FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Deák Zsolt

Web scraping powershellel

Automatizált adatgyűjtés a világhálón

Konzulensek

Nguyen Gábor Loi

Dr. Goldschmidt Balázs

BUDAPEST, 2016

1. Tartalomjegyzék

[Abstract 7](#_Toc467876173)

[1 Bevezetés 8](#_Toc467876174)

[2 Irodalomkutatás 9](#_Toc467876175)

[2.1 PowerShell, és képességei 9](#_Toc467876176)

[2.1.1 Invoke-Webrequest, Invoke-RestMethod 9](#_Toc467876177)

[2.1.2 Internet Explorer object 10](#_Toc467876178)

[2.2 Front-end 10](#_Toc467876179)

[3 Tervezés 11](#_Toc467876180)

[3.1 Funkcionalitás (felhasználói szint) 12](#_Toc467876181)

[3.1.1 Back-end 12](#_Toc467876182)

[3.1.2 Front end 17](#_Toc467876183)

[3.2 Funkcionalitás (rendszer és komponens szint) 19](#_Toc467876184)

[3.2.1 Back end 19](#_Toc467876185)

[3.2.2 Front end 20](#_Toc467876186)

[3.3 Architektúra 21](#_Toc467876187)

[3.3.1 Front end 23](#_Toc467876188)

[3.3.2 Back end 24](#_Toc467876189)

[4 Megvalósítás 28](#_Toc467876190)

[4.1 Back end 28](#_Toc467876191)

[4.1.1 Link gyűjtő 28](#_Toc467876192)

[4.1.2 REST végpont 28](#_Toc467876193)

[4.1.3 Scraper 29](#_Toc467876194)

[4.1.4 Comparator 30](#_Toc467876195)

[4.1.5 Tesztelés 32](#_Toc467876196)

[4.2 Front end 35](#_Toc467876197)

[5 Összefoglalás 36](#_Toc467876198)

[6 Irodalomjegyzék 37](#_Toc467876199)

[7 Rövidítések jegyzéke 39](#_Toc467876200)

[Függelék 40](#_Toc467876201)

1. Ábrajegyzék

[ábra 1: V- modell 12](#_Toc467876202)

[ábra 2: Back end alapvető Use Case diagramja 12](#_Toc467876203)

[ábra 3: Back end Use Case diagramja 13](#_Toc467876204)

[ábra 4: Back end Use Case diagramja 2 13](#_Toc467876205)

[ábra 5: Front end használati esetek diagramja 17](#_Toc467876206)

[ábra 6: Hibaüzenet példa 18](#_Toc467876207)

[ábra 7: Front end – back end interakció 19](#_Toc467876208)

[ábra 8: Rendszer-kommunikáció szekvencia diagram 22](#_Toc467876209)

[ábra 9: Parse-olási sebesség eredmények 26](#_Toc467876210)

[ábra 10: Oldal lekérdezés átlagsebességek 27](#_Toc467876211)

[ábra 11:Példa a rossz paraméterezésre 30](#_Toc467876212)

[ábra 12: Futtatás eredménye egy autó adataira 33](#_Toc467876213)

[ábra 13: Futtatás eredménye 2 autóra 34](#_Toc467876214)

[ábra 14: Futtatás eredmény egy része 150 autóra 35](#_Toc467876215)

Hallgatói nyilatkozat

Alulírott **Deák Zsolt**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2016. 11. 27.

...…………………………………………….

Deák Zsolt

1. Kivonat

Adva van egy erőteljes eszköz, a PowerShell, aminek első számú célja a rendszeradminisztráció automatizálása, megkönnyítése. Szintén adott a probléma, hogy az Interneten strukturáltalanul jelen levő információhoz hozzáférhessünk. Ebben a munkámban egy konkrét esettanulmányon keresztül bemutatom, hogy hála a PowerShell sokszínűségének egy alapfeladatától távol eső területen is hatékony megoldást lehet vele készíteni.

A példában a Használtautó.hu autóhirdetéseinek adatait nyerem ki és dolgozom fel (az ilyen eljárások gyűjtőneve a web scraping). A feldolgozás célja, hogy a site egy hiányosságát, az összehasonlító funkciót pótolja. Eredetileg egy az árukereső.hu azonos lehetőségéhez hasonló, a termékeket adataikkal együtt egymás mellett oszlopokban, táblázat formájában megjelenítő összehasonlítás volt a cél. Ehhez hasonló már elérhető a Használtautó.hu-n is. Ezt kiegészítettem egy rangsorral, amit az autók tulajdonságaiból számított érték alapján állítok fel. Mivel különböző korú és állapotú járművek összehasonlítása lineáris módszerekkel, néhány tulajdonság kiválasztásával még megközelítőleg sem ad valós képet, így ennek alapját egy általam kidolgozott egyszerű (és determinisztikus) algoritmus adja. A nagyjából azonos korú és értékű autók összehasonlítása láthatóan értékes információval is szolgálhat ennek segítségével. A példa teljessége érdekében létrehoztam egy egyszerű weblapot is, hogy online elérhető legyen a szolgáltatás. Ez utóbbi nem PowerShell nyelven van írva, hanem egy szokványos PHP és JavaScript alapú website.

1. Abstract

There is a strong tool for system administration given, the PowerShell. There is the problem to be able to access unstructured data and information laying around on the Internet. In this work I made a proof of concept to prove that PowerShell can be used efficiently for tasks very much different from its original usage, thanks to the variety of features it has.

The proof of concept is about processing data of Használtautó.hu’s car pages (which is called web scraping in general). This processing is focused on the car comparison functionality that is not present on the site. The idea is coming from árukereső.hu’s similar functionality, a table based, side by side comparator of products’ details. One very similar is already available at the target page. I improved this idea by ranking the cars based on their main parameters. Due to the ineffectiveness of linear methodologies in comparing cars of varied ages and conditions by a handful of data, I needed to develop my own simple (and deterministic) algorithm. This gives the base of the car ranking that can produce valuable information about cars of similar ages and prices. For the sake of completeness, I created a webpage for the service to be available online. This is a user friendly abstraction in place of the PowerShell command line, written in PHP and JavaScript.

# Bevezetés

# Irodalomkutatás

## PowerShell, és képességei

A PowerShell egyszerre képes kiváltani a hagyományos Windows Management Instrumentation Command-line-t és nyújt hozzáférést a .NET keretrendszerhez egy konvencionális script-nyelven keresztül. Lehetőség nyílik rajta keresztül objektumokat használni, függvényeket definiálni és rendelkezik a script-nyelvekre jellemző tömörséggel is. [1]

Alapvető kérdés, hogy az egyszerű, HyperText Markup Language (HTML) alapú weblapok automatikus feldolgozásához rendelkezésre állnak-e eszközök. Még pontosabban, hogy milyen lehetőségeket nyújt a PowerShell a Hypertext Transfer Protocol (HTTP) lekérdezésekhez és az általuk visszaadott adatok kezelhető struktúrába átalakításához. Verziótól függően több megoldást is nyújtanak az előre definiált könyvtárak. PowerShell 3.0 – tól elérhetőek az Invoke-Webrequest[2] és Invoke-RestMethod[3] függvények. Régebbi verzió esetén az Internet Explorer object[5] segítségével lehet elérni azonos eredményt.[5]

### Invoke-Webrequest, Invoke-RestMethod

Az Invoke-Webrequest és az Invoke-RestMethod nagyon hasonló metódusok, szembeszökő, hogy paraméterezésük megegyezik. Az első különbség, ami észrevehető, hogy a részletes leírás szerint az Invoke-Webrequest HTTP, HTTPS, FTP, FILE kérések küldésére alkalmas weblapoknak és webes szolgáltatásoknak, míg az Invoke-RestMethod vonatkozó részénél csak a HTTP, és a HTTPS kérések küldése szerepel Representational State Transfer (REST) web szolgáltatások felé. Ennek ellenére az Invoke-RestMethod Uri paraméterének leírásánál mind a négy féle kérés fel van tüntetve.

Különbség lehetne még, hogy Invoke-WebRequest esetén a UseBasicParsing paraméter dokumentációjából kiderül, hogy enélkül a paraméter nélkül a háttérben az Internet Explorer (vagy annak modulja) van használva a parse-oláshoz. Ugyanez az Invoke-RestMethodról nem mondható el biztosan, mivel a dokumentációja nem szól róla. Ez azonban azért nem döntő értékű, mivel azon a kijelentésen kívül, hogy „Indicates that the cmdlet uses basic parsing.”, vagyis hogy az alapvető parse-olás használatát jelöli ez a paraméter, azon kívül csak egy másik paraméter leírása van duplikálva itt.

A fentiek alapján a hivatalos dokumentáció nem elégséges annak eldöntésére, hogy melyik függvényt érdemes használni, így a tervezés szakaszban az irodalomkutatáson túlmutató kísérletek segítségével választom ki a megfelelőt.

### Internet Explorer object

Az Internet Explorer (IE) ComObject[7] egy valódi IE példány programozott irányítását teszi lehetővé. A Visible nevű tulajdonság segítségével állítható, hogy a folyamat közben látható legyen-e a böngésző, vagy se. Az első kézenfekvő hátránya annak, ha az ember egy valódi böngészőt akar automatizálni az, hogy nem erre fejlesztették eredetileg. Az IE esetében probléma lehet az oldalak betöltésének hatékonysága.

A navigációs Application Pogramming Interface (API)[8] egy nemblokkoló híváson keresztül szolgáltatja a funkcionalitását. Ennek következtében vagy egy eseményre való feliratkozással, vagy polling módszerrel lehet értesülni az oldal betöltésének végéről. A betöltés jelentősen lassabb lehet, mint a fent tárgyalt két esetben, mivel itt az összes forrás is betöltődik (képek, gifek, flash stb.), köszönhetően annak, hogy egy hagyományos böngészőről van szó.

## Front-end

A front-endre azért van szükség, hogy könnyebben fogyaszthatóvá, használhatóbbá tegye a programot. Kiküszöböli a parancssor használatának nehézségeit: nem kell külön engedélyezni a scriptek futtatását (vagy egy megbízható szervtől aláírást szerezni rájuk)[9], utánanézni a paraméterezésnek, elolvasni a dokumentáció egy részét. Minden úgy működik, ahogy azt a felhasználók már megszokták az évek során használt programokkal, weblapokkal. A visszakapott eredményeket is könnyebb megérteni, ha valamilyen szép, vizuális megjelenítés van társítva hozzájuk, nem csak szimpla logok, vagy excel fájlok a kimenetek. Mindehhez nincs másra szükség fejlesztői szemszögből csak egy egyszerű weboldalra és a legelterjedtebb[10] webes front end keretrendszer megértésére és használatára.

# Tervezés

A back-end három fő és egy mellék komponensre osztható. Ezek a scaper, a rest service, a comparator és a funkcionalitásban részt nem vevő autó linkgyűjtő script.

Először ezek funkcionális képességeit ismertetem a pontos architektúra és belső felépítés bemutatásának mellőzésével, külön tárgyalva a felhasználói elvárásokat - az ennek való megfelelés kritériumait - és az egyéb funkcionális követelményeket, amelyek nem érintik közvetlenül a felhasználót (rendszer- és komponens szintű követelmények). A szétválasztás nem kizáró, vannak átfedések. Ezen felül felhasználói követelmények megvalósulásának szükséges feltétele a rendszer- és komponens szintűeknek való megfelelés, ámde ez utóbbiak az itt tárgyalttól eltérő rendszerrel is képesek lehetnek kielégíteni az előbbit.

Ez után következik az architektúrális tervezés, amiben a hangsúly a komponensek és a rétegek összekapcsolásán van. Szinén itt kerülnek bemutatásra az alacsony szintű tervezési döntések.

A használt stratégia leginkább a V modell[11] bal szárához (fejlesztői életciklus) hasonlítható, azzal a különbséggel, hogy nem minden tervezési szint jelenik meg elkülönülten, így egy logikusabb és koherensebb terv tud készülni. Magánál a fejlesztésnél az inkrementális iterációk módszerét használtam, tehát egyfajta agilis irányvonalat, viszont dokumentációs szempontból sokkal praktikusabbnak tűnt ez a fajta csoportosítás. Megvalósítástól függően változó lehet az egyes szintek definíciója, de talán a leginkább használtak a követelmények, specifikáció, magasszintű design (architektúra), alacsony szintű design.



ábra 1: V- modell

Látható az ábra 1-en, hogy az általam vázolt tervezési lépések hogyan feleltethetők meg a gyakorlatban használt egyik fajta V-modell szintjeinek.

## Funkcionalitás (felhasználói szint)

A funkcionalitás tervezésénél a használati eseteknek való megfelelés szempontjait járom körbe. Ehhez szükséges első sorban a használati esetek feltérképezése, majd az egyes komponensekben a megvalósítás megtervezése.

### Back-end

Mivel a back end teljes egészében script alapú, ezért kétféle képpen lehet rá tekinteni: mint kiszolgáló a front endnek, vagy mint önálló applikáció. A back end tervezésénél bemutatom mindkét szempontot. Ez a szakasz a felhasználói szempontok tervezését hivatott bemutatni, így a két réteg interakciója nem itt kerül tárgyalásra.



ábra 2: Back end alapvető Use Case diagramja

A Unified Modeling Language (UML) 2 Use case diagramon látható a felhasználó, mint aktor. Az aktor képes egy vagy több URL-t megadni az alkalmazásnak (scriptnek), ezt jelképezi az „URL bevitele” és ennek a leszármazottja az „URL tömb bevitele” use case-ek. Az előbbi a „kapott oldalak feldolgozása” használati esettel van asszociálva, ami maga a magja a szolgáltatásnak. Ennek a magnak részegysége az „algoritmus futtatása az adatokon” használati eset, mely minden esetben lefut (ezt jelöli az „include” stereotype a dependecia jel mellett), mégis kiemelendő.



ábra 3: Back end Use Case diagramja

A back end front endtől független működésének egyik Use Case diagramja látható a ábra 3 -on. A feldolgozási rész itt is ugyan úgy működik, mint az első esetben, ha az oldalak címei azonos formában rendelkezésre állnak már. Ehhez először a felhasználó meg kell adjon egy elérési utat, melyen egy text fájl található, ezt a back end feldolgozza és előállítja a szükséges formátumú URL listát.



ábra 4: Back end Use Case diagramja 2

A back end harmadik használati esete, mikor a felhasználó a scraper scriptet közvetlenül, az ábra 4-en ábrázolt „UseSaved” paramétert megadva indítja. Ebben az esetben, mint látható, a kívánt URL lista helyett közvetlenül az algoritmus futásához szükséges adatok képezik a „fájlok feldolgozása” use-case kimenetét.

A back endnek van egy kiegészítő funkciója is stand-alone használatkor: a felhasználó képes kigyűjteni általa egy, vagy több (autó) találati lista linkjeit. Ez akkor jelenthet előnyt, ha egy bonyolult preferencia rendszerrel meghatározta az elfogadható paraméterű autók körét, viszont a program segítségével meg akarja keresni a találatok közül a legjobbat, legjobbakat.

#### A scraper

Ez a script (skyscreper\_ie.ps1) képezi az egész projekt alapját, mivel ez végzi a weblapok automatikus feldolgozását, az adatok kinyerését. Háromféle teljesen szeparált működésre képes paraméterezéstől függően.

A legfontosabb, hogy képes egy URL, vagy egy URL tömb feldolgozására, melyet az Uri nevesített bemeneti paraméteren keresztül vár. A feldolgozás során vagy a megadott honlap adatait használja fel, vagy ennek egy aznap mentett (gyorsítótárazott / cache-elt) verzióját.

Másodsorban képes egy megadott elérési útvonalon lévő text állományból kiolvasni az URL-eket (soronként egy URL-t) és ezeken elvégezni a fent említett feldolgozást. Ehhez a működéshez a Path paramétert kell használni a script indításakor.

A harmadik típus használatához a UseSaved paramétert kell megadni bemenő érték nélkül. Ennek a kapcsoló (switch) fajtájú paraméternek a jelenléte indikálja, hogy a mentett adatok alapján kell futtatni a programot. Így az eddigi futtatások során keletkezett adatokon fog lefutni a kiértékelés, melyek már nem HTML formában vannak tárolva, hanem xml (Extensible Markup Language) fájlokban.

Látható, hogy a három bemenetből kettő csak kényelmi szempontból szerepel, mivel ugyan azt a szerepet töltik be. Annyiban szerencsés egy fájlból beolvasó módot is alkalmazni, hogy így nem vagyunk ráutalva a pipeline használatára, ez fontos lehet a processek közötti kommunikációban, ha nem közvetlenül az URL-eket szolgáltató folyamat indítja a scriptet. Fordítva pedig hasznos a pipeline-ra hagyatkozni, ha közvetlenül indítható egy másik programból, vagy kézzel hívjuk meg a scriptet, mondjuk, ha csak egy bemeneti URL-t tartalmazna a fájl, amit beolvas. Ebben a két esetben az az elvárás, hogy a megadott webcímeken lévő adatokat beolvassa és átalakítsa a script programozottan kezelhető struktúrába. Az adatokat és a lapokat későbbi offline tesztelés céljából a script képes elmenteni.

A maradék esetben (mikor xml fájlokat használ a program) már strukturált adatok kerülnek visszaolvasásra, így nincs szükség átalakításra. Ezután mindhárom esetben véget ér a script futása, innentől az algoritmusnak kerülnek átadásra az adatok, illetve ennek visszatérési értéke továbbítódik a hívónak.

#### A Comparator

A comparator service (compare.ps1) tovább viszi egy lépéssel a web scraping elvét. A bemenetén érkező adatokat feldolgozza, majd visszatér a feldolgozás eredményével. A bemeneten egy kulcs-érték párokból álló objektumot vár (hashtáblák tömbje), kimenetként pedig egy ember által olvasható HTML szöveget ad vissza és ment el a lokális állományok közé is. Az elmentett verzió mindig a legutolsó futtatás eredményeit tartalmazza, míg a HTML kódot azért kell vissza is adni, hogy egyszerű adatelérés legyen biztosítva a ráépülő szolgáltatásoknak.

#### Linkgyűjtő script

Ez a script ugyan nincs elérhetővé téve a REST endpointon keresztül, tehát nincs rá épülő front end szolgáltatás, viszont stand alone alkalmazásként használva a scriptcsomagot nagyon hasznos. Azt a használati esetet fedi le, mikor a felhasználó nem kifejezetten kiválasztott különböző márkájú, vagy évjáratú autókat akar összehasonlítani, amit biztosít maga a scraper és a comparator, hanem azt az esetet, mikor a felhasználó kihasználná az egymás mellé helyezés és rangsorolás adta lehetőségeket. Például van egy felhasználó, akinek megvannak a preferenciái: adott márkát keres (Suzuki), azon belül adott típust (Swift), meghatározott korút (2010 és utána gyártott), és természetesen van egy büdzséje a vásárlásra (max 4 millió forint). Ezek alapján futtat egy keresést a Hasznaltauto.hu-n és kap egy eredményt (majdnem száz autóhirdetés).

Ezeknek az egyesével végigböngészése helyett a találati oldal URL-jét bemenetül adva a linkgyűjtő scriptnek az kimenti egy fájlba az összes hirdetés linkjét, amely fájl közvetlen bemenetként át lehet adva a scaper-nek. A scraper lefuttatása után pedig lesz egy rangsor a látszólag ugyanolyan, vagy nagyon hasonló autókról. Természetesen ez a rangsor is tartalmazhat azonos elemeket, viszont egyfajta szűrést mindenképp jelent, már csak a hirdetésekben található adatok mennyiségét tekintve is, mivel negatívabban bírál el olyan hirdetéseket, ahol valamely adatok hiányoznak.

A linkgyűjtő ezen kívül képes egy másik paraméterben átvenni a linkgyűjtés mélységét, vagyis, hogy hány lapozást végezzen. Lapozás alatt a találati lista következő tíz autója értendő. Ugyanígy, mikor megnyitja a kapott linket, a tíz autót megjelenítő változatot fogja értelmezni. Ha a megadott mélység túlmutat az oldalak számán, a script leáll az utolsó még valós lap linkjeinek mentését követően.

### Front end

A front end feladata kiszolgálni a fent részben már tárgyalt funkcionalitások egy részhalmazát, egy sokkal könnyebben használható, és ami még fontosabb, sokkal könnyebben elérhető módon. A program használatakor a felhasználó kizárólag a felhasználói felülettel (ezesetben GUI, Graphical User Interface) áll interakcióban. Ezen kívül van még egy front end komponens, egy köztes réteg, egy front end domain-beli back end, amivel csak közvetve áll kapcsolatban a felhasználó.

Ez a közvetett kapcsolat az ábra 5-ön az „URL ek elküldése” és a „Feldolgozott adatok” use case-ekben jelenik meg működési szinten. A „Felhasználó” nevű aktor el tudja küldeni az általa megadott URL-ek listáját a „Back end” aktornak, aki képes a „Feldolgozott adatok” használati eseten keresztül eredményt szolgáltatni. A felhasználó szintén ezen utóbbi használati eseten keresztül tud hozzáférni az eredményekhez. A fent említett négy kapcsolat irányított asszociációkkal van jelképezve az ábrán.



ábra 5: Front end használati esetek diagramja

Ahhoz, hogy az URL-eket el lehessen küldeni, lehetőséget kell biztosítani a felhasználónak, hogy felvegyen URL-eket. Ezt jelképezi értelemszerűen az „új URL bevitele” use case, aminek van egy „URL módosítása” extensionje is, vagyis nem minden bevitelt követ (vagy része) egy módosítás, de van rá lehetőség, amennyiben teljesül a függőség mellé írt megszorítás, vagyis van már bevitt és nem törölt URL. Van tehát lehetőség törlésre is, két formában: lehet törölni az URL-t, vagy az egész mezőt, amely az URL-t tartalmazza. Utóbbi csak akkor lehetséges, ha legalább két mező van a képernyőn.

#### GUI

A GUI felhasználói szempontból képes kell legyen URL-eket beolvasni (maximum tizet) és lehetőséget kell biztosítson ezek feldolgozásra küldésére, módosítására, törlésére, ahogy ez az előző pontban felvázolt használati esetekben mint front end feladat szerepelt. Ezután a feldolgozás eredményéről képes kell legyen értesíteni a felhasználót. Ez az eredmény lehet a feldolgozásból származó adathalmaz, vagy hibaüzenet.



ábra 6: Hibaüzenet példa

Mivel az egyetlen művelet, ami nem elhanyagolható eséllyel hibát eredményez, az az URL-ek elküldése, így csak ennek eredményeképp kaphat a felhasználó hibaüzenetet a GUI-n. Ez tartalmaz egy segítség részt lehetséges okokkal, illetve a művelet során visszakapott státuszt.

A GUI képes kell legyen a bevitt adatokat tárolni egészen addig, amíg a felhasználó be nem zárja a böngészője azon tabját, amiben a webalkalmazás fut. Erre user experience (UX) szempontból van szükség, mivel ha a felhasználó mondjuk nem elég türelmes és az algoritmus futása közben a frissítésre kattint, elvesznének az addig bevitt adatai, vagyis legrosszabb esetben tíz URL-t is újra be kéne vinnie. Ráadásul, mivel ezek a szövegek nem túl olvasmányosak, nehezen megjegyezhetők, a felhasználó valószínűleg másolni fogja őket, tehát lehet, hogy ezzel az URL-ek újra kikeresését is szükségessé tenné az alkalmazás. Failure Mode and Effect Analysis (FMEA) szempontból ez a veszély viszonylag valószínű előfordulású (Probability), design tekintetében kritikus a bekövetkezés hatásának súlyossága (Severity), így ez a hiba kritikus (Risk Level), tehát a program késznek tekintésének (Minimum Viable Product, MVP) szükséges feltétele ennek a hibalehetőségnek a kiküszöbölése (Mitigation/Requirements).

## Funkcionalitás (rendszer és komponens szint)

### Back end

A back end funkcionalitás, ha nem a felhasználó szempontjából tekintjük, rendszer szinten a front enddel kommunikál csak, két (és fél) rétegű alkalmazásról lévén szó.



ábra 7: Front end – back end interakció

Az aktor (Front end) egy „valid POST request” eseten keresztül kommunikál a back enddel. A Front end actor és a „kapott oldalak feldolgozása” logikailag össze lehetne kötve közvetlenül is, viszont a köztes használati eset hűebben tükrözi a valóságot, mivel van egy közbenső szereplő, egy proxy is a folyamatban, ami szigorúan véve a front end része, mivel ennek a domain-jében van, de nem képezi részét a kliens oldali alkalmazásnak.

#### A REST service

A REST service (skyscraper\_rest\_service.ps1) azért lett létrehozva, hogy távoli eléréssel is lehessen futtatni a scrapert. Tehát online elérhetővé teszi a kulcsszolgáltatást, lehetővé téve, hogy a front end és a back end különválhasson egymástól. Szükségtelenné válik, hogy a front end szerver és a back end szerver ugyanazon a gépen, vagy virtuális gépen fusson. Ez nagy előnyt jelent az implementációban. A PoweShell 2016. augusztusig nem is volt elérhető, csak Windowson, most már elérhető nyílt forrású GitHub projektként és használható Linuxon és OSX-en.[12] A REST service a 8089-es porton várja a kérések beérkezését. Ennek a működéséhez az összes tűzfalnak és hálózatbiztonsági berendezésnek (amely a gép és a nyilvános hálózat között található) engedélyeznie kell a bejövő és kimenő forgalmat ezen a porton.

A service egymással időben nem átlapolódó kérések kiszolgálására képes, tehát szekvenciálisan (szinkron) működik. Ez az elv működőképességének bizonyításához elegendő, viszont valódi termékben nem alkalmazható módszer, mivel gyakorlatilag kizárja a többfelhasználós rendszert. Mivel az alapvető célkitűzés nem egy sokfelhasználós internetes szolgáltatás felépítése volt, ezért nem is lett rá hangsúly fektetve. Ennek ellenére a lehetőség adott, mivel a központi funkcionalitás skálázható újabb PS processek indításával. Viszont ekkor figyelembe kell venni megfontolásokat, mint load balancing[13][14], DDoS[14] támadások elleni védekezés, konkurens működés kezelése, közös erőforrás használat (a teljesség igénye nélkül), hogy egy valóban jól működő multitenancy[15] rendszert lehessen kialakítani.

### Front end

Az alkalmazásra, mint rendszerre tekintve a front end elsődleges szerepe a felhasználó számára absztrahálni a belső, bonyolultabb működést. Ennek a megvalósításához szükség van, hogy eljuttassa a GUI-ról gyűjtött inputokat a back endnek és vica versa. Ezt a rendszer szintű funkcionális követelményt valósítja meg a front end proxy (proxy.php) komponens.

#### Front end proxy

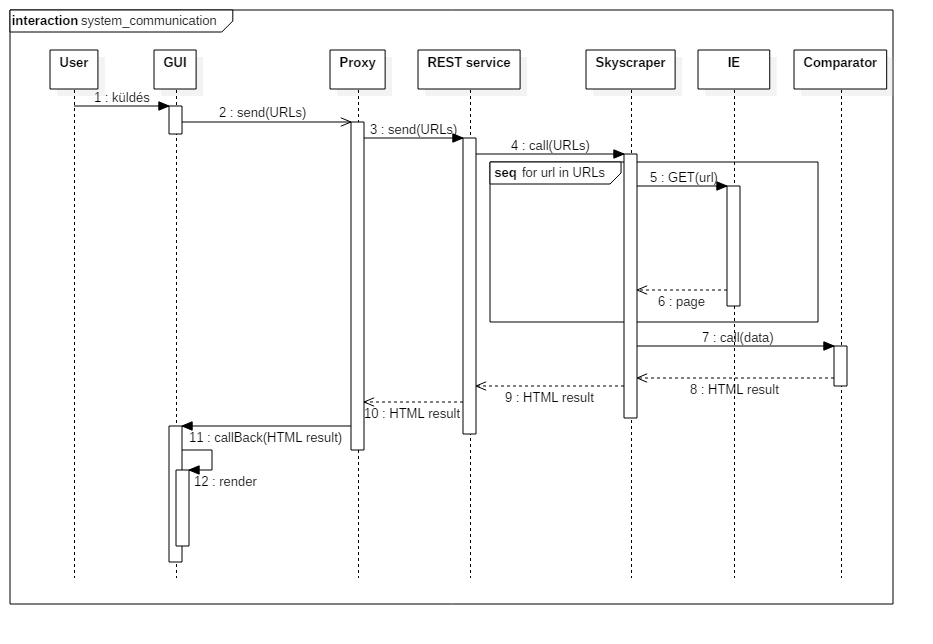
Ez teszi lehetővé a front end és a back end különválasztását. A ma használt böngészők szigorú biztonsági protokollokkal dolgoznak, ezek közül az egyik a Same-origin policy[16]. Ez többek közt nem engedélyezi a különböző domainek közötti Asynchronous JavaScript and XML (AJAX) hívásokat, melyek alapját képezik a back end – font end kommunikációnak. A probléma feloldására vannak szabványosított kiskapuk, mint például a Cross-Origin Resource Sharing (CORS) mechanizmus, ahol a szerver engedélyezheti egy header mezőben a same-origin policy-t sértő requesteket megadott domainek felől, viszont esetemben nem éreztem szükségét a biztonsági szabályok enyhítésének. Így lépett fel a szükség egy GUI-val azonos domainen lévő back end komponensre, amely egyben az első szűrő is, ami az elküldött adatokra alkalmazva van.

## Architektúra

Ebben a részben a technikai megvalósíthatóság szempontjait és követelményeit járom körbe. Az eddig tárgyalt felhasználói igények teljesüléséhez feltétlen szükséges a megfelelő architektúra és a lehetőségekhez képest a legmegfelelőbb technológiák használata. Egyszerre kell egyszerűen megvalósítható legyen és a felhasználó igényeit maradéktalanul kielégítő. A back end szempontjából a témaválasztás miatt adott a technológia.

A front end technológiáinál fontos szempont, hogy a tanulási folyamat gyors legyen, mivel a fejlesztési időnek nagyobb az értéke a PS fejlesztésekor (és tanulásakor), lévén ez a központi elem, ez a magasabb prioritás. Szintén első rangú fontossággal bír, hogy egy megszokott felhasználói felület legyen az eredmény, így elérje a front end a célját, a hatékony absztrakciót. Egy nehezen használható rendezetlen felhasználói felület épp akkora hátrány egy programnál, mint ha csak parancssorból irányítható: mindkettőt meg kell tanulni, meg kell szokni, energiát kell befektetni ahhoz, hogy hozzáférjen a felhasználó a termék érdemi részéhez. A web mai állása szerint viszont a felhasználók az első néhány másodperc alapján eldöntik, hogy használni akarják-e a weblapot amire navigáltak, vagy se.[17] A magas igények és a viszonylag alacsony ráfordítás (költségoptimalizálás) eredménye egy minimalista, letisztult design kell legyen.

Az architektúra a rendszer belső kommunikációinak kiszolgálására kell fókuszáljon magas szinten. A funkcionális leírásból is viszonylag jól kivehető kommunikációs szekvenciát szemlélteti a következő ábra. Ahogy már ott szerepelt a felhasználótól indul a folyamat a GUI-n kattintással („Elküldés”). A GUI elküldi egy POST requestben AJAX-on keresztül az URL-eket a Proxy-nak. Innen egy másik POST kérésben továbbítódnak az URL-ek a back end REST szolgáltatásához. A szolgáltatás a saját PS processén belül meghívja a Skyscrapert, a paraméterek még mindig az URL-ek („Uri” nevesített paraméter). A szekvencia diagramon jól látszik, hogy ekkor következik a legidőigényesebb feladat, az oldalak adatainak lekérdezése GET requestekkel.



ábra 8: Rendszer-kommunikáció szekvencia diagram

Az oldalakból ismét nem elhanyagolható, ám az előbbinél jóval rövidebb idő alatt felépül az adathalmaz, ezt kapja meg a Comparator, még mindig ugyanannak a PS processnek egy újabb szintjén. Ez előállítja a kimenetet az adatokból, lefuttatva rajta az algoritmust. Az eredmény visszaszivárog a hívási láncon a Proxy-ig. Mivel az eredeti hívás a Proxy felé aszinkron volt, ezért a GUI egy callbacken keresztül értesül az eredményről, amit megjelenít (ha valós, ha error). Ez utóbbi két folyamat a használt diagram editor technikai akadályai miatt ilyen hosszú, ezzel szemben a valóságbán attól függ a hosszuk, hogy milyen gépen fut a böngésző. Normál körülmények között olyan elhanyagolható időt vesz igénybe, ami nem is megjeleníthető az ábra többi folyamatának méretéhez viszonyítva. Ugyan ez igaz a hívások között eltelő időkre, csak a szemléletesség kedvéért vannak helyek hagyva. A lényegi információ a sorrendiség, az IE-rel való kommunikáció és az algoritmus futásának időigénye, a többi híváshoz képest, mivel itt több nagyságrend különbség is lehet.

Fontos kérdés, hogy a kommunikáció során milyen formátumban terjednek az üzenetek (az ábrán zárójelben írt paraméterek). Az URL-ek a lehető legegyszerűbb módon szöveg payload formájában utaznak végig a hívásokon. Egy sorban egy URL, maximum tíz URL. Visszafele pedig egy weblapon megjeleníthető valid HTML részlet jön, szintén text payloadként.

### Front end

Az architektúra részleteit az adat áramlásával megegyező irányból közelítem meg. Az adatok a front enden születnek meg, mikor a felhasználó beviszi őket a szövegmezőkbe.

A felület az internetes trendeknek megfelelően HTML(5) alapú. Ez a nyelv strukturáltságával, széleskörű támogatottságával ideális alap, jól kombinálható technológiák egész tárházával. A felület szerkezete tehát ezen a nyelven kell elkészüljön. A szerkezet magában pont olyan, mint a lakóházak esetében: lehetne lakni vakolatlan, kifestetlen házakban, ahol nincs áram, ajtók, ablakok, víz, hiszen kész van a kívánt modell. Az általános architektúrális követelményeknek való megfelelést ez az egy specializált alkövetelmény tehát nem teszi lehetővé. A felületnek szüksége van dinamikára (víz, áram, ajtó…) és elfogadható kinézetre (festés, bútorok, képek…).

A dinamika technológiája szintén adja magát, részben. Manapság egy HTML weboldal szinte kizárólagosan JavaScriptet (JS) használhat a kliens oldali műveletekhez. JS-en belül igen sok keretrendszer áll rendelkezésre, különböző módokon téve egyszerűbbé a kódot és a fejlesztést. Ezekre külön idő fordítása egy ilyen vékony kliens réteg esetén overengineering, architektúrális szempontból tehát irrelevánsak.

Nagyjából hasonlók igazak a kinézetre is. HTML külső formázásra egyértelműen CSS-t (Cascading Style Sheets) használ a webfejlesztő társadalom. Az erre épülő keretrendszerek szintén nagy számban vannak jelen az interneten. Sok közülük szabadon használható, jól dokumentált, egyszerű példákkal illusztrált, elterjedten használt. Ezek közt vannak kiemelkedők, mint a jQuery UI, ami főleg a felület dinamikusságát könnyíti meg és előre definiált elemeket (widget) nyújt, a Foundation (pl Pixar weblap designja), vagy a Bootstrap (Twitter fejlesztés). Ezek közül a Bootstrap tűnik legjobban támogatottnak és a legegyszerűbbnek is használat szempontjából. Követve a leírásokat, egy ilyen vékony front endet ezzel lehet a leghatékonyabban felhasználó-barát kinézetűvé alakítani, miközben látható a referencia oldalakon, hogy a jövőbeni terjeszkedésnek, fejlesztésnek is teret enged. Mindezt úgy teszi lehetővé, hogy magát a CSS-t jóformán nem is kell használni, csupán az általuk definiált kinézeti elemeket kell megfelelően kombinálni az elemek class attribútumain keresztül.

Az adatáramlás következő megállója a Proxy komponens. Ez a komponens egy viszonylag egyszerű funkcionalitást lát el, így tetszőleges back end technológián megvalósítható. A kiválasztás szempontja ezek alapján a támogatottság és elterjedtség kell legyen, így a választás egyértelműen a PHP-ra esett, elsősorban az elsöprő többségű piaci részesedése[18] miatt, másodsorban amiatt, hogy ha valóban a nyilvánosság számára elérhető weblap a cél, a PHP gyakorlatilag minden PaaS szolgáltatónál elérhető (future-proof design).

### Back end

A back endnél a technológia adott, PowerShell. Néhány architektúrális szempontot azonban itt is figyelembe kell venni. A legfontosabb, hogy egyáltalán képes legyen helyes működésre a back end, ha egy PS-el (Windows Management Frameworkkel) rendelkező operációs rendszeren, megfelelő beállításokkal próbálják futtatni. Az Irodalomkutatásban már megjegyeztem, hogy a REST hívás csak PS 3-tól került be a standard API-ba. Ennek következményeképp lehetséges, hogy egy Windows 7-es operációs rendszer alapértelmezésben még nem rendelkezik a megfelelő könyvtárral (például a clouds.bme.hu virtuális gépe [2016.11.07]), így az IE objektumot használja a dokumentum lekérdezéshez. A visszakapott dokumentumnak ezesetben más a típusa, mint a WebRequest által visszaadottnak. Ezen kérdések kezelésére a programot fel kell készíteni.

Egy másik fontos szempont a biztonság. Habár az ISTQB (International Software Testing and Qualification Board) szerint a biztonság funkcionális jellegű, ettől a standardtől eltérek, mivel a funkcionális terveknél inkább az egyes elemek használata volt előtérbe helyezve, és így ott ez a technikai jellegű követelmény némileg kilógott volna. Ha a felhasználó mint alkalmazást használja a scripteket, akkor csak magának tud ártani ráadásul rendelkezik a kóddal, bármilyen biztonsági intézkedést felülbírálhat. Tehát csak a front end felől és felé kell garantálni a biztonságos működést. A funkcionális követelmények garantálják a minimális szükséges hozzáférést engedélyezését a külvilág felé. Egy ponton tudhat bejönni az adat és ugyan itt ki. A rendszer szinkron működik, így temérdek biztonsági kockázat megszűnik, többek közt a párhuzamos folyamatok biztonságos izolációja, vagy a DDoS támadások. Egy rosszindulatú harmadik fél le tudja foglalni a számítási kapacitást így is. Ennek kiszűrése jóval túlmutat a projekt keretein. A rendszer szintén nincs felkészítve a man in the middle jellegű támadásokra. Fel kell legyen készülve azonban az injection jellegű támadásokra, mivel ezek nem csak a szolgáltatás kimaradását, vagy egy-egy felhasználónak adott rossz válaszokat eredményezhet, hanem az egész back end környezet meghibásodását, rosszabb esetben véglegesen. Ennek elkerülésére a bemeneten érkező URI-kat white-listelni kell, vagyis csak a megengedett típusú (potenciálisan hasznaltauto.hu autó weblap) címek érkezhetnek meg feldolgozásra.

Harmadrészt fontos nemfunkcionális szempont a rendszer sebességét leginkább szolgáló megoldások használata. Az architektúra áttekintésénél a szekvencia diagramon látszott, hogy a lapok lekérdezése és parse-olása a sebesség szempontjából legkritikusabb szakasz. Ezek optimumának megtalálására tehát külön mini projekteket kell létrehozni, a megfelelő tervezés lefolytatásához. Az egyik ilyen a parseSpeedTest.ps1.

A parse-olási sebességek összehasonlításánál az első lépésben letöltöttem egy dokumentumot, az új verziókban használatos Invoke-WebRequesttel. Meghívhattam volna bármelyik hasonló, rendelkezésre álló metódust is, de ezt nyújtja az újabb API és ez van a legjobban ledokumentálva (mi a visszatérési érték, mik a paraméterek).

A betöltött dokumentumot először a pipeline használata nélkül dolgozom fel:

$tables = $doc.ParsedHtml.GetElementsByTagName("TABLE")

ForEach($item in $tables){

if($item.className -eq "hirdetesadatok"){

$elements = $item

}

}

Ez a kódrészlet először leszűkíti a lap elemeit (node-jait) táblázatokra, majd ezek közt megkeresi azt a táblázatot, amely a hirdetés adatait tartalmazza. Ezt a folyamatot tízszer megismételve az időt átlagolva kijön egy nagyságrendi viszonyítási pont.

Ezután a dokumentumot ismét tízszer beparse-olja a pipeline-t használó verzió is:

$elements = $doc.ParsedHtml.GetElementsByTagName("TABLE") | Where-Object className -eq "hirdetesadatok"

Végül csak az összehasonlítás végett az IE parse-olását kikapcsolva is lefut ugyanaz, azzal az egy szembetűnő különbséggel, hogy itt nekem kell gondoskodni a „parse-olásról”, amit egy egyszerű reguláris kifejezéssel valósítok meg.

$doc = Invoke-WebRequest -Uri $uri -Method Get -**UseBasicParsing**

[regex]$regex = '<table class="hirdetesadatok">(.\*?|\r|\n)\*<\/table>'

For($i = 0; $i -lt $maxIter; $i++){

…

$elements = $regex.Match($doc.Content).Value

…

}

Itt is átlagolódik az idő, majd kilép az alkalmazás. Meglepő módon a következő eredményeket kaptam:



ábra 9: Parse-olási sebesség eredmények

Az első két módszer között többszörös az eltérés, továbbá látható, hogy a parse-olás a rosszabbik esetben is csak néhány tizedmásodperces nagyságrendbe esik. A harmadik eset szembeszökően gyorsabb volt a többinél.

A másik sebességet mérő script a getPageSpeedTest.ps1. Ennek hivatása eldönteni melyik web lekérdező megoldás a leggyorsabb. A mérési elv hasonló az előzőhöz, annyi különbséggel, hogy itt nem szeretném, ha esetleg torzítaná az eredményeket, ha valamelyik megoldás a háttérben cache-elné az adatokat, így itt annyiszor fut le a ciklus, ahány URL-t kap a script a bemenetén. Szintén különbség, hogy jelentős késleltetés tettem az egyes lekérdezések közé, hogy véletlenül se tűnjön rossz szándékú próbálkozásnak a tesztem.

Az első a megszokott WebRequest:

$doc = **Invoke-WebRequest** -Uri $uri -Method Get

A második a RestMethod:

$doc = **Invoke-RestMethod** -Uri $uri -Method Get

Ennek az előbbi kettőnek a dokumentációjuk alapján nemigen kéne eltérniük. A harmadik módszer azonban egy teljesen más stratégiával dolgozza fel az adatokat, így itt már várható különbség.

$doc = Invoke-WebRequest -Uri $uri -Method Get -**UseBasicParsing**

Végül pedig az IE objektum használatával töltöm le a dokumentumot:

$ie.**Navigate**($uri)

$i = 0

while ($ie.busy) {

Start-Sleep -Milliseconds 10

$i++

if($i -ge 300) {

Write-Host "Navigation timed out" -ForegroundColor Red

Continue

}

}

$doc=$ie.Document

A pollingos várakozásra aszinkron callback hiányában van szükség. Enélkül akkor is el lehet kérni a Docmentet, ha az még üres.

Ennek a kísérletnek az eredménye sokkal kevésbé egyértelmű eredményeket adott:



ábra 10: Oldal lekérdezés átlagsebességek

Erről az ábráról is azt lehet leolvasni, ami többszöri futtatás után is kijött. Az átlagos sebesség hozzávetőleg egy tizedmásodpercen belül van minden módszer esetén, tehát nem jelentős a különbség. Annál inkább igaz, hogy az egyes módszerek átlagos futási ideje is nagyjából egy tizedmásodperces sávon belül változik.

A fenti két kísérlet alátámasztja a szekvencia diagramon látott időviszonyokat, valamint az is kiderül belőlük, hogy még a parse-olás ideje se szignifikáns a weblapok letöltéséhez képest. A praktikai szempontokat figyelembe véve az Invoke-Webrequest a legjobb választás a REST hívásokhoz. A parse-oláshoz látszólag a UseBasicParsing-os verzió lenne a legjobb. Ellene szól azonban, hogy túlzottan specializálja a scrape-elést erre az egy weboldalra, illetve hatalmas regressziós lehetőségeket rejt magában és plusz komplexitást visz a rendszerbe. Nem elég ugyanis egy reguláris kifejezéssel kinyerni a megfelelő TABLE tag kódját, ebből a kapott HTML részletből ki kell bontani a belső szöveges tartalmat. Ha automatikus parse-olást alkalmazok, csak néhány elemre kell hagyatkoznom a HTML Document Object Model struktúrát tekintve. Ehhez hozzátéve, hogy habár sokkal gyorsabb a „kézzel” parse-olás, a teljes futási időhöz képest nem akkora nyereség. A koncepció, hogy ez a nyelv is alkalmas a scrapingre elvesztené a lényegét, ha egy nyelvi szinten generális (más nyelveken is könnyen megvalósítható), viszont csak egy adott feladatra alkalmazható megoldást választanék. Így marad a második leggyorsabb parse-olás, mint felhasználandó technika.

# Megvalósítás

Adva vannak az irányelvek, a felhasználónak való megfeleléshez. Meg van határozva az MVP eléréséhez szükséges feature lista. Meg vannak tervezve a kommunikációs interface-ek, csakúgy, mint a kommunikáció tartalmi sémája. Ki vannak találva a folyamatok, a sorrendiség és az együttműködő komponensek. Ami hátra van, az a kivitelezés fontosabb részleteinek tárgyalása és a teszt megfontolások.

## Back end

Először a kisegítő funkciókat ismertetem, majd a hívási láncon lefele haladva a back end funkcionalitás lényegi elemeit.

### Link gyűjtő

A link gyűjtő segédalkalmazás a bemenetén egy URL-t vár és opcionálisan egy számot, aminek alapértelmezése 5. Belül egy ciklus fut annyiszor, ahányat a felhasználó ebben a paraméterben megadott, kivéve, ha nincs annyi találati oldal. A ciklus belsejében egy a tervezésből már ismert WebRequest lekérdezi az aktuális oldalt, majd ebből egy egyszerű reguláris kifejezés (http://www.hasznaltauto.hu/auto/(.\*/.\*)\*`">) kigyűjti a használt autó linkeket és elmenti egy előre meghatározott fájlba. A ciklus következő futása előtt a program átírja az aktuális találati lista linkjét. Itt tehát tamperinget alkalmazok a találati lista oldalak közötti navigáláshoz. Ennek előnye, hogy a háttérben a linkek elnevezési struktúrája sokkal kevésbé valószínű, hogy változzon, mint mondjuk a node-ok struktúrája, amelyek tartalmazzák a linket a következő oldalra. Hátránya, hogy validálni kell, valóban van-e következő oldal. Ezt a válaszban kapott link és a lekért link összehasonlításával teszem meg, tehát legrosszabb esetben egy felesleges lekérdezés történik.

### REST végpont

A végpont dolga a beérkező kérések továbbítása a szolgáltatásnak és a megfelelő válaszok konstruálása. Ehhez egy .NET osztályt, a HttpListenert használja. A kérések feldolgozásakor ellenőrzi, hogy a használt metódus POST-e, van-e payload az üzenetben, illetve ha van, ezt maximum 10 sorra korlátozza. Ha valamelyik feltétel nem teljesül, egy kliens oldali hibakóddal és üzenettel tér vissza. Szintén kliens oldali hibát küld el, ha az ellenőrzések sikeresek, a belső szolgáltatás meghívása sem akad el, viszont nem érkezik felhasználható válasz. Függ tehát a REST szolgáltatás a Scraper szolgáltatás működésétől, mivel ez nem ad vissza eredményt, ha nem valid a bemeneti adat. Ez a validálási kör azért került a Scraperbe, mivel az ő felelősségi köre eldönteni milyen adatokat fogad el, viszont nem hibával tér vissza, mivel a működése során nem feltétlen történik ilyenkor hiba. Ha történik, azt a REST szolgáltatás már mint szerver oldali hibát jelzi a kliensnek. Szintén szerver oldali hibával tér vissza a saját kivételei elkapásakor. Egyetlen Scraper felől érkező kivétel van amit külön kezel a REST szolgáltatás, ha nem megfelelő formátumú weblap URL-t kap a bemenetén. Ez szintén kliens oldali hibakódot generál, viszont technikai okokból nem lehet elnyomni a dobó oldalán.

### Scraper

A Scraper szolgáltatás háromféle bemenete és ezek használati lehetőségei már fentebb tárgyalásra kerültek. Implementációjában a három diszjunkt használati eset a PS nyelvi lehetőségeit kihasználva lett szétválasztva.

Param

(

[ValidatePattern('^http:\/\/www.hasznaltauto.hu\/auto\/([a-zA-Z]|\d|\/|\_|-|\.)+$')]

[Parameter(Mandatory=$true, ParameterSetName = "Online")]

[String[]]

$Uri,

[Parameter(Mandatory=$true, ParameterSetName = "MultiOnline")]

[string]

$Path,

[Parameter(ParameterSetName = "Offline")]

[switch]

$UseSaved

)

Az Uri paraméter lehet egy String, vagy egy String tömb. Ezt a fölötte lévő ValidatePattern attribútum egy reguláris kifejezés argumentummal validálja. Ez a validáció kivételt okoz, ha a feltétel nem teljesül. Az Uri paraméternél és a Path paraméternél is látható a Parameter attribútum Mandatory argumentummal, ami alapvetően az jelentené, hogy mindkettőt kötelező megadnia a hívónak, viszont a ParameterSetName argumentummal definiálva van, hogy külön futtatási módhoz tartoznak. A hívó nem kell megadja a ParameterSetName értékét futtatáskor, amelyik paramétert használja, az a set lesz aktív.



ábra 11:Példa a rossz paraméterezésre

UseSaved futtatási esetben az előre definiált (beégetett) kimeneti mappa fájljain van hívva egy Import-Clixml metódus, ami egyből a kulcs – érték párokba rendezett adatokkal tér vissza. Ennek haszna, hogy az algoritmus tesztelését lehet akár offline is végezni, ráadásul jelentősen gyorsabban, sok adaton.

Path és Uri paraméter használatakor is az első lépés annak megállapítása, hogy milyen verziójú PS-ben fut az alkalmazás. A kritikus pontokon később ez alapján disztingvál a rendszer.

If(**$PSVersionTable.PSVersion.Major -ge 3**){

$data = ScrapeWebPages

} Else{

$data = ScrapeWebPages -compatibilityMode

}

A ScrapeWebPages funkció tartalmazza innentől a lényegi részeket a scriptben. Először az alapján, hogy melyik paraméter volt használva, épít egy URI tömböt, majd a tömb minden elemére elvégzi a következőket:

* Whitelisteli a címet, csak akkor dolgozza fel, ha a jó domainre mutat. Erre azért van szükség, mert nem feltétlen az Uri paraméterből jönnek a címek.
* Megnézi a mentett adatokat, és ha megtalálja köztük az aktuálisan kiértékelendő autóét, és ez nem régebben lett mentve egy napnál, akkor eltárolja az adatokat az aktuális indexhez egy másik tömbben (dataTable) és tovább lép a következő URL-re. A keresés egy lépésben történik, mivel az URL-ekből generálódik az adatokat tároló XML fájlok neve.
* Meghívja a spike-okban[20] kiválasztott letöltő és parse-oló műveleteket annak megfelelően, hogy compatibility módban fut-e, vagy se
* Hozzáadja a dataTable tömbhöz az adatokból és a címből álló hashmapet
* Elmenti az adatokat XML-ben (cache-eléshez)

Innentől a Comparator szolgáltatás veszi át az adatokat a Scraper közvetlenül ennek kimenetét adja tovább.

### Comparator

A Comparator szolgáltatás felépít egy HTML szöveget, ami egy táblázatot tartalmaz az autók adataival. Táblázat sorai az autók paraméterei, az oszlopai az autók. Az utolsó sor egy számított értéket tartalmaz minden autóhoz, ez alapján vannak sorrendbe rakva az oszlopok. Az autók adatainak valódiságát, helyességét a program nem kérdőjelezi meg. Az értékek egy hét tulajdonságot tartalmazó determinisztikus algoritmus futtatásával állnak elő. Az egyes tulajdonságok és súlyaik:

1. Kilowattban mért teljesítmény. Az autónál megadott érték tizennegyedével kerül számításba. Ha nincs megadva, értéke egy.
2. Állapot. Ha bármilyen módon sérül, vagy nincs megadva, értéke mínusz húsz. Minden más esetben értéke nulla.
3. Literben mért csomagtartó méret. Ha meg van adva, a száznegyvened részével kerül számításba, egyébként értéke nulla.
4. Saját tömeg, kilógrammban kifejezve. Ha meg van adva, a mínusz ötszázadával kerül számításba, egyébként értéke mínusz három.
5. Kilóméterben mért futásteljesítmény. Az első százezer kilométer mínusz tíz pont, egyenletesen elosztva (ha még nem ment volna annyit). Száz- és kétszázezer kilométer között további öt pont a levonás, szintén egyenletes eloszlással, valamint minden további százezer kilométer további kettő egész öt tized pont mínusz, egyenletesen.
6. Ha meg volt adva teljesítmény és meg van adva az ár, akkor a következő képlet szerint kerül számításba: . Egyéb esetekben az értéke egy.
7. Ha meg van adva az autó gyártási ideje, akkor egy Edmunds[21]-et alapul véve, a következőképp kerül számításba:

Ahol M a gyártás hónapja. Az így kapott szám additív inverzének harmada kerül a végső összegbe.

Ezen hét érték összege plusz hetven egy autó értéke. A plusz hetvenre azért van szükség, hogy ne zavarja össze a felhasználót, hogy az érték sokszor negatív. Az egyes tulajdonságok pozitív vagy negatív előjele az érték meghatározásból származtathatók. Mivel egy használati eszköz valós értékét igyekszem meghatározni, ezért az egyes tulajdonságokról el kell dönteni, hogy valóban értéket képviselnek-e, vagy pont hogy csökkentik az értéket. A számok a személyes heurisztikáim az értékmeghatározáshoz, mivel évek óta figyelem a mainstream autó médiát és használt autó piacot, ezért ez egy hozzávetőleg szakértői heurisztikának tekinthető. Ugyanígy, az érték, vagy nem érték kérdés is szubjektív megítélésű, például, hogy az autó tömege egy látszólag neutrális paraméter, de valójában egy súlytalan autó, vagy egy versenytársaihoz képest fele súlyú nagyon kelendő lenne. A kevésbé fontos karakterisztikák, mint tömeg és csomagtartó méret kisebb, míg a fontosak, mint futásteljesítmény, ár, kor, teljesítmény jóval nagyobb súllyal kerülnek számításba. Az állapot kilóg a sorból. Ennek legfontosabb információtartalma akkor van, ha véletlenül hiányoznak a hirdetésből, de természetesen jelentős levonást eredményez az értékből, ha sérült, csupán nincs megkülönböztetés a sérülés vagy hiányosságok mértékei között. Ezt a logikát követve: minden hiányzó valamilyen formában rontja, vagy legalábbis nem növeli az értéket, típusától függően. Az egye súlyok valós értékeire a tesztelésben lehet majd példát látni.

### Tesztelés

Mivel a scriptben nagyon közvetlen módon jelennek meg a felhasználói, magas szintű követelmények, ezért a back end tesztelésénél is ezeknek lefedésén volt a hangsúly. A back end mint rendszer a Scraper komponensen keresztül tesztelhető. Ennek bemenetein részben whitelisting, részben validáció van alkalmazva. Ha rosszul paraméterezi fel a felhasználó a scriptet, nem kap eredményt. Mivel ebben az esetben a forráskód rendelkezésre áll más failure módok tárgyalása értelmetlen, mivel nem lehet rájuk felkészülni. A performance kérdéseket a tervezési szakasz letesztelte. Amit lehet és érdemes tesztelni, hogy hogy működik a back end egy adaton, néhány adaton, és sok adaton. Az elvárás, hogy soha, amikor komolyabb változás van a kódban, ne lépjen fel olyan regresszió, amely a bemeneti paraméterek helyes megadása esetén sikertelen futtatáshoz vezet.

1. *Teszt egy autó adatain:*

Az autó eredeti adatainak egy részlete (+ nincs csomagtartó méret és tömeg):

|  |  |
| --- | --- |
| Teljesítmény | 320 kW, 435 LE |
| Kilométeróra állása | 17 078 km |
| Vételár | 29.900.000 Ft |
| Állapot | Sérülésmentes |
| Évjárat | 2014/1 |

Az ebből kapott eredmények:

|  |  |
| --- | --- |
| Mass | -3.00 |
| AgeLoss | -13.33 |
| Price | 10.00 |
| Speedometer | -1.71 |
| Condition | 0.00 |
| Power | 22.86 |
| Trunk | 0.00 |
| **Worth** | 84.82 |



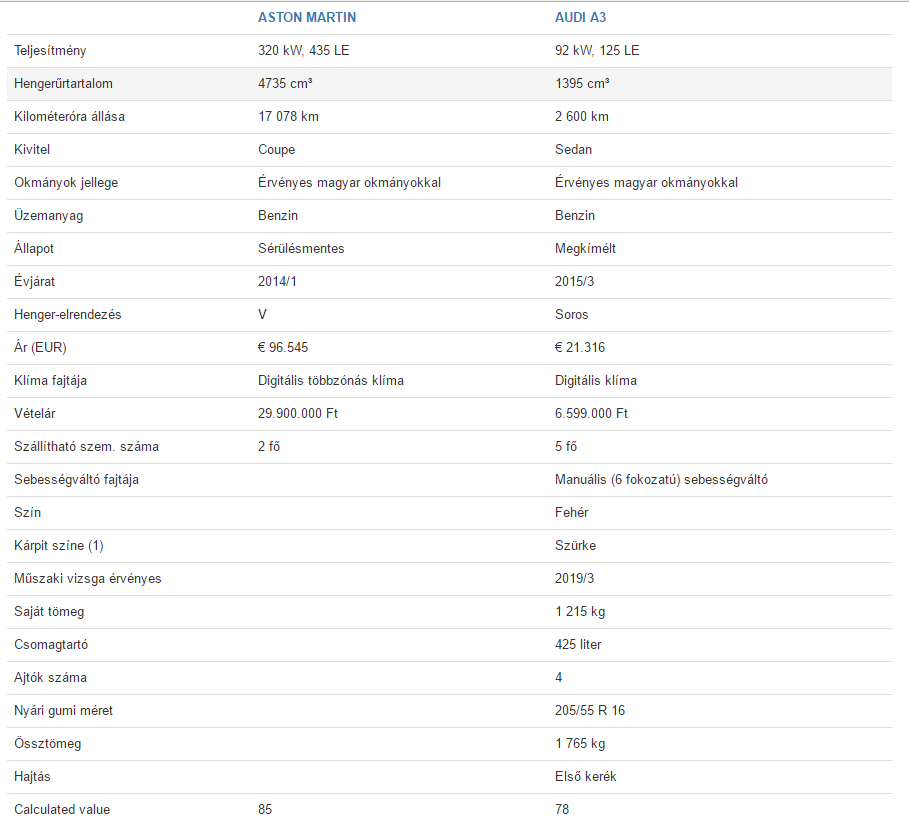
ábra 12: Futtatás eredménye egy autó adataira

1. *Teszt két autó adatain:*

A második autó adataival a bemenet releváns része:

|  |  |  |
| --- | --- | --- |
| Teljesítmény | 320 kW, 435 LE | 92 kW, 125 LE |
| Kilométeróra állása | 17 078 km | 2 600 km |
| Vételár | 29.900.000 Ft | 6.599.000 Ft |
| Állapot | Sérülésmentes | Megkímélt |
| Évjárat | 2014/1 | 2015/3 |
| Tömeg |  | 1 215 kg |
| Csomagtartó |  |  |

A futtatás eredménye:

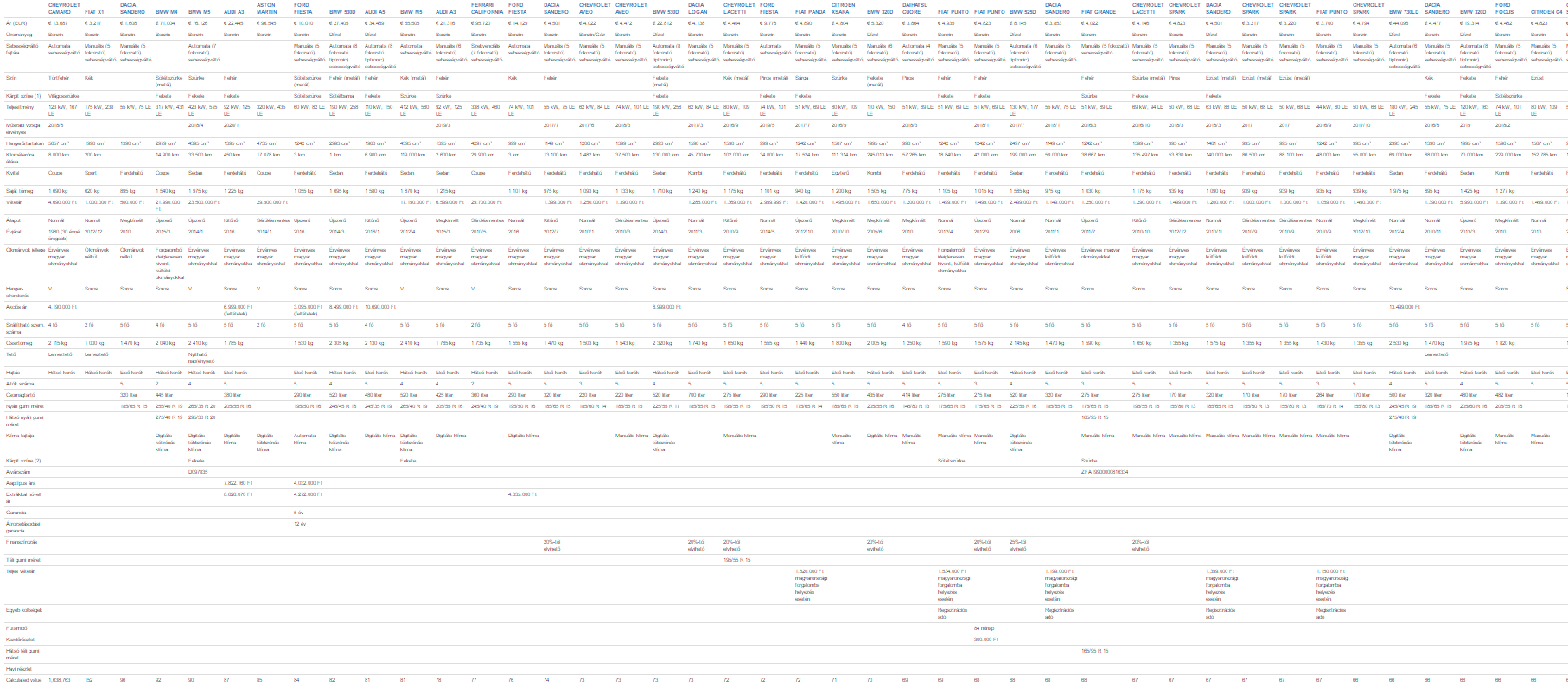


ábra 13: Futtatás eredménye 2 autóra

Látható, hogy a második autónál jóval több alapadat meg volt adva. Ez természetesen nem okoz híbát, minden kiírásra kerül.

1. *Teszt 150 adaton:*

Ennél a tesztnél már meglévő xml fájlba mentett adatokat használtam. Az algoritmus ilyen körülmények között 1,6 másodperc alatt futott le. Az adatok lementése csak teszt üzemmódban történik meg, rendes futáskor csak cache-elés céljából készül átmeneti fájl.



ábra 14: Futtatás eredmény egy része 150 autóra

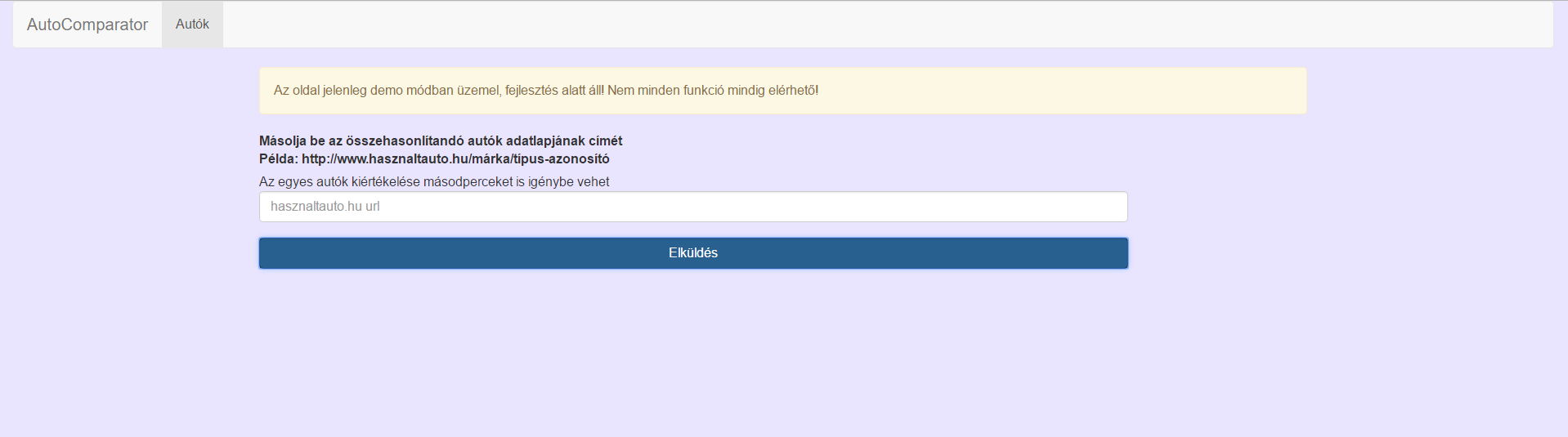
## Front end

A front end kialakításához nem használtam fel külső modularizálást támogató frameworkot. A legnagyobb része a front endnek „natív” JS, néhol JQuery-vel kiegészítve. Az legfontosabb szempontok, ahogy a tervezésben is szerepelt, az egyszerűség és a használhatóság.

A front end kiszolgálója egy tetszőleges webszerver, amely támogatja a PHP futtatást. Tetszőleges alatt értem az apache mockot, amely egy WAMP(Windows Apache MySQL PHP) stack formájában van használva teszteléshez, épp úgy, mint egy PaaS szolgáltató (kívülről nem feltétlen látható) szerver szolgáltatását, amely PHP futtatást biztosít, mondjuk egy Git szinkronizáción keresztül. A PHP index oldal semmi mást nem csinál mint meghivatkozza a valódi index HTML oldalt, ezt kapja meg a browser.

A megoldás könnyedén hordozható szolgáltatók között. Példának okán, ahhoz, hogy egy Heroku nevű PaaS providertől átmigráljam AWS (Amazon Web Services) Elastic Beanstalkra, a kódon nem kellett módosítani, csak konfigurációs fájlokon, melyek az adott környezet jellemezték.

### Graphical User Interface



ábra 15: GUI kezdőképernyő

A UI modularizálását natív módon oldottam meg, valamilyen struktúrálásra még egy ilyen egyszerű felület esetén is szükség volt. Van az alapvető működést támogató index.js Ez kezeli a lap betöltődését és a kommunikációt a PHP proxy-val.

Van még a navigációs bart felépítő NavbarBuilder.js. Single-page applikációnál nincs kifejezett szükség ennek a különválasztására, viszont praktikus, ha több oldal is lenne egyszer.

Van végül egy InputElementBuilder.js fájl, ez végzi a szövegmezők létrehozását és törlését. Mikor a felhasználó ráklikkel az aktuális utolsó szövegmezőre, létrejön egy új üres mező ez alatt, ha még nem volt összesen tíz beviteli mező felvéve. Az első kivételével minden mező mellé felvesz egy gombot is, amivel a mezőt ki lehet venni a listából.

Az aktuális sessionben a felhasználó bevitt értékei nem vesznek el. Ezt a JS sessionStrorage objektuma biztosítja, mely az oldal elhagyásakor elmenti a szövegmezők tartalmát. Mikor a felhasználó a lapra navigál ismét a következő kódrészlet fut le többek közt:

if( !sessionStorage.inputUriCount || sessionStorage.inputUriCount > 10 || sessionStorage.inputUriCount < 1 ) {

sessionStorage.clear();

var textDiv = createTextDiv();

formElement.insertBefore(textDiv, sendButtonDiv);

} else {

var savedUris = JSON.parse(sessionStorage.inputUriTextContent);

for(var i = 0; i < Number(sessionStorage.inputUriCount); i++) {

var textDiv = createTextDiv(savedUris[i]);

formElement.insertBefore(textDiv, sendButtonDiv);

}

}

Először ellenőrzi a storage tartalmát, majd ha ez az elvárt, átadja egyesével az InputElementBuilder createTextDiv metódusának, amely létrehozza a beviteli mezőt. Ezután a mező és a gombja elhelyezésre kerülnek a küldés gomb felett.

A futtatás eredménye a back endnél már tárgyalt HTML alapú táblázat, így ez közvetlenül van betöltve az oldal HTML kódjába, amikor megérkezik a válasz.

if (xhttp.readyState == 4 && xhttp.status == 200) {

document.getElementById("resultTable").innerHTML = xhttp.responseText;

}

Ezzel szemben, ha sikertelen a művelet egy piros szövegdobozban egy hibaüzenet tájékoztatja a felhasználót a hiba jellegéről és a potenciális megoldási lehetőségekről, ahogy ez az ábra 6: Hibaüzenet példa képen is látható volt.

### Proxy

A proxy.php szerepe lehetővé tenni, hogy tetszőleges back end service-szel tudjon kommunikálni a front end. Enélkül a back end válaszaiban mindig jelen kéne legyen egy megfelelő whitelist, amely tartalmazza a front end aktuális domainjét. Ez a tervezésben tárgyaltakon kívül hordozhatóság szempontjából is praktikus. A kapott adatokkal felépít egy curl hívást (ha kapott adatokat, ezt az egyet ellenőrzi), amelyet egy előre definiált cím felé indít.

### Teszt

A felhasználói felület tesztelésénél a legfontosabb szempont, hogy helyes bemenetre működik-e a rendszer. Csak egy helyes scenario van, minden más esetben hibát kell jelezzen a felület. Minden scenariot nem lehet tesztelni az ISTQB exhaustive testingről szóló része szerint, ebből kifolyólag csak a fontosabb negatív eseteket fedem le.

1. *Pozitív eset*

Egyedi, valid url-eket megadva, az eredmény az autók adatai kell legyenek az „Elküldés” gomb alatt, legalul, ahogy már feljebb is látható volt a számított értékkel.



ábra 16: GUI tesz pozitív eset

1. *Azonos URL-ek*

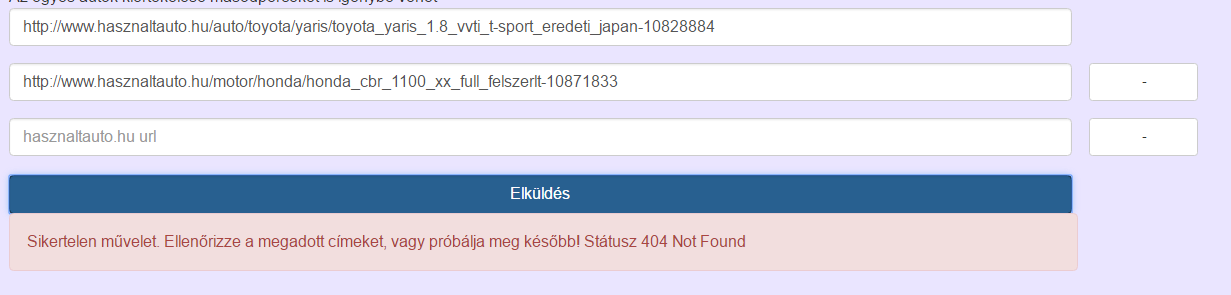
Ha azonos URL-eket ad be a felhasználó, az autó adatai csak egyszer jelennek meg. Az üres mezők nem kerülnek elküldésre.



ábra 17: GUI teszt azonos bemenetek

1. *Hibás URL megadása*

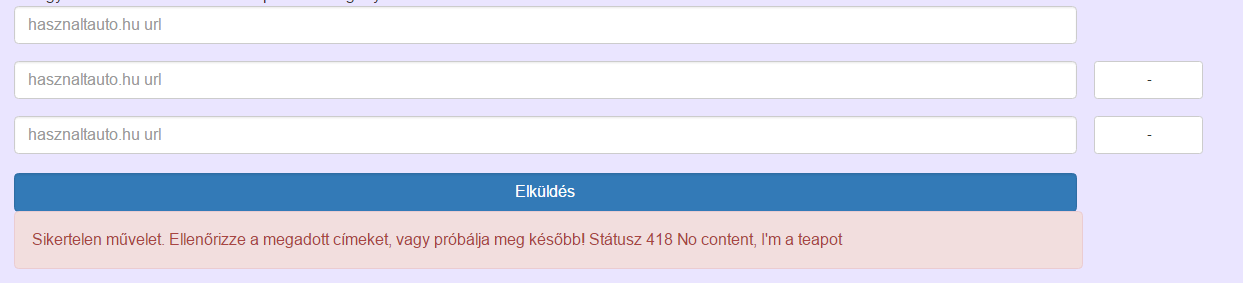
Ha a felhasználó nem jó URL-t ad meg (nem hasznaltauto.hu-ra mutat, vagy nem autóra mutat …), hibaüzenetet kap vissza.



ábra 18: GUI teszt hibás bemenet

1. *Üres bemenet*

Ha a felhasználó nem ad meg egy URL-t se, szintén hibaüzenetet kap.



ábra 19: GUI teszt üres bemenetek

1. Összefoglalás

A tervekből és megvalósításból látható, hogy az MVP teljesíti a vele szemben támasztott elvárásokat. A felhasználók képesek általa egy találati listát tovább szűkíteni, így megkönnyíti a döntési folyamatot. A folyamat teljesen automatizált, egyszerűen használható. A megvalósítás hordozható és kihasználja a technológia adta lehetőségeket a maximális teljesítmény eléréséhez, ami lehetséges vele. Az algoritmus képes felderíteni nagyjából azonos hirdetések különbségeit.

Mindezek mellett rengeteg fejlesztési potenciálja is van a programnak. Lehetnének felhasználói profilok, amelyekhez hozzá lehetne rendelni keresési előzményeket és preferenciákat, kedvenceket. A találati listán való keresés ki lehetne ajánlva a GUI-ra. Lehetne a keresések eredményét szűrni, sorrendbe állítani más szempontok szerint, mint ahogy alapértelmezésben van. Az egyes tulajdonság szerint a legjobb érték ki lehetne emelve a találati táblázatban. Ezekhez a változtatásokhoz érdemes lenne megfontolni egy rendes modularizálást használó front end keretrendszer használatát, mivel ekkor már a front end is sokkal nagyobb komponens lenne, így a fenntarthatóság érdekében megfelelően kellene struktúrálni. A statikus súlyokkal dolgozó algoritmus kaphatná a preferenciákat a felhasználótól. A mostani megvalósítás helyett a jobb értékmeghatározáshoz lehetne használni külső értékeléseket, új autó ár figyelést és ebből adott típusra értékcsökkenést, lehetne fenntartási költségeket számolni, megbízhatóságot figyelni. Ki lehetne bővíteni az algoritmushoz használt tulajdonságok listáját az extrákkal (tolatóradar, bőrülés, klíma…), lehetne falhasználónként profilt analizálni. Összefoglalóan, lehetne az algoritmusban valamilyen mesterséges intelligenciát alkalmazni, tanulóvá tenni a folyamatot, miközben kiegészíteni a rendelkezésre álló adatok listáját. Ezen többnyire back end átalakításokhoz már nyilvánvalóan kellene egy adatbázis is, esetleg ha a számításokhoz kell egy in-memory adatbázis, akkor kettő. Teljesítmény szempontjából, ha a PS technológiánál maradva kellene fejleszteni, akkor az aszinkronitás kérdéskörét és az ezzel kapcsolatos biztonsági feladatokat kéne megoldani.

Kitűnik a leírásból, hogy további lehetőségeket fog nyújtani a PowerShell is ebben az irányban, hiszen csak az elmúlt néhány évben is publikáltak új könyvtárakat a fejlesztői, amelyek segítik a scrapelést. Van tehát egy kész Proof of Concept, amely megmutatja a PowerShell képességét scraping feladatok ellátására, ezen belül pedig egy konkrét felhasználási mód példázza ennek hasznosságát.

1. Irodalomjegyzék
2. TechNet: Scripting with Windows PowerShell, <https://technet.microsoft.com/en-us/library/bb978526.aspx> (revision 15:03, 4 July 2016)
3. TechNet: *Invoke-WebRequest*, <https://technet.microsoft.com/en-us/library/hh849901.aspx> (revision 09:57, 10 September 2016)
4. TechNet: *Invoke-RestMethod*, <https://technet.microsoft.com/en-us/library/hh849971.aspx> (revision 09:58, 10 September 2016)
5. MSDN: *InternetExplorer object*, <https://msdn.microsoft.com/en-us/library/aa752084(v=vs.85).aspx#properties> (revision, 14:25, 12 September 2016)
6. PowerShell Team: *Controlling Internet Explorer object from PowerShell*, <https://blogs.msdn.microsoft.com/powershell/2006/09/10/controlling-internet-explorer-object-from-powershell/> (revision, 14:22, 12 September 2016)
7. TechNet: *about-Variables*, <https://technet.microsoft.com/en-us/library/hh847734.aspx> (revision 21:32, 8 September 2016)
8. JuanPablo Jofre (MSDN): *Creating .NET and COM Objects (New-Object)*, <https://msdn.microsoft.com/en-us/powershell/scripting/getting-started/cookbooks/creating-.net-and-com-objects--new-object-#creating-com-objects-with-new-object> (revision 19:42, 14 September 2016)
9. MSDN: *Navigate method*, <https://msdn.microsoft.com/en-us/library/aa752093(v=vs.85).aspx> (revision 20:26, 14 September 2016)
10. TechNet: *Using the Set-ExecutionPolicyCmdlet*, <https://technet.microsoft.com/en-us/library/ee176961.aspx> (revision 18:14, 17 September 2016)
11. Bootstrap: *Bootstrap index page*, <http://getbootstrap.com/> (revision 19:10, 17 September 2016)
12. Wikipedia: *V-Model*, <https://en.wikipedia.org/wiki/V-Model> (revision 11:35, 24 October 2016)
13. PowerShell Team: *PowerShell on Linux and Open Source!*, <https://blogs.msdn.microsoft.com/powershell/2016/08/18/powershell-on-linux-and-open-source-2/> (revision 12:40, 19 September 2016)
14. Wikipedia: *Load balancing (computing)*, <https://en.wikipedia.org/wiki/Load_balancing_(computing)> (revision 21:10, 25 September 2016)
15. Wikipedia: *Denial-of-service attack*, <https://en.wikipedia.org/wiki/Denial-of-service_attack> (revision 21:06, 25 September 2016)
16. Wikipedia: *Multitenancy*, <https://en.wikipedia.org/wiki/Multitenancy> (revision 19:05, 29 October 2016)
17. Wikipedia: *Same-origin policy*, <https://en.wikipedia.org/wiki/Same-origin_policy> (revision 19:40, 29 October 2016)
18. Peep Laja: *First Impressions Matter: The Importance of Geat Visual Design*, <http://conversionxl.com/first-impressions-matter-the-importance-of-great-visual-design/> (revision 19:22, 30 October 2016)
19. W3Techs: *Usage of server-side programming languages for websites*, <https://w3techs.com/technologies/overview/programming_language/all> (revision 10:08, 7 November 2016)
20. The Scripting Guys: *Backwards Compatibility in PowerShell*, <https://blogs.technet.microsoft.com/heyscriptingguy/2015/09/14/backwards-compatibility-in-powershell/> (revision 18:04, 7 November 2016)
21. Wikipedia: *Spike (software development)*, <https://en.wikipedia.org/wiki/Spike_(software_development)> (revision 18:23, 21 November 2016)
22. Edmunds: *Depreciation Infographic: How Fast Does My New Car Lose Value?*, <http://www.edmunds.com/car-buying/how-fast-does-my-new-car-lose-value-infographic.html> (revision 19:47, 21 November 2016)
23. Rövidítések jegyzéke

|  |  |
| --- | --- |
| HTML | HyperText Markup Language |
| IE | Internet Explorer |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol Secure |
| REST | Representational State Transfer |
| API | Application Programming Interface |
| URL | Uniform Resource Locator |
| URI | Uniform Resource Identifier |
| XML | Extensible Markup Langugage |
| PaaS | Platform as a Service |
| PS | PowerShell |
| GUI | Graphical User Interface |
| AJAX | Asynchronous JavaScript and XML |
| CORS | Cross-Origin Resource Sharing |
| UX | User eXperience |
| FMEA | Failure Mode and Effect Analysis |
| MVP | Minimum Viable Product |
| JS | JavaScript |
| CSS | Cascading Style Sheets |
| PHP | PHP: Hypertext Processor |
| ISTQB | International Software Testing and Qualification Board |

Függelék