[Külső fekete borítólap formátuma]

Széchenyi István Egyetem

Gépészmérnöki, Informatikai és Villamosmérnöki Kar

Informatika Tanszék

**SZAKDOLGOZAT**

**Bólya Richárd**

**Mérnök Informatikus BSc szak**

2021

|  |
| --- |
| [Gerincen:] Hallgató Neve, Évszám {Titkosított} |

**SZAKDOLGOZ****AT**

**Versenyautóból származó telemetriai adatok élő közvetítése és kielemzése**

**Bólya Richárd**

**Mérnök Informatikus BSc szak**

**2021**

# Nyilatkozat

Alulírott, [Hallgató Neve (Neptun-kód), Hallgató szakja, képzési szintje] szakos hallgató kijelentem, hogy a [Szakdolgozat Címe] című szakdolgozat feladat kidolgozása a saját munkám, abban csak a megjelölt forrásokat, és a megjelölt mértékben használtam fel, az idézés szabályainak megfelelően, a hivatkozások pontos megjelölésével.

Eredményeim saját munkán, számításokon, kutatáson, valós méréseken alapulnak, és a legjobb tudásom szerint hitelesek.

Győr, [beadás dátuma]

hallgató

# Kivonat

Versenyautóból származó telemetriai adatok élő közvetítése és kielemzése

[1 oldalas, magyar nyelvű tartalmi kivonat]

# Abstract

Live telemetry system for racecars

[1 oldalas, angol nyelvű kivonat]

# Tartalomjegyzék

Contents

[Nyilatkozat 10](#_Toc87185969)

[Kivonat 11](#_Toc87185970)

[Abstract 12](#_Toc87185971)

[Tartalomjegyzék 13](#_Toc87185972)

[Bevezetés 1](#_Toc87185973)

[Adat küldő 2](#_Toc87185974)

[Program felépítése 2](#_Toc87185975)

[Session 5](#_Toc87185976)

[Külső könyvtárak 5](#_Toc87185977)

[Program működése 5](#_Toc87185978)

[Web Api 6](#_Toc87185979)

[Kontrollerek 6](#_Toc87185980)

[Package controller 6](#_Toc87185981)

[Sensor controller 6](#_Toc87185982)

[Session controller 6](#_Toc87185983)

[Swagger 7](#_Toc87185984)

[Adatbázis 12](#_Toc87185985)

[Entity Framework 12](#_Toc87185986)

[Táblák 14](#_Toc87185987)

[Sensor 15](#_Toc87185988)

[SensorValue 15](#_Toc87185989)

[Session 15](#_Toc87185990)

[Telemetria alkalmazás 16](#_Toc87185991)

[Program felépítése 16](#_Toc87185992)

[Felhasználói felület 16](#_Toc87185993)

[Settings menü 16](#_Toc87185994)

[Live settings menü 17](#_Toc87185995)

[Groups settings menü 18](#_Toc87185996)

[Templates settings menü 18](#_Toc87185997)

[Configurations menü 19](#_Toc87185998)

[Live menü 20](#_Toc87185999)

[Program felépítése 21](#_Toc87186000)

[Adatelérési réteg 22](#_Toc87186001)

[Logikai réteg 24](#_Toc87186002)

[Megjelenítési réteg 24](#_Toc87186003)

[Színek és stílusok 31](#_Toc87186004)

[Validáció 32](#_Toc87186005)

[Validáció létrehozása 33](#_Toc87186006)

[Tesztek 35](#_Toc87186007)

[Manuális tesztek 35](#_Toc87186008)

[Unit tesztek 35](#_Toc87186009)

[Automatizált tesztelés 40](#_Toc87186010)

[Git 41](#_Toc87186011)

[GitHub 41](#_Toc87186012)

[Ágak 41](#_Toc87186013)

[Fejlesztés menete 41](#_Toc87186014)

[Taszkok 41](#_Toc87186015)

[Címkék 42](#_Toc87186016)

[Mérföldkövek 42](#_Toc87186017)

[Rendszer telepítése 44](#_Toc87186018)

[Adat küldő 44](#_Toc87186019)

[Web Api 44](#_Toc87186020)

[Telemetria alkalmazás 44](#_Toc87186021)

[Rendszer konfigurálása 45](#_Toc87186022)

[Adat küldő 45](#_Toc87186023)

[Telemetria alkalmazás 45](#_Toc87186024)

[Irodalomjegyzék 46](#_Toc87186025)

[Mellékletek 48](#_Toc87186026)

# Bevezetés

A motorsportok világában a legfontosabb dolog a telemetria. Telemetria nélkül nem lehet megfelelően kialakítani és beállítani az adott járművet. A csapatok erre rengeteg pénzt költenek és nem véletlenül.

Ebben a szakdolgozatban egy olyan telemetria rendszert készítettem el, amit elsősorban versenyautókhoz lehet használni és élőben lehet vele figyelni az autó adatait. A rendszer képes az autó adatait elmenteni, amit később szintén ezzel a programmal ki lehet elemezni. A rendszer használata nincsen egy számítógéphez kötve, akár több klienssel is lehet kapcsolódni az élő adatfolyamhoz.

A rendszer három fő részből áll. Van egy adat küldő része, amit a versenyautóban kell elhelyezni. Van maga az adatokat kielemző kliens program, amit egy Windows alapú számítógépen lehet futtatni. A harmadik rész, ami összeköti magát a rendszert, egy web api. Ez a modul van összeköttetésben a központi adatbázissal.

Ennek a rendszernek az elkészítéséhez legfőképp C# nyelvet használtam. A fejlesztés menetét sprintekre bontottam, Git alapú rendszerben követtem és a kód biztonságos fejlesztését automatizált unit tesztekkel támogattam.

# Adat küldő

Az adat küldő modul felelős a versenyautóból származó adatok szerverre történő eljuttatásáért. A rendszer ezen része Python programozási nyelven lett elkészítve, mégpedig pont azért, hogy a beágyazott eszközök színes skáláján elérhető legyen.

Én ezt kimondottan egy Raspberry PI-ra terveztem, mivel ezt az eszközt nagy vonalakban lehet testreszabni és mindezek között található hozzá olyan modul is, ami képes egy versenyautó CAN hálózatára csatlakozni.

## Program felépítése

A program egy fő, több kisebb scriptből és egy konfigurációs fájlból áll. Az a script ami felelős az adatok küldéséért a data\_sender.py. Ezt a scriptet kell futtatnia az autónak akkor amikor használatba kerül. Mivel ezt a rendszert nem tudtam valós körülmények között kipróbálni, ezért az adatok küldését szimuláltam.

A script elején betöltődnek a konfigurációk a configuration.json fájlból. Ebben a fájlban vannak definiálva azok a paraméterek, amikkel a program képes kommunikálni a web api-val.

{

"HTTP\_status\_code\_ok" : 200,

"wait\_between\_tries" : 5,

"wait\_between\_sending" : 2,

"max\_buffer\_size" : 5,

"isHTTPS" : false,

"url" : "10.0.0.100",

"port" : 5000,

"ignore\_warnings" : true,

"get\_live\_session\_api\_call" : "api/Session/live",

"post\_sensor\_api\_call" : "api/Sensor?sensorName=",

"post\_package\_api\_call" : "api/Package?packageJson="

}

A cím, amin elérhető a web api, az api hívások nevei, mennyi idő teljen el az adatok küldése között és az, hogy maximum mekkora lehet az aktuális csomag mérete.

Utána definiálom azokat a változókat, amiket fogok használni majd az adatok szimulálását.

Az adatok szimulálásához fájlból olvasok be adatokat. Erre három fájlom van, amik mind egy egy szenzort jelképeznek. Ezek a data\_files mappában találhatóak. A szimulálás csak annyiban tér el valós adatok küldésétől, hogy a beolvasott tömbbökből olvasom be az adatokat, nem pedig például egy CAN hálózatról.

Ez után definiáltam kettő függvényt. Az egyikkel előkészítem a szenzor adatokat, a másikkal pedig elkészítem a csomagot, amit ki fog küldeni az alkalmazás.

def prepare\_sensor\_values(sensor\_values):

prepared\_sensor\_values = []

sensor\_value\_index = data\_index

for value in sensor\_values:

if len(sensor\_values) > sensor\_value\_index:

if sensor\_value\_index<data\_index + MAX\_BUFFER\_SIZE:

sensor\_values[sensor\_value\_index].session\_id = live\_session\_id

prepared\_sensor\_values.append(sensor\_values[sensor\_value\_index])

sensor\_value\_index = sensor\_value\_index + 1

return prepared\_sensor\_values

def make\_package(prepared\_sensor\_values\_list) :

package = Package(package\_id, live\_session\_id, datetime.now().timestamp())

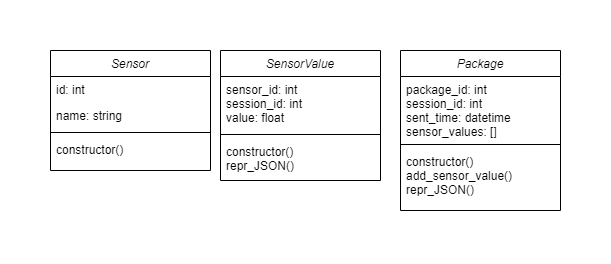
for prepare\_sensor\_values in prepared\_sensor\_values\_list:

for value in prepare\_sensor\_values:

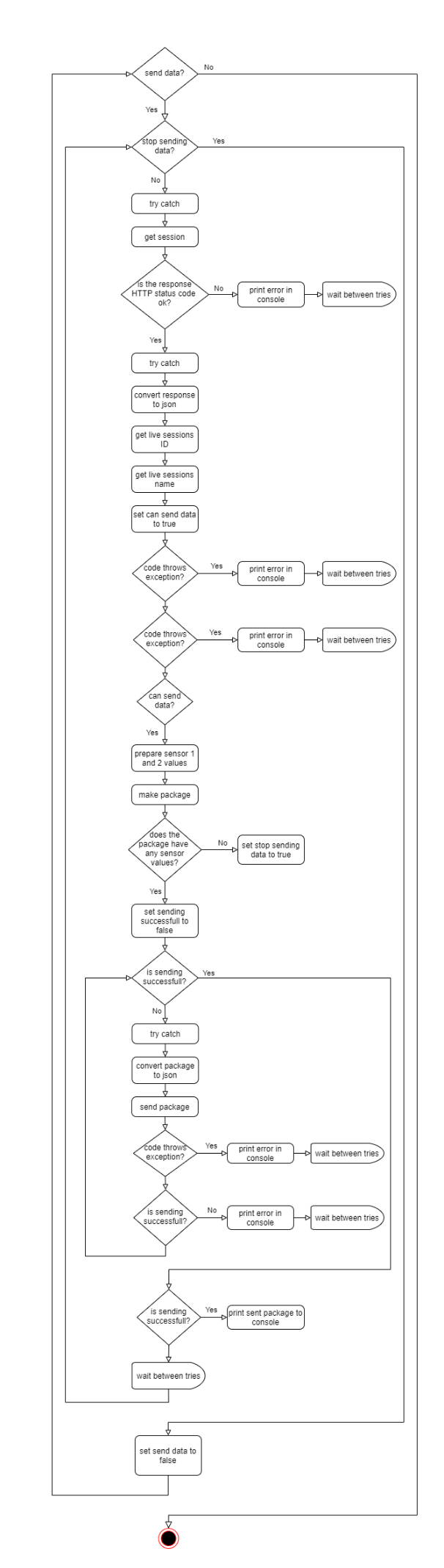
package.add\_sensor\_value(value)

return package

Ezekhez létrehoztam osztályokat, amik külön fájlokban találhatóak. Van egy Package, Sensor és SensorValue osztályom.



A SensorValue és Package osztályokban található egy repr\_JSON() függvény ami a rekurzív json konverzióban segít. Erre még bővebben is ki fogok térni később.

Az adatok küldése ez után kezdődik, amit az alábbi folyamatábrán szemléltetem.

Az adatokat addig gyűjtöm, amíg meg nem telik a buffer, majd ezeket egy csomagba rendezve elküldöm a szerverre.

Legvégül pedig kiírom a konzolra az elküldött adatokról egy összegzést.

### Session

Minden olyan esemény amikor a versenyautó működik, session-nek nevezek. Ez lehet egy versenyszám vagy akár egy teszt is. Minden session-t a versenymérnöknek kell kézzel felvennie a Telemetria alkalmazásban.

Minden session-nek van egy neve, létrehozási dátuma es státusza. Ezek mellett hozzá vannak rendelve azok a szenzorok, amiket az autó küld.

Miután a program talált egy élő session-t, először elküldi a szerverre azokat a szenzorokat, amik be vannak állítva, majd elkezdi az adatok küldését. Az adatokat a program csomagokba rendezi. A konfigurációs fájlból beolvasott idő közönként kiküldi az éppen aktuális csomagot, ellenkező esetben pedig gyűjti a szenzor adatokat az aktuális csomagba.

### Külső könyvtárak

Több külső könyvtárat is használok, amik elengedhetetlenek a web apival való kommunikációhoz. Ilyenek például a requests és a json. Az első modullal lehet kommunikálni egy külső api-val, az utóbbival pedig bármilyen objektumot lehet jsonná konvertálni. Mivel én összetett adatokkal dolgozok, ezért minden osztálynak van külön json convertere.

## Program működése

A program, az elindítása utána addig fog futni, amíg az eszközt le nem állítjuk vagy estleg a Telemetria alkalmazásból állítjuk le. Indítás után a program rögtön elkezdi gyűjteni a szenzorok adatait, viszont még nem küldi őket sehova.

A program addig vár az adatok küldésével, amíg nem talál egy élő session-t. Amint talált, elkezdi küldeni rá az adatokat.

# Web Api

Ez a modul a rendszer közepe. A rendszer ezen része áll összeköttetésben a többi modullal. Ez az az interface, amin keresztül el lehet érni az adatbázist.

## Kontrollerek

### Package controller

Ebben a kontrollerben Get illetve Post hívások találhatóak.

* Get
  + Ez a hívás adja vissza a paraméterként kapott session és csomag ID alapján az összes rsensor értékeket.
* GetAfter
  + Ez a hívás visszaadja a paraméterként kapott utolsó csomag ID és session ID alapján az utoljára elküldött csomag óta keletkezett szenzor adatokat.
* GetAll
  + Ez a hívás visszaadja a paraméterként kapott session-höz tartozó összes csomagot.
* Post
  + Ez a hívás egy új csomagot dolgoz fel, amit json formában paraméterként kap.
  + Ha sikeres volt a csomag feldolgozása, 200-as Http státusz kóddal tér vissza, ellenkező esetben pedig 500-al.

### Sensor controller

* GetAll
  + Ez a hívás visszaadja az összes szenzort.
* GetSensorNames
  + Ez a hívás a paraméterben kapott sessionhöz adja vissza a sensorokat.
* Post
  + Ez a hívás egy sensort ment el amit Json formában kap paraméterként.

### Session controller

* HealthCheck
  + Ezzel a hívással lehet könnyedén kideríteni, hogy lehet e kapcsolatot létesíteni a szerverrel.
  + Ez a hívás visszaad egy igaz értéket.
* Get
  + Ez a hívás visszaadja az éppen elő sessiomt.
  + Ha nem létezik ilyen, akkor null-al tér vissza.
* GetAll
  + Visszaadja az összes sessiont.
* GetActiveSessionSensors
  + Visszaadja aktív session összes szenzorát.
* Post
  + Ez a hívás egy új sessiont ment el.
  + Sikeres mentés esetén 200-as Https státusz kóddal tér vissza.
  + Sikertelen mentés esetén 500-as kóddal tér vissza.
  + Abban az esetben, ha ilyen session már létezne, 409-es kóddal tér vissza ami conflict-ot jelent.
* ChangeToLive
  + Ez a hívás élőre állítja a paraméterként kapott sessiont.
  + Sikeres mentés esetén 200-as Http státusz kóddal tér vissza.
  + Sikertelen mentés esetén 500-al.
  + Abban az esetben, ha már létezik élő session, 409-el tér vissza, mivel egyszerre csak egy élő session létezhet.
* ChangeName
  + Ez a hívás a paraméterként kapott session-nek a nevét változtatja meg. A session amit megkap json formátumban van, és már benne van az új név.
  + Sikeres mentés esetén 200-as Http státusz kóddal tér vissza.
  + Sikertelen mentés esetén pedig 500-al.
* Delete
  + Ez a hívás kitöröl egy sessiont.
  + A törlendő session ID-ját parameterként kapja
  + Sikeres törlés esetén 200-as Http státusz kóddal tér vissza.
  + Sikertelen törlés esetben pedig 500-al.

A kontrollerek felelősek az api hívásokért. Ezeket az osztályokat a ControllerBase osztályból kell származtatni, illetve meg kell adni az osztálynak kettő attribútumot. Először is, hogy mi az útvonal, illetve egy olyat, amivel jelezzük, hogy ez az osztály egy api controller.

[Route("api/[controller]")]

[ApiController]

public class PackageController : ControllerBase

…

Egy új api hívást a következő képpen lehet definiálni. Létre kell hozni egy függvényt, amiben implementáljuk az adott hívás logikáját, majd kettő attribútumot kell rárakni. Az egyikkel a hívás típusát adjuk meg, a másikkal pedig az útvonalat.

[HttpGet]

[Route("get\_single/{packageId}/{sessionId}")]

public List<SensorValue> Get(int packageId, int sessionId)

{

return PackageManager.GetPackageSensorValues(packageId, sessionId);

}

Ezen kívül lehet megadni más attribútumot is, például egy olyat, amivel a válasz típusát adjuk meg. Az alábbi példában egy http státusz kóddal tér vissza, ami egy enum. Itt megadom, hogy melyik három értékkel térhet vissza.

[HttpPut]

[Route("change\_live")]

[ProducesResponseType(200)]

[ProducesResponseType(409)]

[ProducesResponseType(500)]

public HttpStatusCode ChangeToLive([FromBody] int sessionId)

{

return SessionManager.ChangeActiveSession(sessionId, isLive: true);

}

## Swagger

A Swagger egyfajta interaktív dokumentáció az API-hoz. Ez a kiegészítő automatikusan legenerálja az alábbi képen látható módon az összes hívás rövid leírását és használati útmutatóját. Ezeket ki lehet egészíteni megfelelő szintaktikájú kommentekkel.

Graphical user interface, text, application, email

Description automatically generated

Szépen látható, ahogy a Swagger minden controllert szétválasztja és a különböző típusú hívásokat megegyező színekkel ábrázolja. A get hívások kékek, a post hívások zöldek, a put hívások narancssárgák és a delete hívások pirosak.

Ennek a kiegészítő modulnak a legnagyobb előnye amelett, hogy egyéretlműen látható az API működése az, hogy interaktív.

Graphical user interface, text, application

Description automatically generated

Az alábbi képen a get all package hívás látható. A Swagger mutat egy példa json-t, amit az API vissza fog adni. A Try it out gombra kattintva ezt ki tudjuk próbálni. Az alábbi képen azt láthatjuk, ahogy a 70-es ID-jű session csomagjait hozza le json formátumban. Az is látható, hogy a sensor és session mindenhol null, mivel ezek a modell osztály virtuális property-i, amik az idengekulcshoz szükségesek. Ezeket a json serialization modul sajnos belegenerálja.

Graphical user interface, text

Description automatically generated

Az alábbi ábrán arra láthatunk példát, hogy speciális kommentekkel hogyan fejleszthetjük ezt a dokumentációt.

Graphical user interface, application

Description automatically generated

Látható, ahogy a szekció elején a summary szövege van, látható a paraméter leírása, illetve látható a három visszatérési értéknél a magyarázat. Mindez komment szinten dinamikus, mivel a paraméterek és a visszatérési értékek referálva vannak, tehát ha például változik a paraméter neve, akkor a kommentben is változni fog, amivel pedig a felületen is változni fog.

/// <summary>

/// Changes the live session to the session with <paramref name="sessionId"/> if there no live session.

/// <param name="sessionId">Id of the session.</param>

/// </summary>

/// <response code="200">Sucessfully changed.</response>

/// <response code="409">There is already an active session.</response>

/// <response code="500">There was an error with the database.</response>

[HttpPut]

[Route("change\_live")]

[ProducesResponseType(200)]

[ProducesResponseType(409)]

[ProducesResponseType(500)]

public HttpStatusCode ChangeToLive([FromBody] int sessionId)

{

return SessionManager.ChangeActiveSession(sessionId, isLive: true);

}

## Adatbázis

Kizárólag ez a modul képes direktbe elérni az adatbázist. Ebben a projektben kettő rétegű szerkezetet használtam. Elkülönítettem a prezentációs réteget és az adatelérési réteget. Úgy gondoltam, hogy itt nincsen értelme egy harmadik köztes rétegnek a prezentációs és adatelérési réteg között. Az ilyen dolgokat manager osztályok kezelik, amiket könnyedén el tudnak érni a kontroller osztályok. Ezek az osztályok az adatelérési rétegben vannak.

### Entity Framework

Az Entity Framework segítségével kezeltem az adatbázist. Az Entity Framework egy, a Microsoft által készített adatbázis kezelő framework. Ezzel könnyedén lehet C#-ban adatbázist kezelni.

A táblákat a **DatabaseContext** nevű osztályba hoztam létre, amit a **DbContext** osztályból származtattam.

Itt megadtam az adatbázis connection string-jét és létrehoztam a táblákat, amikr DbSet property-k. Ezt az alábbi képen láthatjuk.

A connection string az alábbi képpen néz ki: Server=RICSI; Initial Catalog=DataCenter; Integrated Security=SSPI;

Itt az adatbázis szerver neve **RICSI** és az adatbázis neve **DataCenter**.

Az alábbi kódrészletben láthatunk példát arra, hogy hogyan lehet az Entity Frameworkben táblát létrehozni. Az az osztály, amit a DbSet-be megadtam, annak formájára fogja a framework létrehozni a táblát.

using Microsoft.EntityFrameworkCore;

using DataLayer.Models;

namespace DataLayer

{

public class DatabaseContext : DbContext

{

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=RICSI; Initial Catalog=DataCenter; Integrated Security=SSPI;");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

}

public DbSet<Session> Session { get; set; }

public DbSet<Sensor> Sensor { get; set; }

public DbSet<SensorValue> SensorValue { get; set; }

}

}

Az elsődleges kulcsot a **[Key]** attribútummal kell ellátni. Az idegen kulcsnak nem kell külön attribútum, viszont a nevének követnie kell a következő szabályt: tábla neve + Id. Emellett egy virtuális property-t is létre kell hozni aminek meg kell egyeznie teljes mértékben a **DatabaseContext**-ben létrehozott táblának típusával és nevével. Az alábbi képen a kommentek segítenek ezeknek a könnyed megtalálására.

A **[Required]** attribútum felelős azért, hogy az adott mezőnek mindenképpen kell értéket adni. Ellenekző esetben hibát fog dobni a framework.

using System.ComponentModel.DataAnnotations;

namespace DataLayer.Models

{

public class SensorValue

{

[Key]

public int SensorValueId { get; set; } // PK

[Required]

public double Value { get; set; }

[Required]

public int SensorId { get; set; } // FK

[Required]

public int SessionId { get; set; } // FK

[Required]

public int PackageId { get; set; }

public virtual Sensor Sensor { get; set; } // required for foreign key

public virtual Session Session { get; set; } // required for foreign key

}

}

### Táblák

Az alábbi képen látható az adatbázis struktúrája. Lehet látni a Microsoft SQL Server Management Studio nevű program által létrehozott adatbázis diagrammon, hogy az Entity Framework jól létrehozta az elsődleges és idegenkulcsokat.

Diagram

Description automatically generated

Az alábbi képen pedig az látható, hogy a framework automatikusan legenerált egy indexet az idegen kulcsoknak és az elsődleges kulcsnak. Az indexenek fontos szerepük van egy adatbázisban mivel felgyorsítják az adatok keresését.

Graphical user interface, text, application

Description automatically generated

Az alábbi ábrán látható a három tábla felépítése.

#### Sensor

Table

Description automatically generated

#### SensorValue

Table

Description automatically generated

#### Session

###### Table Description automatically generated

# Telemetria alkalmazás

A rendszer ezen része felel az adatok vizuális megjelenítéséért. Ez egy Windows alkalmazás, ami a Windows Presentation Foundation keretrendszerben készült. A program képes a session-öket, a szenzorokat és az adatokat kezelni. A program nyelve angol.

## Program felépítése

### Felhasználói felület

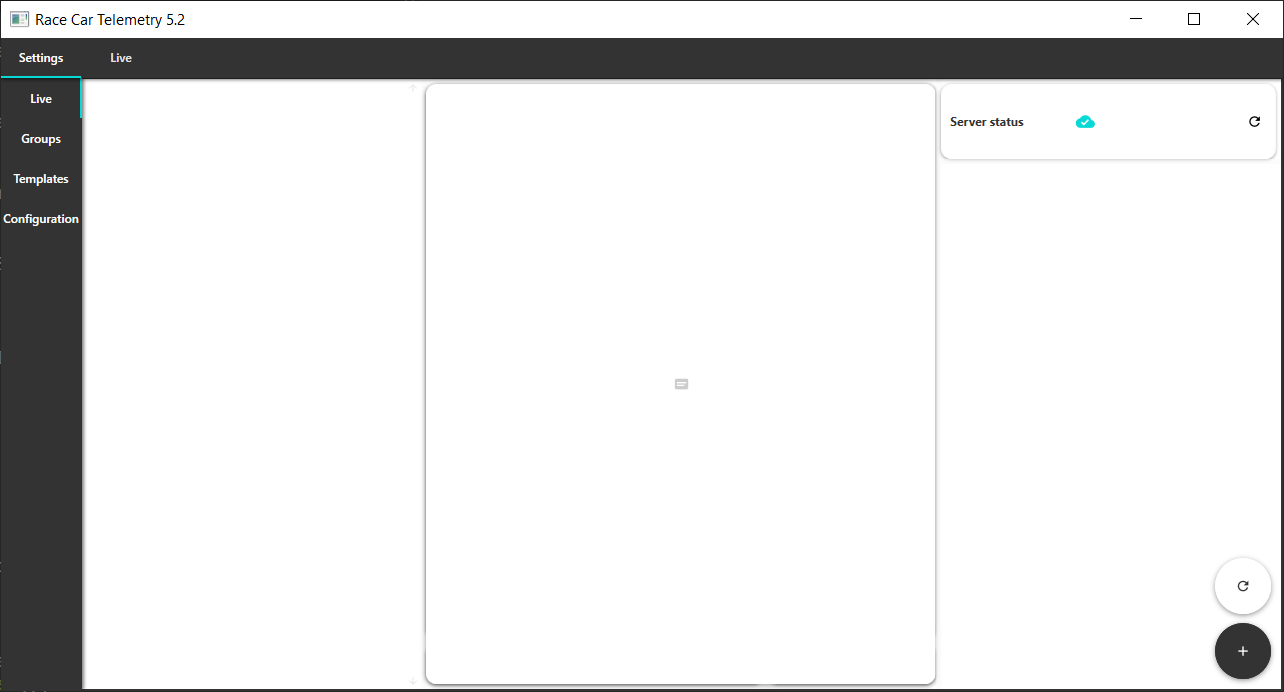
A felhasználói felület kettő fő részből áll. Egy felül található menüsorból és az alatta található tartalmi részből. Kettő fő menü található a menüsorban. **Settings** és **Live**.

A picture containing shape

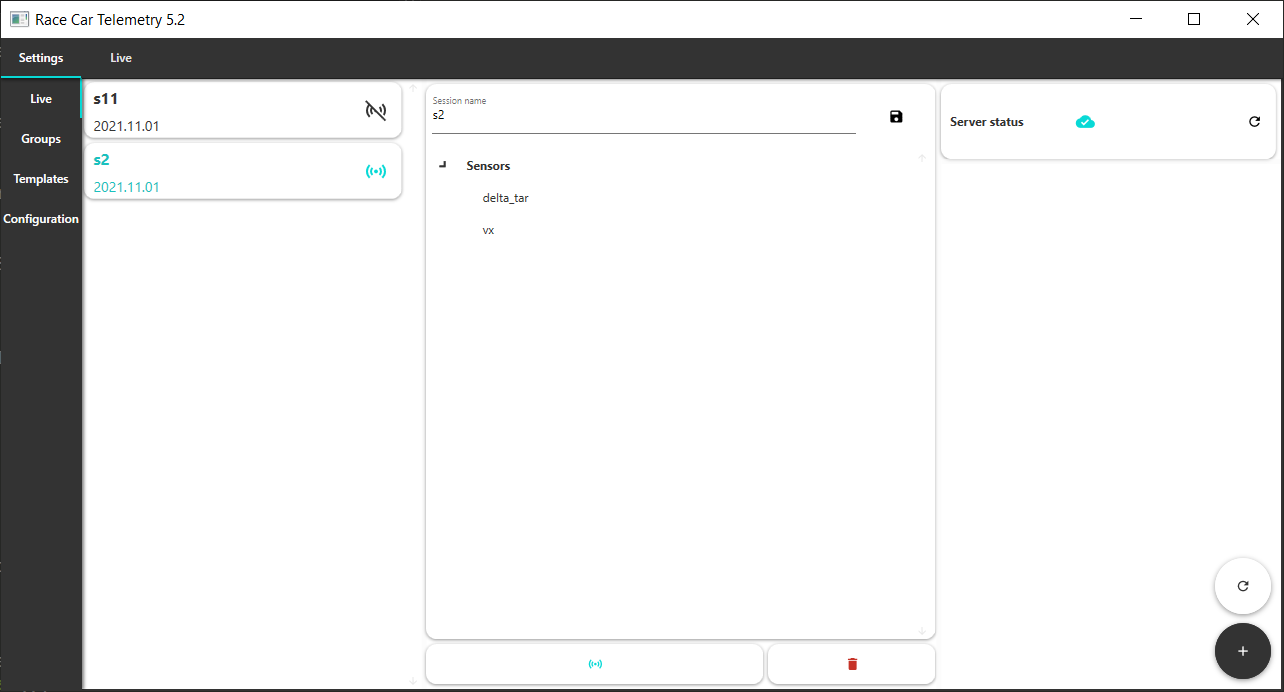
Description automatically generated

#### Settings menü

Ebben a menüben található az alkalmazás összes beállítása, pont ezért ennek a menünek van egy saját al menüje ami a képernyő bal oldalán található. Itt négy al menüpont található.



Live, Groups, Templates és Configuration.



#### Live settings menü

Ebben a menüben találhatóak az élő telemetriához kapcsolatos beállítások. A program indulásakor ez a menü látszódik alapból. Ez az oldal három részből áll, ami három oszlopként szerepel. Az első oszlopban az eddig létrehozott session-ök találhatóak. A második oszlopban az éppen kiválasztott session tulajdonságai látszódnak. A harmadik oszlopban pedig egy általános képet kapunk a rendszerről. Láthatjuk a kapcsolat státuszát és a szerver státuszát.

Középen lehet a kiválasztott session nevét állítani, illetve alatta egy fanézetben felsorolja a felület a sessionhöz tartozó szenzorokat. Alul található kettő gomb. A bal oldalival lehet az aktuális session státuszát állítani, azaz, ha offline akkor live-ra, ha live akkor pedig offline-ra. A jobb oldali törlés gombbal pedig ki lehet törölni a session-t. Utóbbi kettőnél egy popup ablak nyílik meg ahol a program megkérdezi, hogy biztosak vagyunk-e a döntésben. Ez a felhasználói élménybe segít be, mivel semmi képpen sem szeretnénk véletlenül kitörölni egy sessiont vagy élőről véletlenül offline-ra állítani.

Graphical user interface, text, application, chat or text message

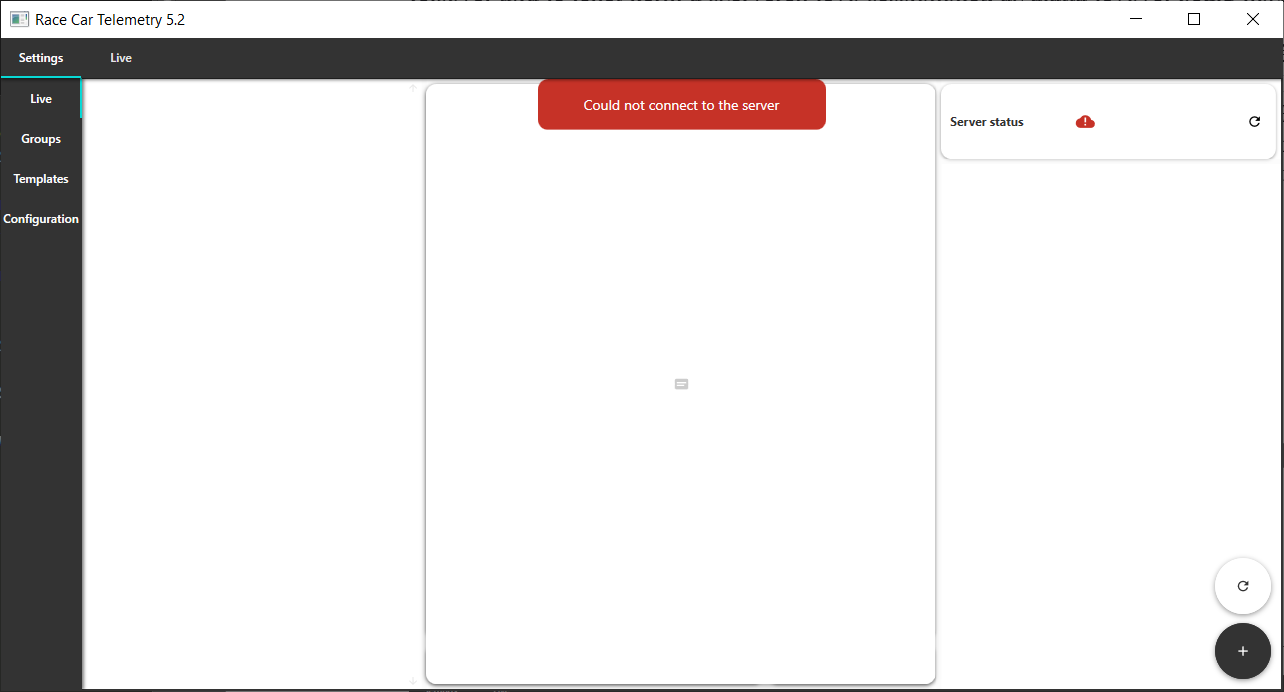
Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generated

A képernyő jobb alsó sarkában található kettő úgynevezett *floating action button*-s. A felsővel újra le lehet kérni a szerveren lévő sessionöket, az alatta lévővel pedig hozzá lehet adni egy újat.

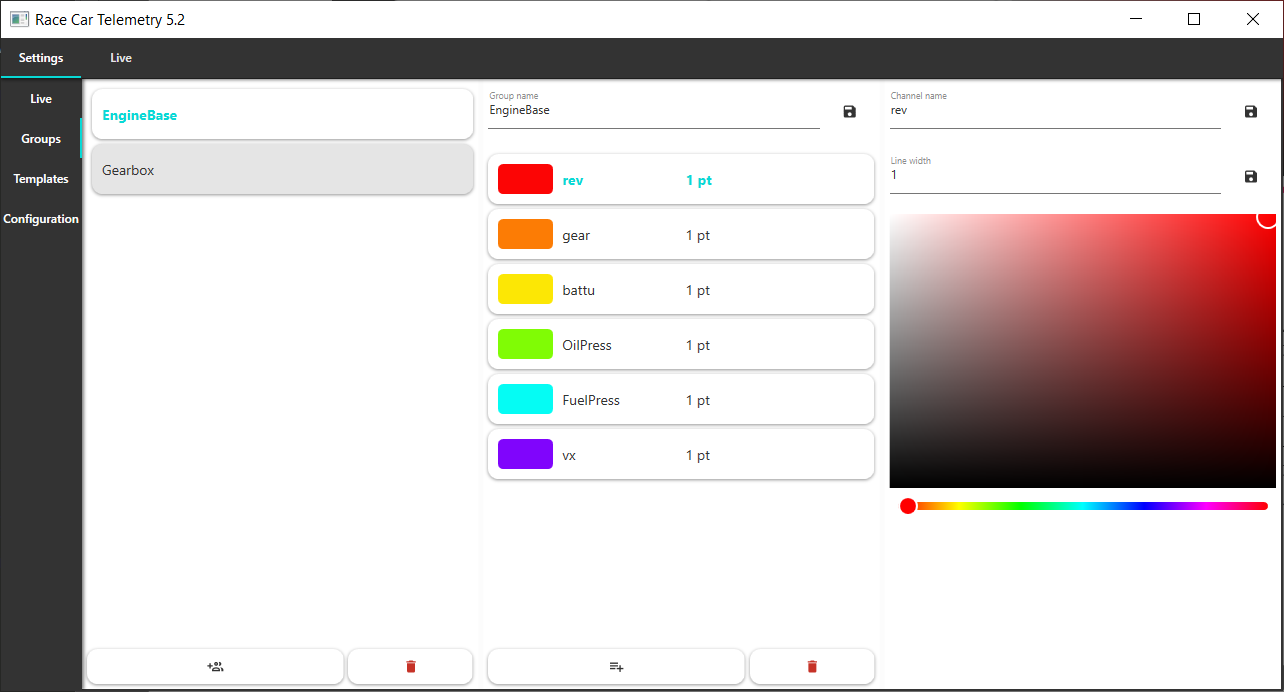
Abban az esetben, ha nem lenne kapcsolat a szerverrel, egy snackbar jelenik meg a képernyő tetején. Az egész alkalmazásban minden üzenet egy snackbarban jelenik meg. Ha nincs kapcsolat a szerverrel, azt jobb oldalon a server status mellett lévő kis ikon is jelzi. Ha csak azt szeretnénk megnézni, hogy van e már kapcsolat a szerverrel, a server status sor végén lévő újratölt gomb megnyomásával tehetjük meg. Ilyenkor a health check api hívást hívja a program.



#### Groups settings menü

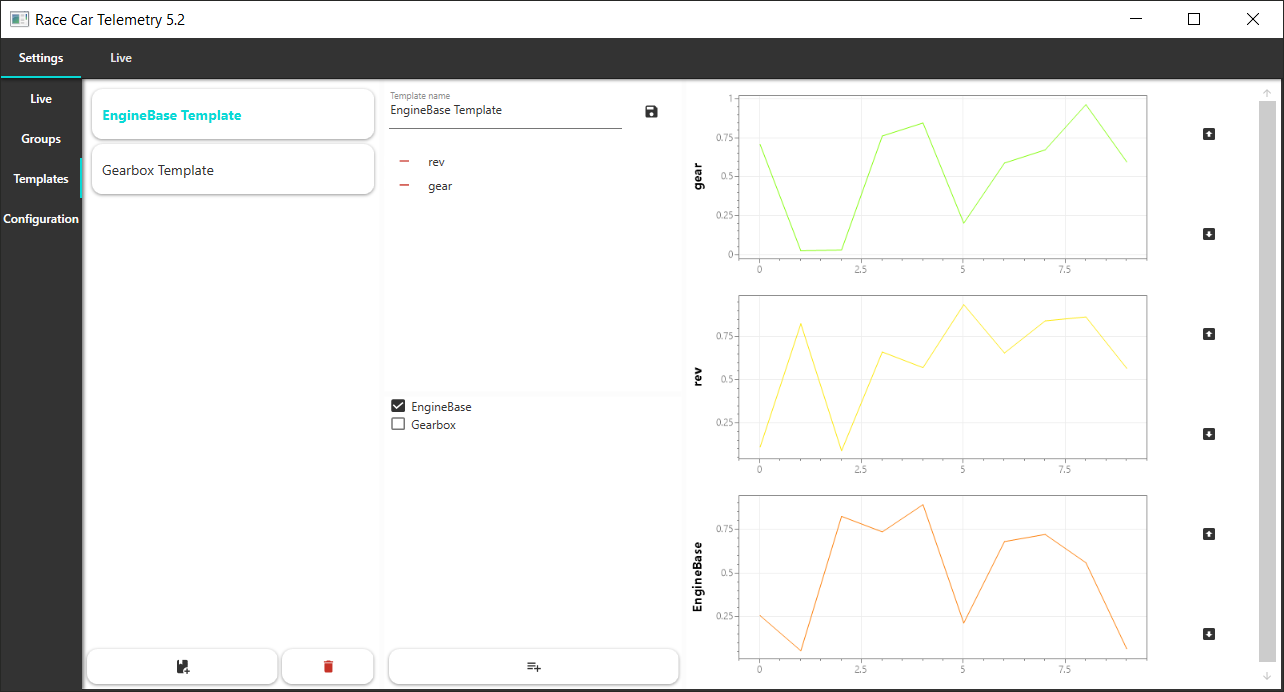
Ebben a menüben találhatóak a csoportok. Ennek az oldalnak is hasonló a felépítése. Három oszlopra van bontva. Az első oszlopban az elmentett csoportok találhatóak. Ezeket a program fájlból olvassa be. Az a csoport, ami kicsit ki van szürkítve azt jelzi, hogy nincsen hozzá elmentve szenzor.

A középső oszlopban találhatóak az éppen kiválasztott csoport beállításai. Felül lehet állítani a csoport nevét, alatta pedig fel vannak sorolva a csoporthoz tartozó szenzorok. Ezt a mintát követve az utolsó oszlopban az éppen kiválasztott szenzornak a beállításai láthatóak. Itt lehet beállítani a szenzor diagrammokon lévő megjelenését. Vonalvastagság és szín. Minden beállítás szerkesztése után rögtön frissülnek a diagrammok.



#### Templates settings menü

Ebben a menüben találhatóak a **Live** menühöz előre elmentett sablonok. Itt azt lehet beállítani, hogy milyen csoportok milyen sorrendben jelenjenek meg. Ezeket természetesen később lehet élesben is állítani, ez csak egy sablon. Az előző menühöz hasonlóan itt is három oszlopra van osztva az oldal és mindig a kiválasztott sablon beállításait láthatjuk. Középen be lehet pipálni a megjelenítendő csoportokat, jobb oldalon pedig ezeknek a sorrendjét lehet állítani. A felhasználói élmény szempontjából véletlenszerű adatokkal jelenítem meg a diagrammot, nem pedig csak a nevét rakom oda.



#### Configurations menü

Ebben a menüben található minden olyan beállítás, ami ahhoz kell, hogy elérjük a web api-t. A configurations.json fájl tartalmát láthatjuk ezen a felületen. Ez nagyon hasonlít az adatküldő modul konfigurációs fájljához, csak itt nincsenek api hívások. Azok egy külön fájlban vannak, az api\_configurations.json fájlban. Amint itt változtatunk valamit és a mentés gombra kattintunk, rögtön az új konfigurációval fog tovább üzemelni az alkalmazás.

Graphical user interface, text, application

Description automatically generated

#### Live menü

Miután kiválasztottunk egy session-t, a **Live** menüben megjelennek a szenzor adatai és a hozzá tartozó szenzorok. Miután kiválaszottuk a szenzorokat és csoportokat, a play gombra kattintva elkezdhetjük az adatok lekérését.

Shape, rectangle

Description automatically generated

A picture containing table

Description automatically generated

Amikor még nem kértünk le adatot, de kiválaszottunk már csoportot vagy szenzort, akkor lehet látni, hogy jobb oldalt kiszürkítve vannak a szenzorok. Miutána lekértünk adatot, csak azok lesznek kiszínezve, amikhez van adat.

Chart

Description automatically generated

Ez a menü kettő részre van osztva. Az első oszlopban legfelül található egy legördülő menü, ahol a sablonokat lehet kiválasztani. Alatta a kiválasztott szenzor neve, dátuma és státusza látható. Alatta az első kártyán a szenzorok, a másodikon pedig a csoportok vannak felsorolva checkboxok formájában. Legalul pedig az a gomb található, amivel az adatok lekérését lehet irányítani. Kettő állapota van ennek a gombnak. Play, pause. Értelem szerűen a play állapotban jönnek az adatok, pause állapotban pedig nem.

A második oszlopban találhatóak maguk a diagrammok.

Minden egyes diagram egy **UserControl**, ami egy újra felhasználható sablont jelent. Ennek a **UserControlnak** **Chart** a neve, amiben bal oldalon található a diagramm, jobb oldalon pedig a diagrammhoz tartozó szenzorok. A diagrammhoz a **ScottPlot** nevű külső könyvtárat használtam. Minden itt található szenzor egy újabb user control, ami megjeleníti a szenzor színét, nevét és az aktuális értékét.

A diagrammok alatt látható kettő csúszka. Mind a kettővel az adatok megjelenítését lehet szabályozni. A felső egy range slider, ahol a minimum és maximum indexet lehet megadni. Az alsó csúszkával pedig a diagrammokon látható kék szaggatott vonalat lehet mozgatni. Ez mutatja a jobb oldalon található szenzorok aktuális értékét. Mind a kettő csúszka mellett bal oldalon található egy egy gomb, amivel azt lehet szabályozni, hogy az adott csúszka együtt mozogjon-e az élő adatokkal vagy statikus legyen. Ha be van kapcsolva, azaz világos kék lesz, akkor dinamikusan mozog, ellenkező esetben statikus.

## Program felépítése

A szoftvert a három rétegű modell szerint készítettem el. Alulról felfele: adatelérési réteg, logikai réteg, megjelenítési réteg. A programban ezeket **DataAccess**, **BusinessLogic** és **UI** néven neveztem el.

Diagram

Description automatically generated

### Adatelérési réteg

Ez a réteg felelős az adatok feldolgozásáért, legyen az az adatok beszerzése vagy elküldése. Ebben a rétegben olvasom be fájlból a konfigurációs fájlt, illetve azokat a fájlokat, amik szükségesek a program működéséhez, mint például a csoprotok (groups) és az oldal sablonok (page templates).

Ezek mellett ebben a rétegben történnek meg az api hívások, amik a versenyautóból származó szenzorok, illetve a hozzá tartozó adatok elérését jelentik. Ebben a rétegben nem szűröm meg az adatokat, itt csak az adatok elérése van biztosítva.

Minden data access osztályt a **BaseDataAccess** osztályból kell származtatni, ahol olyan függvények vannak definiálva, amik szükségesek lehetnek a fájl olvasáshoz vagy az api hívásokhoz.

A fájl beolvasásához az alábbi függvényeket definiáltam:

* CheckFile
* IsFileExists
* IsFileEmpty

Ezek mindegyike egy logikai értékkel tér vissza és egy hibaüzenettel, ami később a Snackbar-ban és a log fájlban fog megjelenni. Mind a három függvény virtuális, azaz tetszés szerint ki lehet egészíteni a származtatott osztályban, illetve a láthatósági szintek protected. Erre egy példát az alábbi kódrészletben láthatunk.

protected virtual bool IsFileEmpty(string fileName, out string errorMessage)

{

if (new FileInfo(fileName).Length == 0)

{

errorMessage = $"'{fileName}' is empty";

return false;

}

else

{

errorMessage = string.Empty;

return true;

}

}

Minden api hívás típushoz definiáltam egy függvényt.

* CallGetApi
* CallPutApi
* CallPostApi
* CallDeleteApi

Ezek a hívások is protected virtual-ok és emellett aszinkronok. Mindegyiknek az első kettő paramétere a http kliens és az api hívás neve. Emellett a put és postnak a harmadik paramétere egy object ami a frissítendő vagy új érték.

A **CallGetApi**-nak még a visszatérési értékét is meg kell adni a hívásánál, mivel az alapján fogja a válaszként kapott json stringet deszerializálni.

protected virtual async Task<T> CallGetApi<T>(HttpClient client, string apiCall)

{

try

{

HttpResponseMessage response = await client.GetAsync(apiCall).ConfigureAwait(false);

ConfiguredTaskAwaitable<string> result = response.Content.ReadAsStringAsync().ConfigureAwait(false);

string resultString = result.GetAwaiter().GetResult();

return JsonConvert.DeserializeObject<T>(resultString);

}

catch (Exception exception)

{

throw exception;

}

}

A get híváson kívül mindegyik függvény egy számmal tér vissza, ami a hívás http result code-ja.

Lehet látni, hogy mindegyik hívás szinte ugyan az. Mindegyik egy try catch-ben van és maga a hívás csak a http kliensen meghívandó függvényben tér el.

protected virtual async Task<int> CallPostApi(HttpClient client, string apiCall, object value)

{

try

{

HttpResponseMessage response = await client.PostAsJsonAsync(apiCall, value).ConfigureAwait(false);

ConfiguredTaskAwaitable<string> result = response.Content.ReadAsStringAsync().ConfigureAwait(false);

int resultCode = int.Parse(result.GetAwaiter().GetResult());

return resultCode;

}

catch (Exception exception)

{

throw exception;

}

}

Diagram

Description automatically generated

### Logikai réteg

Ez a réteg köti össze az adatelérési és a megjelenítési réteget. Ebben a rétegben történik az adatok feldolgozása. Az én esetemben legtöbbször ez a réteg csak tovább hív az adatelérési rétegbe.

Ezeknek az osztályoknak nincsen közös ősük.

Diagram, text

Description automatically generated

### Megjelenítési réteg

Ez a réteg felelős a felhasználói felületért. Az adatok eléréséhez egyszerűen csak behív a logikai rétegbe, az pedig visszaadja azokat az adatokat, amik már készek a megjelenítésére.

Az alábbi szekvencia diagrammon a **LiveSettings** menü betöltése látható. Amikor ez az oldal legenerálódik, beolvassa a konfigurációs fájlt és az alapján megnézi, hogy van e kapcsolat a szerverrel.

Chart

Description automatically generated with low confidence

Az alábbi képen pedig a session-ök frissítése látható.

A screenshot of a computer

Description automatically generated with medium confidence

Az alábbi képen az adatok lekérése látható.

Graphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

A felhasználói felület kialakításához a Material Design nevű külső könyvtárat használtam, ami nem csak szép, hanem előre el vannak készítve olyan kontrollok, amik nagyban segítették a fejlesztést. Emellett én is létrehoztam saját kontrollokat. Ezek a **CustomControls** mappában találhatók.

Létrehoztam különböző gombokat, például olyat, amin szöveg helyett egy ikon van, illetve ennek mintájára készítettem külön delete gombot is, aminek fixen megvan adva a háttérszíne és az ikonja.

Saját kontrol még például az interaktív kártya. Ezt a beállítások menüben használtam minden listában.

Graphical user interface, application

Description automatically generated

Azért volt hasznos ilyeneket elkészítenem, mivel például az interaktív kártyánál csak egyszer kellett definiálnom azt, hogy amikor fölé visszük az egeret kicsit elszürkülkön a háttere, amikor rákattintunk akkor mégszürkébb legyen, illetve amikor ki van választva, a rajta található szövegek ki legyenek vastagítva, illetve világoskékek legyenek.

Az előző képen látható **AttributeCard** kódja itt látható. Egy ilyen custom control kettő részből áll.

Egy XAML és egy codebehind fájlból, ami természetesen C#. Mivel ez egyfajta gomb, ezért a Button osztályból származtattam. Emellett bindig-al kell megoldani azokat az attribútumokat, amiket kívülről lehet szerkeszteni. Az alábbi kódrészleten ennek a konstruktora és egy bindig látható.

public class AttributeCard : Button

{

static AttributeCard()

{

DefaultStyleKeyProperty.OverrideMetadata(typeof(AttributeCard),

new FrameworkPropertyMetadata(typeof(AttributeCard)));

}

public static readonly DependencyProperty AttributeNameProperty =

DependencyProperty.Register(nameof(AttributeName),

typeof(string),

typeof(AttributeCard),

new PropertyMetadata(string.Empty));

public string AttributeName

{

get => (string)GetValue(AttributeNameProperty);

set => SetValue(AttributeNameProperty, value);

}

A stílus része a **Themes/Generic.xaml** fájlban található. Ebben a fájlban van az összes saját kontrol design része. Az alábbi kódrészletben ezt láthatjuk.

<Style TargetType="{x:Type local:AttributeCard}">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type local:AttributeCard}">

<materialDesign:Card UniformCornerRadius="{DynamicResource CardButtonCornerRadius}"

materialDesign:ShadowAssist.ShadowDepth="Depth1"

Background="{TemplateBinding Background}"

FontSize="{DynamicResource ButtonFontSize}">

<Grid Margin="5">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="65"/>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="65"/>

</Grid.ColumnDefinitions>

<materialDesign:Card Grid.Column="0"

Margin="5"

UniformCornerRadius="5"

Background="{TemplateBinding AttributeColor}"

materialDesign:ShadowAssist.ShadowDepth="Depth0"

BorderThickness="0"/>

<Label Grid.Column="1"

Content="{TemplateBinding AttributeName}"

Foreground="{TemplateBinding LabelColor}"

VerticalContentAlignment="Center"

HorizontalContentAlignment="Left">

</Label>

<Label Grid.Column="2"

Content="{TemplateBinding LineWidth}"

Foreground="{TemplateBinding LabelColor}"

VerticalContentAlignment="Center"

HorizontalContentAlignment="Left"/>

</Grid>

</materialDesign:Card>

<ControlTemplate.Triggers>

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Background" Value="{DynamicResource AccentButtonMouseOver}"/>

<Setter Property="Cursor" Value="Hand"/>

</Trigger>

<Trigger Property="IsMouseOver" Value="False">

<Setter Property="Background" Value="{DynamicResource AccentButtonMouseLeave}"/>

<Setter Property="Cursor" Value="Arrow"/>

</Trigger>

<Trigger Property="IsMouseCaptured" Value="True">

<Setter Property="Background" Value="{DynamicResource AccentButtonMouseDown}"/>

</Trigger>

<DataTrigger Binding="{Binding RelativeSource={RelativeSource Mode=Self},

Path=(local:AttributeCard.IsSelected)}" Value="True">

<Setter Property="FontWeight" Value="Bold"/>

</DataTrigger>

<DataTrigger Binding="{Binding RelativeSource={RelativeSource Mode=Self},

Path=(local:AttributeCard.IsSelected)}" Value="False">

<Setter Property="FontWeight" Value="Normal"/>

</DataTrigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

Itt kihasználtam a WPF egyik nagy erősségét, mivel nem kellett kódban foglalkoznom az egér eseményekkel meg a kártya színezésével, hanem mindent az xaml fájlban csináltam. Ehhez létrehoztam egy stílust, aminek a target type-ja az **AttributeCard**. Ezen belül létrehoztam egy control template-et és itt készítettem el a desing-t. Van egy kártya, amiben van egy grid aminek van 4 oszlopa. Az első oszlopban van egy másik kártya, ami az attribútum színét jelöli. A második oszlopban az attribútum neve található, a következőben pedig a vonalvastagsága.

A control template triggers részében pedig megadtam a megfelelő eseményre a megfelelő dolgok színezését.

Ha a mouse over igaz, akkor a hátteret színezem és a kurzor-t állítom át egy kézre, ellenkező esetben visszaszínezem eredeti színre és a kurzort is visszaállítom.

A klikk eseménynél szintén átszínezem a hátteret viszont ebben az esetben egy kicsit szürkébbre.

A triggerek után láthatóak data triggerek, ahol jelen esetben azt bindolom, hogy az aktuális interaktív kártya ki van e választva. Ha igen, akkor a szövegeket kivastagítom.

A bindolásnak köszönhetően roppant egyszerűen tudom változtatni a kártya állapotát, azaz azt, hogy ki van e választva vagy nem. Az alábbi kódrészleten ezt láthatjuk.

foreach (GroupSettingsItem item in GroupsStackPanel.Children)

{

item.ChangeColorMode(item.Id == activeGroupId);

}

Meghívom a **ChangeColorMode** függvényét a control-nak, ahol pedig csak az **IsSelected** property-t változtatom. Mivel ez hozzá van kötve a design-hoz, ezért automatikusan átállítódik a szöveg megjelenése.

Emellett más hasznos saját kontrolokat is létrehoztam például az **EmptyPanel**-t. A programban sok helyen amikor még nincs betöltve semmilyen adat, egy halványszürke ikon látható. Ezt ezzel oldottam meg. Ez tulajdonképpen egy grid, amiben van egy középreigazított ikon. Emellett bindolva van az ikon.

public void ChangeColorMode(bool change)

CardItem.IsSelected = change;

}

<Style TargetType="{x:Type local:EmptyPanel}">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type local:EmptyPanel}">

<Grid Background="{DynamicResource MaterialDesignBackground}">

<materialDesign:PackIcon Kind="{Binding Kind, RelativeSource TemplatedParent}}"

Foreground="{DynamicResource FontColor}"

VerticalAlignment="Stretch"

HorizontalAlignment="Stretch"

Opacity="{DynamicResource EmptyPanelOpacity}"/>

</Grid>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

A custom controllokat elérhetjük az xaml fájlokból és a C# fájlokból is egyaránt. Az xaml fájlokban ehhez a **UserControl** tag-ben definiálni kell egy nevet, amin keresztül el tudjuk érni azt a namespace-t, amiben a custom controllok vannak. Utána, mint bármlyik másik WPF controll-t, ezt is elérhetjük.

A codebehind fájlban ezt hasonlóan kell csináni. Usingoljuk a namespace-t a könyebb programozás érdekében, majd példányosítjuk a custom controlt mint egy osztályt, mivel ez egy osztály.

using UI.UserControls.Groups;

...

GroupSettingsItem groupSettingsItem = new GroupSettingsItem(group, new ChangeActiveGroup(UpdateGroups));

<UserControl x:Class="UI.UserControls.Live.LiveMenu"

...

xmlns:custom="clr-namespace:UI.CustomControls">

...

<custom:ImageCardButton x:Name="PlayImageCardButton"

Grid.Row="6"

Margin="5,0,0,5"

Kind="Play"

Click="PlayImageCardButton\_Click" />

## Színek és stílusok

A telemetria alkalmazásban próbáltam a lehető legszebb és legmodernebb felhasználói felületet létrehozni. Ehhez előre definiáltam színeket és stílusokat. Ezeket a **Themes/UiDesign.xaml** fájlban tettem meg, így azt később a kód bármelyik részéről el tudtam érni. Ez több szempontból is jó, legfőképpen azért, mert ha változtatnám az egyik színt, csak egy helyen kell ezt megtennem.

A programban négy fő színt használok. Elsődleges, másodlagos, hiba és háttér szín.

Elsődleges szín: #08D9D6

Másodlagos szín: #333333

Hiba szín: #c63227

Háttér szín: fehér

Az elsődleges színnel jelölöm azokat a részeket, amik éppen ki vannak jelölve. Legyen az menü vagy egy elem egy lista nézetben.

Másodlagos színnel láttam el a menüsörök hátterét, a gombok előterét, a szövegeket és a felugró üzenet hátterét.

A hiba színnel a felugró hibaüzenet hátterét és azoknak a gomboknak az előterét jelöltem, amikkel valamit törölni vagy elutasítani lehet.

<Color x:Key="Primary">#08D9D6</Color>

<Color x:Key="Secondary">#333333</Color>

<Color x:Key="Accent">#c63227</Color>

<SolidColorBrush x:Key="Delete" Color="{DynamicResource Accent}" po:Freeze="True" />

<SolidColorBrush x:Key="SelectedBackground" Color="{StaticResource Primary}" />

<SolidColorBrush x:Key="NotSelectedBackground" Color="{StaticResource White}" />

<SolidColorBrush x:Key="Selected" Color="{StaticResource PrimaryDark}" />

<SolidColorBrush x:Key="NotSelected" Color="{StaticResource Secondary}" />

Stílusok terén olyanokat hoztam létre, amikkel elsősorban globálisan érek el minden kontrollt.

<Style TargetType="materialDesign:Card">

<Setter Property="UniformCornerRadius" Value="10"></Setter>

</Style>

<Style x:Key="Font" TargetType="Label">

<Setter Property="FontFamily" Value="{StaticResource OswaldRegular}"/>

</Style>

<Style BasedOn="{StaticResource {x:Type materialDesign:SnackbarMessage}}"

TargetType="{x:Type materialDesign:SnackbarMessage}">

<Setter Property="HorizontalContentAlignment" Value="Center" />

</Style>

Az alábbi kódrészleten látható, ahogy az összes kártyára 10-es sarok lekerekítést alkalmazok, ahogy az összes címkére beállítom a betűtípust és ahogy a felugró üzenet szövegét középre igazítom.

Mind a színeket és a stílusokat el lehet érni C# oldalon is és a programban ezt alkalmazom is. A **Managers/Colors/ColorManager.cs** osztályban definiáltam erre a célra property-ket, amik ezeket érik el. Ez egy statikus osztály, tehát példányosítás nélkül elérhető a tartalma. Az xaml fájlban definiált színek és stílusok az **Application.Current.Resources** dictionary-ban találhatóak.

public static Color Secondary => (Color)Application.Current.Resources["Secondary"];

public static Color Primary => (Color)Application.Current.Resources["Primary"];

## Validáció

A telemetria alkalmazásban lévő összes beviteli mezőn van egyfajta validáció.

A validáció legnagyobb előnye, hogy amikor használni szeretnénk a bemeneti mezőben lévő értéket, például el szeretnénk menteni, akkor csak a helyes értéket adja tovább függetlenül attól, hogy a bemeneti mezőben hibás érték szerepel.

Shape, rectangle

Description automatically generatedAz alábbi ábrán az IP cím bemeneti mezője látható, amiben hibás érték szerepel.

### Validáció létrehozása

A validációkhoz először is létrehoztam egy **FieldsViewModel** osztályt, amit az **INotifyPropertyChanged** osztályból származtattam. Majd ebben az osztályban létrehoztam minden egyes input field-hez egy property-t. A property set részében meghívódik a **MutateVerbose** extension method, ami összehasonlítja az új értéket a régivel.

private string? ipAddress;

public string? IpAddress

{

get => ipAddress;

set => this.MutateVerbose(ref ipAddress, value, RaisePropertyChanged());

}

Magához a validációhoz egy **ValidationRule**-t kell létrehozni. Ezt úgy lehet megtenni, hogy származtatunk egy osztályt a **ValidationRule**-ból, ahol pedig a **Validate** függvényt kell felülírni, ami maga a ValidationResult-ot fogja visszaadni.

A paraméterként kapott value lesz az input mező értéke. Az alábbi példában az IP cím validációja látható. Először megvizsgálom, hogy üres e. Ha igen akkor visszatérek egy hibás validációval, aminek üzenetként beadom, hogy ez a mező kötelező. Ellenkező esetben tovább vizsgálom az IP címet. Azt, hogy valós IPv4 cím-e, egy Regex-el vizsgálom meg. Ha ez is elhal, akkor szintén hibás validációval térek vissza, ellenkező esetben pedig jóval.

Ezen a validáción kívül használok még **NotEmptyValidationRule** és **IsNumberValidationRule**-t. Ezeket szintén én írtam.

Ahhoz, hogy egy ilyen validációt használjunk, hozzá kell kötnünk az adott bemeneti mezőhöz, illetve C# oldalon össze kell kötni a validáló modellt binding segítségével.

public class IsIpAddressValidationRule : ValidationRule

{

public override ValidationResult Validate(object value, CultureInfo cultureInfo)

{

if (string.IsNullOrWhiteSpace((value ?? "").ToString()))

{

return new ValidationResult(false, "Field is required");

}

else

{

if (Regex.IsMatch(value.ToString(), @"^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]).){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$"))

{

return ValidationResult.ValidResult;

}

else

{

return new ValidationResult(false, "Invalid format");

}

}

}

}

Az alábbi kódrészletben az eddig említett IP cím beviteli mezője látható. Ez egy egyszerű **TextBox**. A **TextBox** Text részébe kell bindig-al összekötni a **FieldsViewModel** osztályban létrehozott property-vel. Binding Path="IpAddress

<TextBox Grid.Row="0"

Name="IpAddressTextBox"

VerticalAlignment="Stretch"

HorizontalAlignment="Stretch"

VerticalContentAlignment="Center"

HorizontalContentAlignment="Left"

materialDesign:HintAssist.Hint="IP address"

materialDesign:HintAssist.IsFloating="True"

Margin="5,0,0,15">

<TextBox.Text>

<Binding Path="IpAddress"

UpdateSourceTrigger="PropertyChanged">

<Binding.ValidationRules>

<validation:IsIpAddressValidationRule ValidatesOnTargetUpdated="True" />

</Binding.ValidationRules>

</Binding>

</TextBox.Text>

</TextBox>

Ebben a tag-ben pedig hozzá kell adni azt, hogy melyik validációs szabály szerint működjön.

<validation:**IsIpAddressValidationRule** ValidatesOnTargetUpdated="True" />

Ehhez a xaml fájl elején hivatkozni kell a validációs szabály namespace-t és a bindigot.

xmlns:validation="clr-namespace:UI.ValidationRules"

d:DataContext="{d:DesignInstance validation:FieldsViewModel, IsDesignTimeCreatable=False}"

A codebehind fájlban a következőket kell megtenni.

Példányosítani kell a **FieldsViewModel** osztályt és **DataContext**-be ezt meg kell adni.

private readonly FieldsViewModel fieldsViewModel;

...

fieldsViewModel = new FieldsViewModel();

DataContext = fieldsViewModel;

Később a kódban pedig a **fieldsViewModel** változóban lévő értékkel dolgozok tovább. Az elején oda töltöm be az értéket és a végén pedig onnan mentem ki.

fieldsViewModel.IpAddress = ConfigurationManager.Configuration.Url;

fieldsViewModel.Port = ConfigurationManager.Configuration.Port;

IsHttpsToggleButton.IsChecked = ConfigurationManager.Configuration.IsHttps;

fieldsViewModel.TimeOut = ConfigurationManager.Configuration.TimeOut;

fieldsViewModel.WaitBetweenCollectData = ConfigurationManager.Configuration.WaitBetweenCollectData;

...

ConfigurationManager.Configuration.Url = fieldsViewModel.IpAddress;

ConfigurationManager.Configuration.Port = (int)fieldsViewModel.Port;

ConfigurationManager.Configuration.IsHttps = (bool)IsHttpsToggleButton.IsChecked;

ConfigurationManager.Configuration.TimeOut = (int)fieldsViewModel.TimeOut;

ConfigurationManager.Configuration.WaitBetweenCollectData = (int)fieldsViewModel.WaitBetweenCollectData;

# Tesztek

A szoftverfejlesztés egyik legfontosabb eleme a tesztelés. A tesztelésnek nem az lényege, hogy hibákat keressünk a kódban, hanem hogy kiküszöböljük azokat a problémákat, amik a felhasználónál felmerülhetnek. Például olyan problémák, amik a felhasználói élményt befolyásolják, például rendszeresen megakad a program bizonyos esetekben. Vagy a program adatkezelési logikáját érintő probléma, ami ritkán lép fel, de az sok adatnál már nem megengedhető.

Több tesztelési módszer létezik. Ezeket végezheti külön erre a célra képzett ember, azaz a tesztelő vagy lehet automatizált, amit már számítógép végez.

Egy szoftver tesztelését kettő szempontból lehet megközelíteni. Vagy értjük a tesztelendő modul működésének a logikáját vagy nem. Ha értjük, hogy pontosan mi történik akkor azt white-box tesztelésnek hívjuk, ha nem akkor black-box-nak.

Mind a kettő esetnek van értelme és mind a kettő esettel kell tesztelni. A fekete dobozos tesztelést mindig olyan ember végzi, aki nem vett részt az adott modul lefejlesztésében tehát olyan valaki, aki nem tudja, hogy milyen logika alapján működik és nem is érdekli. Itt ő csak a különböző bemenetekből kapott kimeneteket vizsgálja. Az ilyen teszteléssel hamar ki lehet szűrni, ha valami nem működik jól.

A fehér dobozos tesztelésnél az illető érti a tesztelendő modul belső működését, illetve akár a forrás kód is rendelkezésére áll. Ilyenkor más megközelítésből is képes tesztelni, mégpedig pont azért, mert érti és látja a belső működést.

## Manuális tesztek

Erre a tesztelésre mindenképpen emberi tesztelő kell. A fő feladata, hogy megértse az adott modul működését viszont nem kell a forráskódot ismernie hozzá elég, ha csak a folyamatot megérti. Ennél a tesztelési fajtánál a tesztelő először is olyan adatokkal nyomkodja végig az alkalmazást, amivel általában találkozhat a tesztelendő modul majd megpróbálja ezt úgy is, hogy a program kiakadjon tehát vagy hibát dobjon, vagy rossz működést eredményezzen.

A szoftverfejlesztésben mielőtt még az adott fejlesztés bemenne a fő ágra, át kell esnie manuális teszten, amit szigorúan másik embernek kell végeznie, mint aki fejlesztette az adott modult.

Vannak végek, ahol ezt külön ember végzi, vannak, ahol nem. Ahol van külön tesztelő annak az az előnye, hogy erre lett kiképezve tehát érti a dolgát. A hátránya viszont az, hogy egyel több embert kell fizetni és ezt nem biztos, hogy megengedheti a cég. Ahol nincsen külön tesztelő ember ott a fejlesztők tesztelik egymás fejlesztéseit. Ennek az előnye az előbbi esetnek a hátránya, viszont a hátránya az, hogy így a másik fejlesztőnek idejébe kerül a tesztelendő modul megértésére és ez nem mindig jó a fejlesztés határidejét nézve.

## Unit tesztek

Ez egy automatizált tesztelési fajta, amit számítógép végez. Itt nem az az elsődleges cél, hogy hibákra bukkanjunk, hanem az, hogy a szoftvert karbantartsuk vele. Minden új modul hozzáadásával ezek a tesztek lefutnak és ezzel a fejlesztőknek van egy visszajelzésük, hogy a régi modulok még jól működnek-e az új modul hozzáadásával.

Text

Description automatically generated with low confidenceA saját rendszerem Windows alkalmazásában találhatóak unit tesztek, amiket össze is kötöttem a GitHub Actions-el. A unit tesztekhez az **NUnit** nevű frameworkot használtam, amit támogat a .NET. A tesztek egy **UnitTest** nevű projektben találhatóak. A Visual Studio ezt egy külön ikonnal megkülönböztetővé is teszi számunkra. Ezt az alábbi képen láthatjuk.

Olyan esetekre írtam unit teszteket, amikor egy fájlt kell feldolgozni. A programban lehet csoportokat, illetve oldal sablonokat elmenteni, ezeket a tesztek ezekre irányulnak.

Az egy modulba tartozó teszteket egy osztályba kell rakni, viszont az átláthatóság kedvéért én szétszedtem ezeket külön fájlokra. Ezt partial osztályok segítségével értem el. A partial osztály C#-ban egy olyan osztály fajta, amivel egy osztályt több fájlra szét lehet szedni, hol ott az ugyan az az osztály. A minta, amit követtem az az, hogy minden teszt modulban van egy fő fájl, amiben csak azok a dolgok vannak, amik minden másik teszthez kellhetnek. Például a csoportok tesztelésénél ez a fájl a **GroupTest.cs**, amiben azt a kettő mappának az elérési útvonalát tárolom, amikben a tesztelendő fájlok találhatóak. Ezek természetesen konstans változók, mivel az értékük nem fog sose változni. Ezt az alábbi képen láthatjuk.

Az **NUnit** tesztelő frameworkben minden modult a **[TestFixture]** attribútummal kell ellátni. Ez mondja meg a fordítónak, hogy ez egy olyan osztály, amiben tesztek vannak.

using NUnit.Framework;

using System.IO;

namespace UnitTest.Group

{

[TestFixture]

public partial class AddGroup

{

private const string GOOD\_GROUPS\_PATH = "../../../Group/good\_groups/";

private const string WRONG\_GROUPS\_PATH = "../../../Group/wrong\_groups/";

private const string TEST\_RESULT\_LOG\_PATH = "../../../test\_results.txt";

[TearDown]

public void TearDownAddGroup()

{

using StreamWriter writer = new StreamWriter(TEST\_RESULT\_LOG\_PATH, append: true);

writer.WriteLine($"{TestContext.CurrentContext.Result.Outcome}\t{TestContext.CurrentContext.Test.FullName}");

}

}

}

A unit tesztek lényege, hogy egy működést több bemenettel is vizsgáljuk. Ezeket test case-eknek hívjuk.

Ebben a frameworkben egy test modul egy függvény. Ezt a **[Test]** attribútummal kell ellátni. A fordító innen tudja, hogy ez egy test. A **[TestCase()]** attribútummal lehet test case-eket hozzáadni, aminek paraméterekben kell átadni azokat az adatokat amiket a teszt során szeretnénk használni. Ezeket a test függvényben is meg kell adni paraméterekként.

Azt, hogy a teszt sikeres volt-e, az **Assert** osztállyal kell elintézni. Az alábbi képen arra láthatunk példát, ahogy a business logic réteget használom fel a teszteléshez.

using NUnit.Framework;

namespace UnitTest.Group

{

[TestFixture]

public partial class AddGroup

{

[Test]

[TestCase(0, "proba\_group", 0, "proba\_group")]

[TestCase(1, "proba group", 1, "proba group")]

[TestCase(2, " proba group", 2, "proba group")]

[TestCase(3, "proba group ", 3, "proba group")]

[TestCase(4, " proba group ", 4, "proba group")]

public void CreateGroup\_TestGood(int id, string groupName, int expectedId, string expectedGroupName)

{

DataModel.Group group = new DataModel.Group(id, groupName);

Assert.AreEqual(group.Name, expectedGroupName);

Assert.AreEqual(group.Id, expectedId);

}

[Test]

[TestCase("proba\_group", "proba group")]

[TestCase("proba group", "proba\_group")]

[TestCase(" proba group", " proba group")]

[TestCase("proba group ", "proba group ")]

[TestCase(" proba group ", " proba group ")]

public void CreateGroup\_TestBad(string groupName, string expectedGroupName)

{

DataModel.Group group = new DataModel.Group(0, groupName);

Assert.AreNotEqual(group.Name, expectedGroupName);

}

}

}

A tesztek eredményeit a Visual Studio a Test Explorer ablakban grafikusan megjeleníti számunkra. Erre az alábbi képen láthatunk példát. Mutatja a test modulokban lévő test case-eket. Minden egyes test case-hez a futási időt is mutatja. A futási idő fontos szempont ezeknél a teszteléseknél, mivel a nagyobb rendszereknél az összes unit teszt lefuttatása akár fél órát is igénybe vehet, ha csak nem többet és ez is képes lassítani a fejlesztés menetét.

using BusinessLogic;

using NUnit.Framework;

using UI.Managers;

namespace UnitTest.Group

{

[TestFixture]

public partial class AddGroup

{

[Test]

[TestCase(GOOD\_GROUPS\_PATH + "groups.json")]

public void DeserializeJson\_AllGood(string fileName)

{

\_ = new GroupBusinessLogic().LoadGroups(fileName, out string errorMessage, ref GroupManager.LastGroupId, out GroupManager.TemporaryGroupIndex);

Assert.IsTrue(string.IsNullOrEmpty(errorMessage));

}

[Test]

[TestCase(GOOD\_GROUPS\_PATH + "grouops.json")]

public void DeserializeJson\_FileNotFound(string fileName)

{

\_ = new GroupBusinessLogic().LoadGroups(fileName, out string errorMessage, ref GroupManager.LastGroupId, out GroupManager.TemporaryGroupIndex);

Assert.IsFalse(string.IsNullOrEmpty(errorMessage));

}

[Test]

[TestCase(WRONG\_GROUPS\_PATH + "missing\_curly\_bracket.json")]

[TestCase(WRONG\_GROUPS\_PATH + "empty\_name.json")]

[TestCase(WRONG\_GROUPS\_PATH + "null\_name.json")]

[TestCase(WRONG\_GROUPS\_PATH + "null\_attribute\_name.json")]

[TestCase(WRONG\_GROUPS\_PATH + "null\_attribute\_color.json")]

[TestCase(WRONG\_GROUPS\_PATH + "empty.json")]

public void DeserializeJson\_Missing(string fileName)

{

\_ = new GroupBusinessLogic().LoadGroups(fileName, out string errorMessage, ref GroupManager.LastGroupId, out GroupManager.TemporaryGroupIndex);

Assert.IsFalse(string.IsNullOrEmpty(errorMessage));

}

}

}

A screenshot of a computer

Description automatically generated

Ha egy teszt hibásan fut le akkor azt mutatja nekünk a felület.

A screenshot of a computer

Description automatically generated

### Automatizált tesztelés

A unit tesztek tökéletesek arra, hogy automatizáljuk őket és beállítsunk olyan policyket amikkel a fejlesztést biztonságosabbá tehetjük a hibák elkerülése végett. A Git alapú alkalmazásokban lehetőségünk van olyan szabályokat beállítani, hogy bizonyos ágakra történő pusholás után fussanak le a unit tesztek és csak akkor engedi a felület bemergelni az ágat az egyel fentebbi ágra, ha a tesztek sikeresen lefutnak. Ezek mellett még számos más dologot be lehet állítani, ezekre a **[Git]** fejezetben fogok részletesebben kitérni.

Ezeket a teszteket kötöttem össze a GitHub Actions-el. Beállítottam, hogy a master és develop ágra történő pull requesteknél ezek fussanak le. Ezeknek a beállításoknak a konfigurációja a **.github/workflows/unit\_tests.yml** fájlban találhatóak. A workflow először futtat egy windows virtuális gépet, amin van .NET környezet. Majd lefordítja a **UnitTest** projectet és lefuttatja azt. A **UnitTest** project lefordításakor lefordul az egész tesztelendő program, tehát azt nem kell külön lefordítani.

A GitHub Actions is mutatja számunkra, hogy mennyi időbe telt a teszteket lefuttatni, illetve, hogy mennyi teszt futott le sikeresen. Az itt mutatott idő nem csak a tesztek futtatásának az ideje, hanem a teszt környezet előállításáé is. Az alábbi képen láthatjuk, ahogy mind az 57 teszt sikeresen lefutott kevesebb mint 2 másodperc alatt.

Graphical user interface, text, application, email

Description automatically generated

Text

Description automatically generated

# Git

A szoftverfejlesztés világában elengedhetetlen valamilyen Git alapú szoftver használása. A Git egy verziókezelő program, amivel a legkisebb programoktól a legnagyobb szoftverekig megkönnyíti a fejlesztés menetét. Ez a program önmagában még nem elég a nagyobb programoknál ezért jöttek létre olyan programok, amik egy felhasználói felület mellett más eszközökkel is támogatják a feljesztés menetét. A legnépszerűbb ingyenes ilyen alkalmazások a GitHub, GitLab, Bitbucket, SourceForge. Fizetősök közül az Azure DevOps-ot említeném meg ami a Microsoft-é.

Ezeknek az alkalmazásoknak a fő előnyük az, hogy egy közös helyen van tárolva a kód előzményekkel együtt, amit az összes fejlesztő bármikor elérhet.

## GitHub

Én ezek közül a dolgozatom elkészítéséhez a GitHub-ot használtam. A GitHub minden olyan eszközzel rendelkezik, ami egy ilyen nagyságú szoftver elkészítéséhez szükséges.

### Ágak

Graphical user interface, text, application, chat or text message

Description automatically generatedA fejlesztést álltalában három ágon végeztem. A master ág a fő ág, amin csak “production quality” kód van, tehát olyan kód ami már kész a kiadásra. Ebből az ágból ágazik a develop ág, amin a fejlesztés zajlik viszont nem ezen az ágon dolgoztam, hanem mindig az adott modul lefejlesztéséhez elágaztam egy új ággal, azon lefejlesztettem azt, majd bemergeltem a develeop ágra. Amikor úgy éreztem, hogy már a program van egy olyan szinten, amit mát ki lehet adni akkor ágaztam vissza a masterre.

Graphical user interface, text, application, chat or text message

Description automatically generatedAz alábbi kettő képen erre láthatunk példát. A pageTemplate ág ment be a develop-ra, ami pedig masterre. A jobb oldali képen láthatjuk azt a kis zöld pipát, ami a unit tesztek hibátlan lefutását eredményezik.

### Fejlesztés menete

Mivel egyedül dolgoztam a fejlesztésen, nem volt nagyon értelme egy szoftverfejlesztési módszert kiválasztani, viszont próbáltam a scrum-hoz tartani magam. A fejlesztést sprintekre bontottam fel és a lefejlesztendő dolgokat egy automatizált kanban stílusú táblán vezettem.

#### Taszkok

Erre a célra egy GitHub projektet hoztam létre, amihez az automatizált kanban sablont választottam ki. Három oszlop található. To do, In progress és Done.

A To do oszlopba gyűjtöttem a lefejlesztendő dolgokat. Ezeket GitHub Issue-kká alakítottam át mivel azokhoz lehet címkét rendelni és hozzá lehet rendelni egy mérföldkőhöz, amik az én esetemben az imént említett sprintek.

Az In progress oszlopba húztam át mindig az adott fejlesztéshez tartozó kártyát.

A Done oszlopba pedig a projekt tábla automatizálása végett, mindig automatikusan kerültek át a taszkok. A pull request létrehozásánál mindig hozzárendeltem az adott pull requesthez tartozó taszkokat és amikor bement a pull request, automatikusan átrakta Done-ba a taszkokat.

Graphical user interface, application, Teams

Description automatically generated

#### Címkék

Több címkét használtam fel a különböző lefejlesztendő taszkok megkülönböztetéséhez. A GitHub mutatja is, hogy az adott címkéhez mennyi Issue tartozik ezzel is megkönnyíti a fejlesztést.

Graphical user interface, text, application, email

Description automatically generated

#### Mérföldkövek

A fejlesztést sprintekre bontottam. Egy sprintnek egy teljes hétnyi időt adtam, ezzel is megtartva a fejlesztés sebességét. Ezekhez GitHub-on mérföldköveket hoztam létre és mindig az adott mérföldkőhöz rendeltem hozzá az adott sprintben lévő issue-kat.

Graphical user interface, text, application, email

Description automatically generated

# Rendszer telepítése

A rendszer három részből áll, és három különböző területen kell telepíteni. Az alábbi alfejezetekben említett telepítők mind elérhetőek a rendszer GitHub oldalán lévő Release-ek között.

<https://github.com/bricsi0000000000000/LiveTelemetry/releases>

## Adat küldő

Ennek a projektnek a scripts és configuration\_files mappáit kell átmásolni egy olyan eszközre, amin van python futtatási környezet.

## Web Api

A web api-hoz első sorban a publisholt állapotát kell egy olyan környezetben elhelyezni, ahol van .NET futtatási környezet és publikus internet elérés. Ez lehet egy Windows szerver vagy akár egy Linux szerver docker környezettel.

Ezután létre kell hozni az adatbázist, amit majd a rendszer használni fog. Ehhez kelleni fog egy MSSQL szerver és le kell futtatni az adatbázis migrációt. Ezt az alábbi parancsokkal lehet megtenni.

dotnet ef database update

## Telemetria alkalmazás

Ehhez egyszerűen csak el kell indítani a **RaceCarTelemetrySetup.exe** fájlt, amivel könnyedén feltelepül a program. Fontos, hogy a programot adminisztrátorként indítsuk el, mivel a fájlkezelés csak úgy fog működni.

# Rendszer konfigurálása

## Adat küldő

A rendszer ezen részén első sorban a konfigurációs fájlban kell átírni az url-t és portot. Ez értelem szerűen az lesz, amin a web api működik. Utána pedig el kell indítani a scripts/data\_sender.py scriptet. Ezt a könyebb használat végett érdemes beállítani az adott eszközön úgy, hogy mindig automatikusan induljon az autóval. Például egy Raspberry Pi-on mivel Linux fut, ezért ezt elég könnyen meg lehet tenni. Ehhez vagy egy service-t hozunk létre, vagy a crontab-ba rakjuk bele.

Ezután a program emberi beavatkozás nélkül fog működni.

## Telemetria alkalmazás

A **Settings/Configuration** menüben lehet konfigurálni az alkalmazást. Meg lehet adni annak a szervernek az elérési útját, amin a web api fut, illetve azt is lehet állítani, hogy mennyi időközönként kérje le az adatokat.

# Irodalomjegyzék

[A felhasznált szakirodalom megadása]

[A szerzők nevét mindenütt “Családnév, X.” formában kell megadni, ahol X. a szerző keresztnevének (keresztneveinek) kezdőbetűje. Magyar cikk esetén a vessző a családnév és a keresztnév kezdőbetűje közt elhagyható. Ha az egyértelműség megkívánja, a keresztnév kiírható teljesen is.

A dolgozat szerzője szabadon választhat, az A vagy a B típust használja.]

**[A-típus:**

A cikkekre való hivatkozás egy []-be írt sorszámmal történik. A sorszámozást folytonosan kell megtenni, a sorba rendezés alapja az első szerző családneve.]

[1] Szerző1 (, Szerző2 ...): *Cikk címe*

Folyóirat neve, sorszám, kezdőoldal-végoldal, év.

[2] Szerző1 (, Szerző2 ...): *Konferencia-kiadvány-beli cikk címe*

„Konferenciakiadvány:” Konferencia neve, hely, kezdőoldal-végoldal, év.

[3] Szerző1 (, Szerző2 ...): *Könyvcím*

„Könyv:” Kiadó, hely, oldalszám, év.

[4] Szerző1 (, Szerző2 ...): *Kutatási jelentés címe (csak publikus elérhető jelentés!)*

„Kutatási jelentés”: Kutatási projekt neve, intézet, oldal, év.

[5] Szerző: *Disszertáció címe*

„PhD/kandidátusi/stb. disszertáció”: Egyetem, kar neve, év.

[6] *Internetes oldal elnevezése:*

*URL, letöltés ideje (csak konkrét dokumentumra mutató URL adható meg!)*

**[B-típus:**

A hivatkozás a szerzők családi nevéből és a kiadás évéből képezett azonosítóval történik. Kettőnél több szerző esetén az „et.al.” rövidítés használható. Pl. „[Vijayasundaram, 1986.]”, „[Meister and Sonar, 1998.]”, „[Felcman et.al., 1994.]”

Abban a ritka esetben, amikor több cikknek is azonos azonosító jutna (megegyeznek a szerzők és a kiadás éve) az év után az „a”, „b”, „c”, stb. betűk csatolandók, pl. „[Stone and Norman, 1993a.]”.

A sorba rendezés alapja a szerzők családneve, végül az év.

Ezt leszámítva a formátum ugyanaz, mint az A-típus esetén, de ekkor a hivatkozási sorszám lehagyható.]

# Mellékletek

1. [A dolgozat mellékletei, ha vannak]