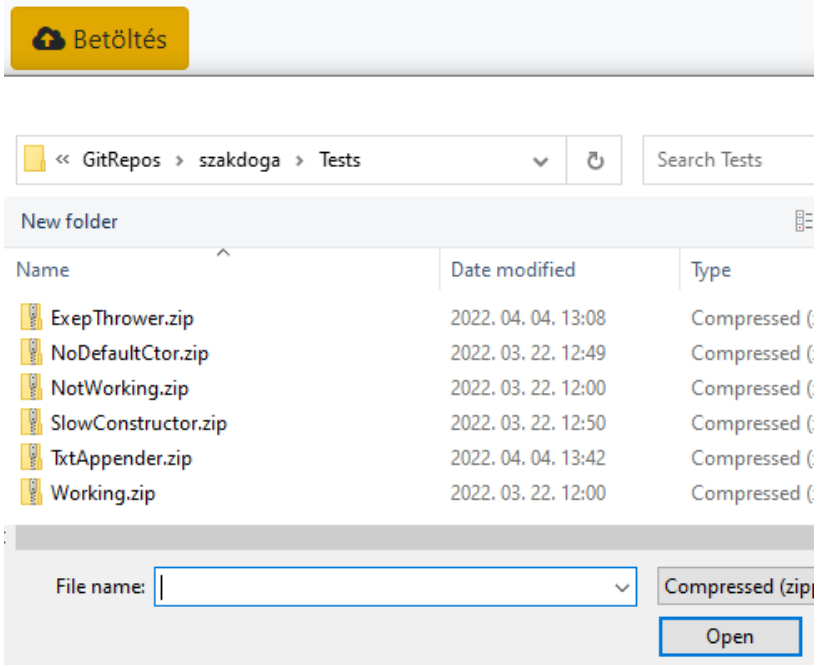
3.5. ábra: Blazor alkalmazás futási felülete



3.6. ábra: Naplóadat a TCP Listener elindulásáról

3.1.2. Betöltés funkció

A betöltés gomb megnyomása hatására egy fájl választó ablak ugrik elő, ami a betöltendő zip kiterjesztésű fájlokat várja.



3.7. ábra: Fájl választó ablak megjelenése a betöltés gomb megnyomásakor

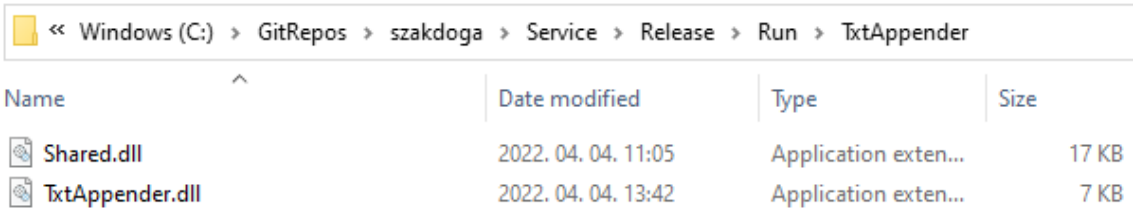
A kiválasztás után a betöltés folyamata részletes naplózásra kerül.

Sikeres betöltés esetén a hibaüzenet nélküli folyamatot, valamint a futtatáshoz szükséges fájlokat láthatjuk.

```
2022-04-04 13:43:02.3079|
   |INFO
   |Service.Implementations.Runable
   |A(z) TxtAppender.Class1 sikeresen példányosítva lett.
   |url:
   |action:

2022-04-04 13:43:02.3079|
   |INFO
   |Service.Implementations.ContextContainer
   |C:\GitRepos\szakdoga\Service\Release\Monitor\TxtAppender.zip sikeresen be lett töltve!
   |url:
   |action:
```

3.8. ábra: Naplóadat sikeres feladat betöltésről



3.9. ábra: Szükséges dll fájlok megfelelő helyre másolva

Sikertelen betöltés esetén a részletes hibák naplózásra kerülnek.

```
2022-04-04 14:00:35.8944|
   |ERROR
   |Service.Implementations.Runable
   |Hiba történt a(z) NoDefaultCtor.Class1 példányosítása során!
   System.MissingMethodException: Constructor on type 'NoDefaultCtor.Class1' not found.
      at System.RuntimeType.CreateInstanceImpl(BindingFlags bindingAttr,
         Binder binder, Object[] args, CultureInfo culture)
      ...
      at Service.Implementations.Runable.ConstructClassFromAssembly(Assembly assembly, Type exportedClass)
         in C:\GitRepos\szakdoga\Service\Implementations\Runable.cs:line 83
   |url:
   |action:
```

3.10. ábra: Naplóadat példányosítási kivételről

```
2022-04-04 14:00:35.8944|
   |ERROR
   |Service.Implementations.Runable
   |Nem sikerült példányosítani a(z) NoDefaultCtor.Class1 nevű osztályt
   |url:
   |action:

2022-04-04 14:00:35.8944|
   |ERROR
   |Service.Implementations.ContextContainer
   |Nem sikerült betölteni a(z)
      C:\GitRepos\szakdoga\Service\Release\Monitor\NoDefaultCtor.zip projektet.
   |url:
   |action:
```

3.11. ábra: Naplóadat sikertelen feladat betöltésről

3.1.3. Futtatás funkció

Sikeres betöltés után a betöltött feladat megkezdi futását. Ezt a funkciót teljes mértékben a szolgáltatás kezeli a háttérben, megállítására a projekt kitörlésével van lehetőség. A futtatást a korábban említett Timer osztály hajtja megfelelő időközönként, amit egy olyan feladattal tesztelünk, ami 4 másodpercenként beleírja egy szöveges fájlba, ha sikeresen lefutott.

```
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:09-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:13-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:17-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:21-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:25-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:29-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:33-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:37-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:41-kor!
Lefutott a TxtAppender projekt Run metódusa 2022. 04. 04. 13:58:45-kor!
```

3.12. ábra: Feladat pontosan időzített futásai

Egy feladat újbóli sorra kerülése esetén megtörténhet, hogy az előző futása programozói hiba miatt még nem ért a végére. Ilyenkor a futtatásának megismétlése kihagyásra kerül addig, amíg a beakadt futás a végére nem ért.

```
2022-04-06 15:18:17.4616|
    |ERROR|Service.Implementations.Runable
    |A(z) TxtAppender.Class1 előző futása még nem fejeződött be, ezért most kihagyásra kerül.
    |url:
    |action:
```

3.13. ábra: Naplóadat már futó feladatról

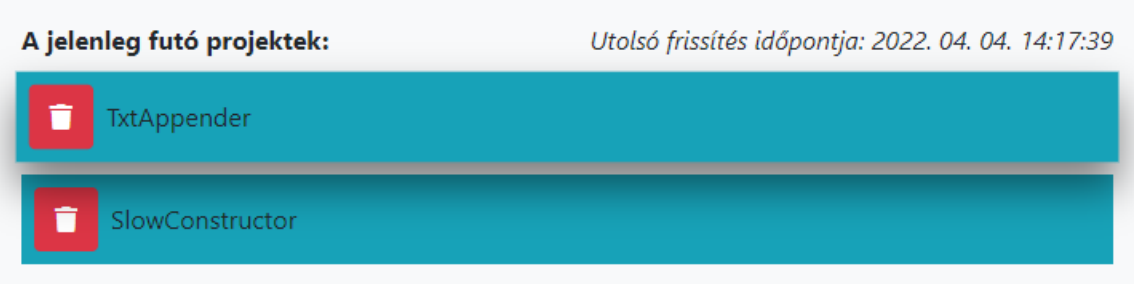
A feladatok lefutása során kivétel is dobódhat. Ezeket a kezeletlen hibákat elkapjuk és naplózzuk.

```
2022-04-04 13:24:23.0075|
   |ERROR
   |Service.Implementations.Runable
   |A(z) TxtAppender.Class1 hibát dobott futás közben.
   System.IO.IOException: The process cannot access the file
      'C:\GitRepos\szakdoga\Tests\text.txt'
      because it is being used by another process.
      at Microsoft.Win32.SafeHandles.SafeFileHandle.CreateFile(String fullPath, FileMode mode,
         FileAccess access, FileShare share, FileOptions options)
      ...
      at TxtAppender.Class1.Run()
      at Service.Implementations.Runable.Run()
         in C:\GitRepos\szakdoga\Service\Implementations\Runable.cs:line 108
   |url:
   |action:
```

3.14. ábra: Naplóadat futás közben történő hiba elkapásáról

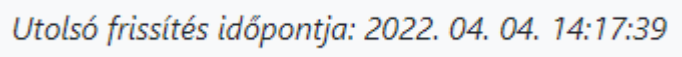
3.1.4. Frissítés funkció

A sikeresen betöltött projektek felületen való megjelenítéséhez a frissítés gomb megnyomására van szükség. Ilyenkor TCP kapcsolaton keresztül megérkeznek a service-ben aktuálisan futó feladatok nevei, amiket megjelenítünk a képernyőn.



3.15. ábra: Aktívan futó feladatok listája

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



3.16. ábra: Utolsó frissítés időpontjának frissülése

3.1.5. Törlés funkció

Egy feladat bármikor leállítható, és törölhető. Ehhez a törlés gomb megnyomására van szükség. Hatására a futó dll-ek el lesznek engedve a függőségek közül, majd a gyökér könyvtárukkal együtt törlődnek is, és a művelet eredménye naplózásra kerül.

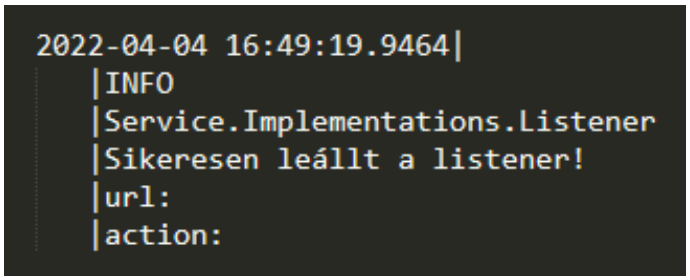
```
2022-04-04 13:24:56.7145|
   |INFO
   |Service.Implementations.ContextContainer
   |A(z) C:\GitRepos\szakdoga\Service\Release\Monitor\TxtAppender.zip
      sikeresen el lett engedve és ki lett törölve.
   |url:
   |action:
```

3.17. ábra: Naplóadat sikertelen törlésről

Amennyiben egy dll törlése nem sikerül, mert a program nem tudja elengedni az assembly-re mutató referenciát, a feladat futtatása megáll a gyökérkönyvtár törlése nélkül is.

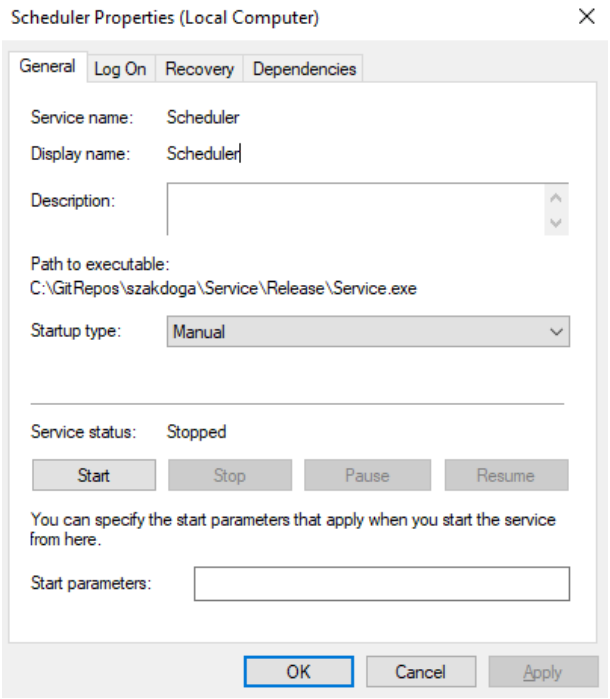
3.1.6. Leállítás funkció

A felületen található leállítás gombnak 2 feladata van. Az egyik, hogy kérvényezze a TCP Listener leállítását egy TCP kapcsolaton keresztül. A Listener sikeres leállítása naplózásra kerül.



3.18. ábra: Naplóadat TCP Listener sikeres leállításáról

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



3.19. ábra: Szolgáltatás leállását jelző panel

Sikeres leállítás esetén a felület visszavált az indítási képernyőre, és az egész folyamat megismételhetővé válik.

3.2. További fejlesztési lehetőségek

A szolgáltatás úgy készült, hogy be tudjon tölteni és futtatni tudjon bármilyen osztályt, ami implementálja az IWorkerTask interface-t. Ez az interface egy tömör és egyszerű módja annak, hogy közölni tudjuk a szolgáltatással, hogy mit csináljon, és hogy mennyi időnként.

```
public interface IWorkerTask
{
	uint Timer { get; }
	public Task Run();
}
```

A Timer property egy nemnegatív számot vár értékül, a Run metódus törzsében pedig bármit implementálhatunk, és Task visszatérési értéke miatt ezt akár aszinkron módon is használhatjuk.

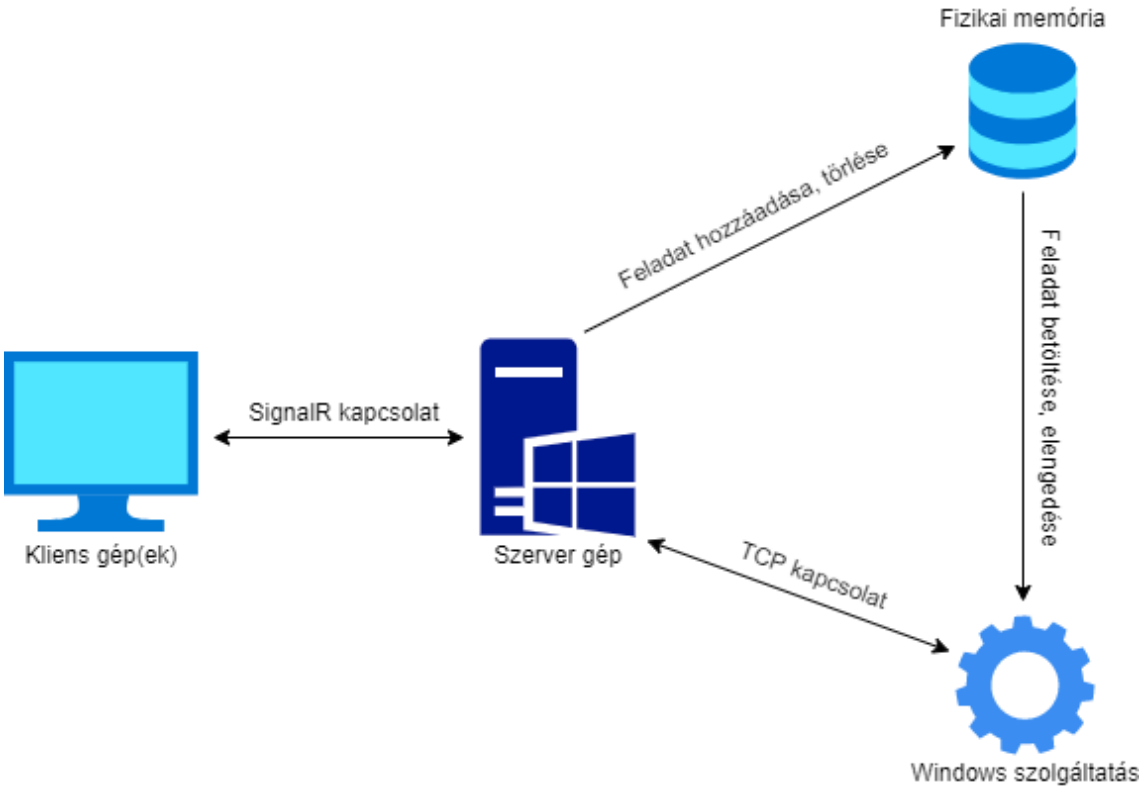
Egy ilyen osztály implementálása egy kezdő programozó számára sem nehéz feladat, az elkészült projekt futtatása pedig néhány kattintással megoldható a felületen keresztül.

Összegzés

A szakdolgozat keretein belül megismerkedtünk azokkal a technológiákkal és eszközökkel, amik segítségünkre voltak a szakdolgozat fejlesztésénél.

A megvalósítás során áttekintettük a Windows szolgáltatások előnyeit, hátrányait és különbségeit egy hagyományos alkalmazással szemben. Megismerkedtünk a Reflection-nel, aminek köszönhetően a szolgáltatásunk dinamikusan képes betölteni, elengedni és futtatni feladatainkat. Elsajátítottuk az aszinkron programozás mélységeit, ami lehetőséget adott a funkciók egyidejű futására.

Elkészítettünk egy Blazor web alkalmazást, ami egy TCP kapcsolaton keresztül képes kommunikálni a szerver számítógépen futó szolgáltatással, és üzeneteken keresztül bizonyos mértékben kezelni azt.



Az működő alkalmazás összes implementált funkcióját leteszteltük, azok hibátlan működéséről megbizonyosodtunk.

Irodalomjegyzék

[1] .NET, https://docs.microsoft.com/en-us/dotnet/, 2022.04.04.

[2] Visual Studio, https://visualstudio.microsoft.com/, 2022.04.04.

[3] Windows Service, https://docs.microsoft.com/en-us/dotnet/core/extensions/windows-service, 2022.04.04.

[4] Timer, https://docs.microsoft.com/en-us/dotnet/api/system.timers.timer?view=net-6.0, 2022.04.04.

[5] Task, https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-6.0, 2022.04.04.

[6] Aszinkron programozás, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/, 2022.04.04.

[7] FileSystemWatcher, https://docs.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher?view=net-6.0, 2022.04.04.

[8] Reflection, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection, 2022.04.04.

[9] AssemblyLoadContext, https://docs.microsoft.com/en-us/dotnet/api/system.runtime.loader.assemblyloadcontext?view=net-6.0, 2022.04.04.

[10] NLog, https://nlog-project.org/, 2022.04.04.

[11] ServiceController, https://docs.microsoft.com/en-us/dotnet/api/system.serviceprocess.servicecontroller?view=dotnet-plat-ext-6.0, 2022.04.04.

[12] Toastr, https://www.npmjs.com/package/toastr, 2022.04.04.

[13] TCP Listener, https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.tcplistener?view=net-6.0, 2022.04.04.

[14] TCP Client, https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.tcpclient?view=net-6.0, 2022.04.04.

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 a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2022. április 19.

.....

aláírás