
**Budapesti Műszaki és Gazdaságtudományi Egyetem**
Villamosmérnöki és Informatikai Kar

Gál Emese

# ‘SKELETON’ TESZT KERETRENDSZER MEGVALÓSÍTÁSA AUTOMATIKUS TESZTELÉSHEZ

Üzemmérnök-informatikus szak

KONZULENS

Dr. Farkas Károly

Malinkó Péter

BUDAPEST, 2021

MŰEGYETEM 1782

SZAKDOLGOZAT FELADAT

**Gál Emese**

szigorló üzemmérnök-informatikus hallgató részére

# ‘Skeleton’ teszt keretrendszer megvalósítása automatikus teszteléshez

Napjainkban már az élet minden területén elengedhetetlen, hogy távolról tudjuk az ügyeinket intézni. A bankok és egyéb szolgáltatók számára lehetőség nyílt, hogy távolról azonosítsák az ügyfeleiket, ami azt is jelenti, hogy pl. egy bankszámla nyitásához már nem szükséges sorba állni a fiókban, hanem bármely mobiltelefonról/laptopról távolról lehetőség van erre. A tervezés és megvalósítás során a Contum Kft. saját és külföldi gyártók/partnerek dobozos termékeit (elektronikus aláírás, elektronikus dokumentum tárolás, hiteles archiválás) integrálja komplex üzleti megoldásaiba. Ahhoz, hogy a lehető leggyorsabban biztosítani lehessen a piacra lépést, valamint egy már meglévő termék továbbfejlesztése során előálló új verziók minőségbiztosításához a tesztelés automatizálása elengedhetetlen.

A hallgató feladata egy olyan Selenium alapú e2e automatikus teszt keretrendszer ‘skeleton’ (váz) létrehozása, ami kiinduló pontja lehet egy tetszőleges, új alkalmazás teszt automatizálásának, valamint annak bemutatása, hogyan lehet ezt alkalmazni egy új alkalmazás tesztelésére. Így a hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a távoli aláírásos és azonosítási technológiákat, EU-s és a magyar szabályozásokat
- Elemezze, milyen komponensek szükségesek és hasznosak egy Selenium keretrendszerben
- Valósítson meg a fentiek alapján egy teszt keretrendszer ‘skeleton’-t, és alkalmazza azt egy új alkalmazásra
- Elemezze a ‘skeleton’-ból felépített valós automatikus teszt rendszer megvalósításához szükséges lépéseket, tanulságokat, előnyöket, használhatóságot

**Egyetemi témavezető:** Dr. Farkas Károly, egyetemi docens, BME-VIK HIT tanszék

**Ipari konzulens:** Malinkó Péter, ügyvezető, Contum Kft.

Budapest, 2021. február 15.

Dr. Imre Sándor

egyetemi tanár
tanszékvezető

**Témavezetői vélemények:**

Egyetemi témavezető: ☐ Beadható, ☐ Nem beadható, dátum: aláírás:

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

1117. Budapest, Magyar tudósok körútja 2. I. Ép. B. 121.
Telefon: 463-3261 Fax: 463-3263
URL: http://www.hit.bme.hu

# Tartalomjegyzék

# HALLGATÓI NYILATKOZAT

Alulírott **Gál Emese**, 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, 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, 2021. 05. 23.

...…….
Gál Emese

# Összefoglaló

Egy alkalmazás minőségbiztosításában kulcsfontosságú szerepet játszik a tesztelés. Nagyobb projekteknél elengedhetetlenné vált a tesztek automatizálása, mivel ezzel gyorsabb és megbízhatóbb tesztelést biztosíthatunk. Ennek megoldására számos eszközt fejlesztettek ki, hogy segítsék a tesztelők és fejlesztők munkáját. Ezek közül a Selenium az egyik leggyakrabban használt megoldás webes felületekhez.

Az automatizált tesztelés gyors bevezetésének segítésére készítettem egy teszt keretrendszer vázat. Ebből kiindulva egyszerűbben lehet felépíteni különböző alkalmazások automatizált tesztjeit. A keretrendszer váz az elektronikus aláírással kapcsolatos funkciók ellenőrzésén alapul, de más projektekhez is könnyedén felhasználható. Ez a váz a legfontosabb és legalapvetőbb funkciókon túl tartalmaz még egyéb hasznos elemeket is, mint a statisztikák készítése és párhuzamos futtatás.

Dolgozatomban először körbejárom a távoli aláírásos és azonosítási technológiákat. Ezután megvizsgálom, hogy milyen komponensek szükségesek egy Selenium keretrendszer összeállításához. Majd részletesen bemutatom az elkészített teszt keretrendszer váz szerkezetét és felépítésének folyamatát. Majd a váz alkalmazásából származó eredményeket és tanulságokat összegzem és elemzem.

# Abstract

Testing plays a key role in ensuring the quality of an application. For larger projects, automating tests has become essential to ensure faster and more reliable testing. To address this, a number of tools have been developed to help testers and developers. Among these, Selenium is one of the most commonly used solutions for web interfaces.

To help the rapid implementation of automated testing, I created a test framework frame. It provides a simple starting point for building automated tests for different applications. The framework frame is based on the verification of electronic signature functions, but can be easily used for other projects. In addition to the most important and basic functions, this frame also includes other useful elements such as statistics generation and parallel execution.

In my thesis, I will first look at remote signature and identification technologies. I will then examine what components are required to build a Selenium framework. I will then describe in detail the created frame structure and the process of building the test framework. I will then summarize and analyze the results and lessons learned from the application of the frame.

# 1 Bevezetés

A távoli aláírás és távoli azonosítás egyre fontosabb szerepet tölt be életünkben. Nem csupán vállalatok, hanem magán emberek számára is egyre több lehetőség áll rendelkezésre távoli ügyintézéshez. Ma már néhány kattintással aláírhatunk elektronikusan szerződéseket akár telefonunkról is. Ezen technológiák népszerűsége a 2019-es világjárványnak köszönhetően ugrásszerűen megnövekedett, mivel rákényszerültünk használatukra. A hagyományos papíralapú ügyvitellel szemben az elektronikus megoldásoknak több előnye is van: megbízható, gyors és egyszerű.

A szoftver fejlesztési folyamatok elengedhetetlen része a tesztelés. Célja, hogy kiszűrje a fejlesztés során fennmaradt hibákat, ezzel növelve a termék minőségét. Számos típusa létezik, melyeknek más-más a funkciója. A tesztelési folyamat megkönnyítésére és meggyorsítására alkalmas eszköz az automatizált tesztelés. Ennek lényege, hogy gép végzi el a tesztelés lépéseit emberek helyett. Alkalmazásával megbízható és gyors teszteket készíthetünk.

A Contum Kft. távoli ügyintézést biztosító nemzetközi termékek integrálást végzi saját megoldásaiba, ezzel biztosítva ügyfeleinek elektronikus aláírást, elektronikus dokumentum tárolást és hiteles archiválást. Feladatom a cég számára egy teszt keretrendszer váz (’skeleton’) megvalósítása automatikus teszteléshez, melynek célja, hogy megkönnyítse más projektek teszt automatizálásának megvalósítását. Ennek megoldását Selenium segítségével oldottam meg, amely webes felületek automatikus teszteléséhez alkalmas módszert biztosít.

# 2 Irodalomkutatás, használt technológiák

## 2.1 Jogi háttér

Egy elektronikus aláírásos vagy azonosítási folyamatnak számos törvénynek, rendeletnek és szabályzatnak meg kell felelnie ahhoz, hogy a bíróság előtt is megállja a helyét. Ezen szabályok közül a két legfontosabb az Európai Parlament és a Tanács 910/2014/EU rendelete, továbbiakban eIDAS (electronic IDentification, Authentication and trust Services) rendelet és a 2015. évi CCXXII. Törvény, továbbiakban magyar E-Ügyintézési-törvény.

### 2.1.1 eIDAS

Ebben a rendeletben többek között szó van az elektronikus azonosításról és bizalmi szolgáltatásokról. Az egész Európai Unió területén egységessé tette az elektronikus aláírás követelményeit és elfogadását. Megfogalmazása előtt problémát jelentett a tagállamok közötti el nem fogadás is, vagyis amit az egyik országban elfogadtak elektronikus aláírásként, azt máshol nem feltétlenül. 2014. szeptember 17-én lépett hatályba. Az 1999/93/EK rendeletet hatályon kívül helyezte, ami csak az elektronikus aláírásra vonatkozó szabályzat volt. Legfőbb célja az emberek bizalmának növelése az elektronikus tranzakciókban [1] [2].


**1. ábra: eIDAS [3]**

Az elektronikus aláírás bizonyos feltételekkel jogilag egyenértékűnek tekintendő a papíron lévő kézzel írt aláírással a rendelet szerint. Ez alapozza meg a távoli aláírásos technológiák létjogosultságát. Az eIDAS által szabályozott bizalmi szolgáltatások közé tartozik az elektronikus aláírás, elektronikus bélyegző, elektronikus időbélyegzők, minősített digitális tanúsítványok és weboldal-hitelesítő tanúsítványok létrehozása, ellenőrzése és érvényesítése. A bizalmi szolgáltatások lehetnek minősítettek vagy nem minősítettek. A minősített bizalmi szolgáltatások magasabb jogi bizonyító erőt képviselnek. Például a minősített aláírással létrehozott dokumentum teljes bizonyító erejű magánokiratnak számít. Azt is meghatározza a rendelet, hogy milyen feltételekkel lehet valaki minősített bizalmi szolgáltató. Csak ők adhatnak ki minősített tanúsítványt, amely egy hiteles forrásból származó igazolás, arról, hogy egy adat egy adott személyhez tartozik. Ilyen tanúsítvánnyal minősített aláírás (legbiztonságosabb típus) hozható létre elektronikusan. Az e-aláírásoknak több biztonsági szintje van melyekkel különböző típusú iratokat lehet létrehozni. Ezeket egy későbbi fejezetben tárgyalom részletesen. A bizalmi szolgáltatók listáját minden EU tagállamnak közzé kell tennie.

### 2.1.2 Magyar E-ügyintézési törvény

A magyar elektronikus ügyintézési törtvény szabályozza az elektronikus ügyintézést és a bizalmi szolgáltatásokat az eIDAS rendeletnek megfelelően. Azonban van egy eltérés a magyar és EU-s jogszabályok között. Magyarországon létezik egy minősített tanúsítványon alapuló fokozott biztonságú aláírás típus is, melyet az eIDAS nem ismer. Ezzel az aláírás fajtával bizonyító erejű magánokiratot lehet létre hozni Magyarországon, de ezt az EU nem fogadja el [3].

## 2.2 Távoli aláírásos és azonosítási technológiák

### 2.2.1 Elektronikus aláírás

Számos előnnyel jár, ha elektronikus aláírást használunk a hagyományos papírra történő kézírás helyett. Pénzt és időt spórolhatunk vele, mivel nem kell iratokat nyomtatni, másolni és postázni. Gyorsabban lehet keresni a dokumentumokban. Magyarországon elérhető a NISZ Zrt. AVDH-szolgáltatása, mellyel olyan személyek is aláírhatnak elektronikusan dokumentumokat, akiknek nincs elektronikus aláírásuk.

Az aláírással, legyen az hagyományos vagy elektronikus az aláíró személy elfogadja az adott dokumentum tartalmát, egyet ért azzal. Van két kifejezés, melyeket sokszor kevernek a köznyelvben, pedig két különböző dologról van. Ez az elektronikus aláírás és a digitális aláírás fogalma. Míg az előbbi jogi fogalom, utóbbi egy technológiai kifejezés. [4] Az eIDAS fogalommeghatározása szerint az elektronikus aláírás: „olyan elektronikus adat, amelyet más elektronikus adatokhoz csatolnak, illetve logikailag hozzárendelnek, és amelyet az aláíró aláírásra használ” [1].

Az eIDAS három féle elektronikus aláírás típust különböztet meg: ’Egyszerű’ Elektronikus Aláírás (’Simple’ Electronic Signature, SES), Fokozott biztonságú elektronikus aláírás (Advanced Electronic Signature, AES) és Minősített elektronikus aláírás (Qualified Electronic Signature, QES).

A SES a legalacsonyabb hatáskörű e-aláírás, ezeknek nincs törvényes bizonyító erejük. Ilyen aláírásnak számít a szkennelt aláírás és a FAX, illetve a jelölő négyzet és nyilatkozat is.

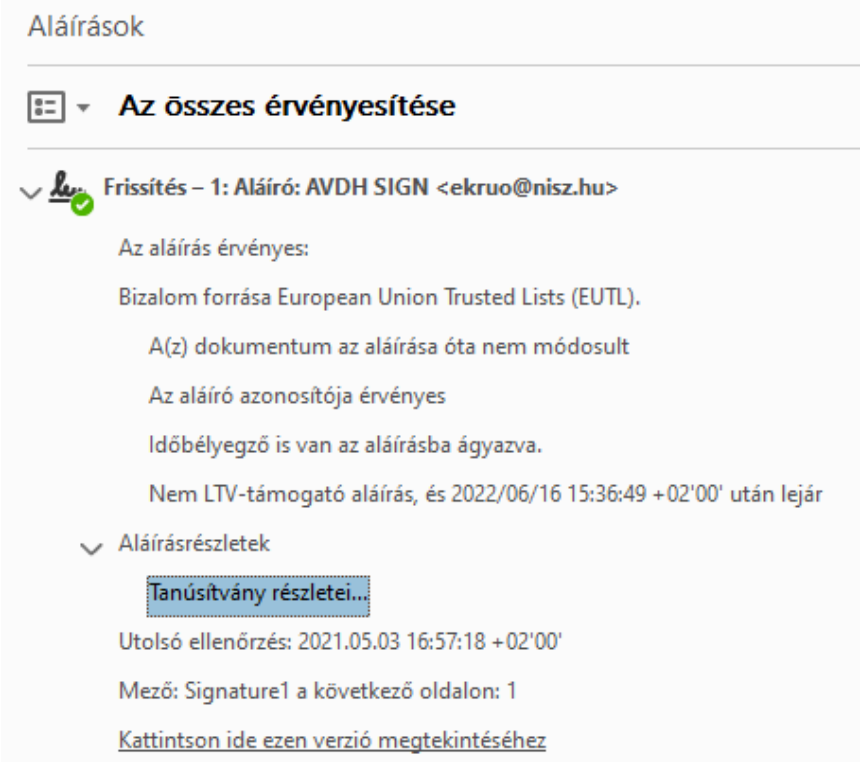
Az AES-re jellemző, hogy kizárólag az aláíróhoz köthető és alkalmas az aláíró azonosítására. Amennyiben az aláírást követően valaki megváltoztatja a dokumentum tartalmát az láthatóvá válik. Ellenőrzéssel ezt ki lehet szűrni. Ilyen biztonsági szintnek felel meg az OTP (One Time Password), ami egyszeri bejelentkezésre jogosít fel.

A legbiztonságosabb e-aláírás fajta a QES. Magába foglalja az AES minden követelményét és ezen felül még több fontos tulajdonsága van: minősített tanúsítványra épül és biztonságos aláírás-létrehozó eszköz (pl. chipkártya) segítségével lehet létrehozni. Amennyiben egy dokumentumot ilyen aláírással látnak el, az teljes bizonyító erejű magánokiratnak számít. Amennyiben az egész EU területén érvényes okiratot szeretnék létrehozni, ezt az aláírást használjuk. Ilyen szintű aláíráshoz leggyakrabban használt technológia a PKI (Public Key Infrastructure), vagyis a nyilvános kulcsú infrastruktúra. Ez a módszer tanúsítványokat és kriptográfiai kulcsokat használ.

Az elektronikus aláírások hitelességének ellenőrzését különböző aláírás-ellenőrző programokkal tehetjük meg. Általánosan egy ilyen ellenőrző program több bonyolult lépést hajt végre az ellenőrzés érdekében.

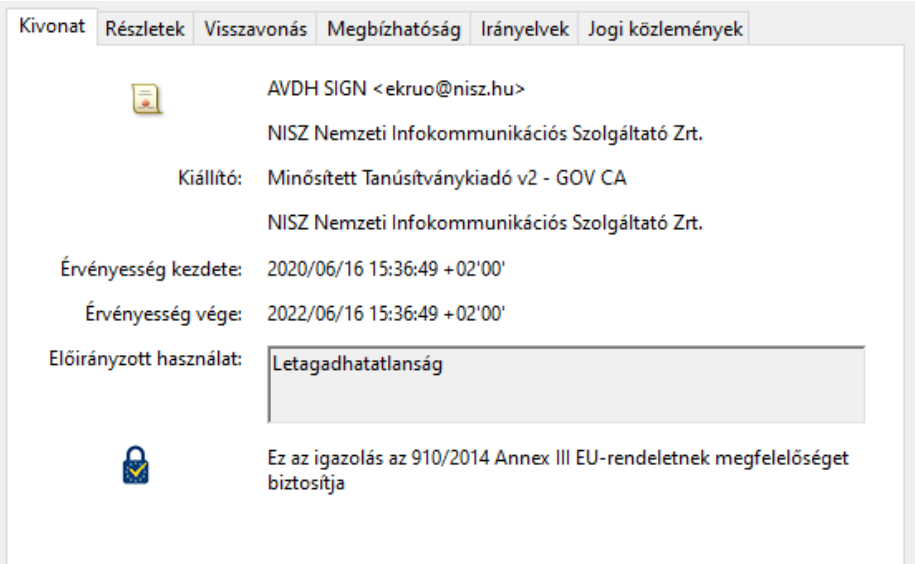
Az aláírás ellenőrzést gyakorlatban megtehetjük például úgy, ha a dokumentumot Adobe Reader segítségével nyitjuk meg. Egy AVDH-val aláírt dokumentum esetén ebben a pdf olvasóban a csatolmányok között találnunk kell egy igazolást, mely biztosítja, hogy

elektronikus bélyegzővel és időbélyegzővel látták el a dokumentumot és az aláíró személyt azonosították. Az igazolás nélkül érvénytelen az aláírás. Egy pdf aláírásának hitelességét az aláíráspanelen lehet ellenőrizni, ahogy az 2. ábra szemlélteti.




**2. ábra: Aláírópanel**

A 3. ábra mutatja, hogyan tudjuk ellenőrizni a tanúsítvány részleteit. Minden fontos adatot meg lehet itt tekinteni, többek között a tanúsítványt kiállító szervezetet és az érvényességi idejét. Ugyan itt a részletek fülön megnézhető az aláírási algoritmus típusa és a nyilvános kulcs is számos más információ mellett.




**3. ábra: Tanúsítvány részletei**

### 2.2.2 Elektronikus azonosítás

Fontos tisztázni két gyakran használt kifejezés közötti különbséget. Az elektronikus azonosítás és az elektronikus hitelesítés két különböző eljárás, melyeket nem szabad összekeverni. Az eIDAS rendelet szerint az elektronikus azonosítás „a természetes vagy jogi személyt, illetve jogi személyt képviselő természetes személyt egyedileg azonosító, elektronikus személyazonosító adatok felhasználásának folyamata” [1]. Az azonosítás során a bemutatott okmány valódiságát vizsgálják, illetve, hogy a bemutatott okmány valóban az okmánybirtokosé. Ezzel szemben az elektronikus hitelesítés „olyan folyamat, amely lehetővé teszi a természetes vagy jogi személy elektronikus azonosításának, vagy az elektronikus adatok eredetének és sértetlenségének az igazolását” [1]. Tehát a hitelesítés célja annak megállapítása, hogy az ügyfél valóban az az egyén, akinek mondja magát, akit korábban a szolgáltató már azonosított.

### 2.2.3 Személyazonosítás

Egy személyazonosítási folyamat célja, hogy ellenőrizzük, hogy egy személy valóban az-e, akinek mondja magát. Többféle szempontból vizsgálhatjuk az egyes személyazonosítási technikákat. Ennek egyik aspektusa az aktivitás. Amennyiben szükség van az azonosítandó személy közreműködésére, aktív módszerről beszélünk. A retina szkennelés vagy ujjlenyomat alapú azonosítás is ilyen aktív eljárás. A közreműködést nem igénylő technikák pedig a passzív csoportba tartoznak. Passzív módszer alkalmazása során az azonosítandó személy nem feltétlenül tud az azonosítási folyamatról. Ez bűnügyi nyomozások során nagyon hasznos lehet. Az arcfelismerés videóképről és a hangazonosítás is ebbe a kategóriába tartozik [5].

Az azonosításhoz felhasznált információ szerint három alapvető módszer létezik. Ezek közül az egyik a tudás alapú azonosítás. Ilyen esetben az azonosítandó személy rendelkezik olyan információval, amit az eljárás során ellenőrizni lehet. Ez lehet például jelszó vagy PIN kód. Birtok alapú azonosítási módszer esetében az azonosítandó személynek birtokolni kell egy eszközt, ami lehet egy kulcs, vonalkód, chipkártya vagy smartcard is. A biometrikus azonosítás során a személy egy fizikai vagy biológiai jellemzőjét használják fel. Többek között ilyen attribútum az ujjlenyomat, arc, hang vagy retina. Mindhárom módszernek vannak hátrányai, ami miatt önmagában egy ilyen módszer alkalmazása nem elegendő a biztonságos azonosításhoz. Legtöbb esetben két vagy három különböző elvű módszert kell alkalmazni a megfelelő biztonság érdekében. Távoli azonosításra a biometrikus módszer a legalkalmasabb és legbiztonságosabb. Habár már 5 évvel ezelőtt is bebizonyították, hogy ujjlenyomatot lopni nem hatalmas feladat. A Chaos Computer Club egyik tagja sajtótájékoztatón szerezte meg a német védelmi miniszter ujjlennyomatát. Mindössze nagyfelbontású képeket kell készíteni az ujjakról és ezeket feltölteni egy ingyenes szoftverbe, ami rekonstruálja a nyomatot [6].

### 2.2.4 Biometrikus azonosítás

Lehet viselkedésbeli jellemzőket vizsgálni, mint a járás, beszéd, kézírás, gépelés, testtartás. A járás elemzésnél alapvetően két különböző technikát alkalmaznak. Holisztikus eljárás esetén foltokat, sziluetteket vizsgálnak statisztikai módszerekkel. Parametrikus vizsgálatnál konkrét fiziológiai paraméterek mérése történik. Testalkat elemzésnél méréseket lehet végezni pontpárokból álló sajátságvektorok segítségével: magasság, végtaghossz, fejméret. Mozgóképről is lehet adatot szerezni - átlagolással javítható a pontosság. Ilyenkor mozgó személy sziluettjéből nyerik a sajátságvektort és átlagolnak.

Másik lehetőség, hogy fizikai jellemzőket vizsgálnak, mint a retina, írisz, ujjlenyomat, arcfelismerés, DNS, érhálózat, kézgeometria, fogazat, hang. A retina szkennelés feltérképezik a szemgolyó legbelső rétegének falát. Ehhez alacsony intenzitású infravörösfényt használnak, mert a vérerek máshogy nyelik el a fényt és nem károsítja a szemet. Ez a módszer rendkívül nagy pontossággal működik, ezért leginkább magas biztonsági szintet igénylő helyeken használják. Írisz szkennelésnél néhány méter távolságból egy nagyfelbontású kamera tapogatja le a szivárványhártya egyedi mintázatát (több száz egyedi tulajdonság, 6x több egyedi részlet, mint az ujjlenyomatnál). Ehhez

infratartományban működő CCD kamerát használnak. Az arcfelismerés az emberi arc karakterisztikáját (a szemgödrök pozícióját, a szemek távolságát, a száj elhelyezkedését és hasonlókat) használja azonosításra. Ezt lehet 2D-s módszerrel vizsgálni. Ez azt jelenti, hogy képet készítenek kamerán keresztül. 3D-s eljárás, amikor lézerfénnyel letapogatják az arcot és erről 3D-s modellt készítenek.

### 2.2.5 Ügyfélátvilágítás

A pénzmosás és a terrorizmus finanszírozása megelőzéséről és megakadályozásáról szóló 2017. évi LIII. törvény (a továbbiakban: Pmt.) kötelezően előírja a Hitelintézetek számára az ügyfelek átvilágítását. Ezt akkor kell végrehajtani, ha üzleti kapcsolatot létesít az ügyfél vagy bizonyos összeget meghaladó pénzátutalás történik vagy bizonyos összeget meghaladó ügyleti megbízást hajtanak végre. Az átvilágítási folyamat során ellenőrizni kell a személyazonosságot, rögzíteni kell a személyes adatokat, az azonosságot igazoló okirat érvényességét ellenőrizni kell, majd erről az okiratról másolatot kell készíteni és két nyilatkozatot kell kitöltenie az ügyfélnek: Tényleges tulajdonosi nyilatkozat és Kiemelt közszereplői (PEP) nyilatkozat [7].

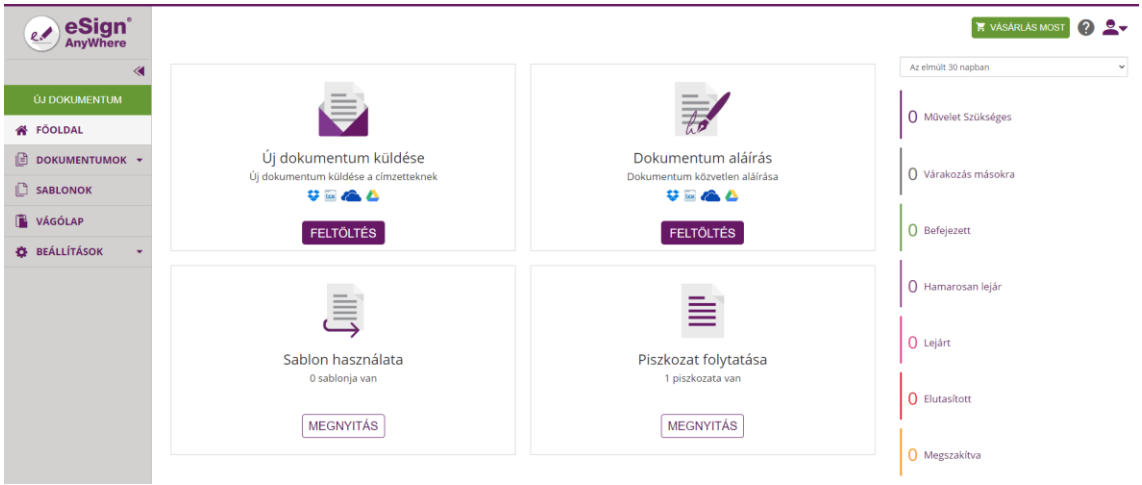
Ezeknek a folyamatoknak két típusát különböztetjük meg, melyek különböző szolgáltatási kör elérését teszik lehetővé. Egyik verzió a valós idejű ügyfél- átvilágítás, amelyben az ügyfél videón, élő emberi közreműködéssel, valós időben kommunikációt folytat. Ez gyakorlatban úgy történik, hogy az ügyfél és a hitelintézet egy munkatársa online videó kapcsolaton keresztül kommunikálnak. A személyt azonosító okiratokról webkamerán keresztül képet készítenek és ellenőrzik azokat. Ez teljeskörű ügyfél-átvilágításnak felel meg. Napjainkban egyre több helyen alkalmazzák ezt az azonosítási formát, mivel gyors, egyszerű és biztonságos. Ennek segítségével például külföldről is nyithatunk számos magyar banknál számlát akár pár perc alatt.

A nem valós idejű ügyfél-átvilágítás korlátozott szolgáltatási kört tesz lehetővé. Ez esetben élő emberi közreműködés nélkül történik az azonosítás. Mindkét fajta átvilágítás auditált elektronikus hírközlő eszközön keresztül végzett folyamat kell, hogy legyen a Pmt. által előírtan. Auditált elektronikus hírközlő eszköz meghatározása: “az ügyfél távoli, elektronikus adatátviteli csatornán történő átvilágítására, az ügyfél nyilatkozatainak megtételére, az ügyfél által tett nyilatkozat értelmezésére, biztonságos tárolására, a tárolt adatok visszakeresésére és ellenőrzésére alkalmas auditált elektronikus rendszer „ [1].

### 2.2.6 eSignAnyWhere

A Namirial egy távoli aláírásos és távoli azonosítási termékeket szolgáltató vállalat. Az eIDAS szerinti minősített bizalmi szolgáltatónak minősül, aki minősített aláírásokat, időbélyegeket és videós azonosítást biztosít a felhasználók számára. A Contum Kft. gyakran integrálja a Namirial termékeit a saját megoldásaiba. Az elektronikus aláírást a Namirial eSignAnyWhere (továbbiakban eSAW) alkalmazás segítségével biztosítja [8].

Az eSAW a felhasználóknak nyújtott szolgáltatásai közé tartozik az aláírási folyamatok elindítása és dokumentumok elektronikus aláírása. Egy ilyen eljárás indítása során mindig van egy küldő, aki elindítja a folyamatot és megjelöli az aláírókat. Egy feltöltött dokumentumot több személlyel is alá lehet íratni. A rendszer értesítéseket küld a megjelölt személyeknek e-mail-ben, tájékoztatva őket, hogy aláírandó dokumentumuk keletkezett. Az összes résztvevő minden lépésről értesít kap. Jelzi a rendszer, ha valaki aláírta a dokumentumot. Az eSAW-nak köszönhetően néhány kattintással alá tudunk írni elektronikusan dokumentumokat. A rendszer eltárolja a múltban aláírt, elutasított vagy még aláírásra várakozó elemeket és ezeket vissza lehet nézni. Az 4. ábra prezentálja az eSAW felületét.




**4. ábra: eSignAnyWhere**

### 2.2.7 Elektronikus aláírási folyamat

A Contum Kft. által biztosított teljes aláírási folyamatot két fő komponens biztosítja: Namirial termékek (eSAW, Significat server) és saját fejlesztésű aláírási folyamatvezérlő. Az előbbiek biztosítják a tényleges elektronikus aláírást. A folyamatvezérlő köti össze az eSAW-t és az ügyfelek alkalmazását. Felelős az aláírandó dokumentumok kezeléséért, előkészítéséért és át adásáért a Namirial termékeknek, illetve értesítések küldéséért is.

## 2.3 Szoftvertesztelés

Egy szoftver fejlesztése során több fázison esik át mire megérkezik a felhasználókhoz vagy a megrendelőhöz. Különböző módszertanok alapján történhet a fejlesztés a projekt típusának megfelelően. Ilyen módszerek közé tartozik például a vízesés modell, a V-modell, a RUP módszertan. Ezeknek a módszereknek különböző fázisaik vannak, de általánosságban véve szeretnék röviden áttekinteni egy szoftver fejlesztési folyamatot. A teljes eljárás legelején nagyon fontos a követelmények meghatározása. A fejlesztő csapatnak tisztázni kell a megrendelővel a ráfordítható idő és költség keretet, a teljesítményt és a minőséget. Ezután a követelmények alapján meg kell tervezni a szoftvert. Ebben a szakaszban fejlődik ki a szoftver felépítése. Kialakítják a rendszer egységeit és azok kapcsolatait. Ezután következik az implementáció a terveknek megfelelően. Itt öltenek testet a kívánt funkciók. Mivel nincs tökéletes szoftver ezért az elkészült kódot át kell nézni és ki kell szűrni a hibákat. Ezek például lehetnek implementációs hibák vagy rossz kommunikációból eredő hibák. Majd le kell tesztelni a szoftvert, hogy kiderüljenek a fennmaradó hibák. Ez biztosítja a minőség és megbízhatóság növelését. Mindezek után következik a telepítés és üzembe helyezés. Sikeres átadást követően az utolsó és leghosszabb fázis jön, az üzemeltetés és karbantartás. Ennek során javítani kell az esetlegesen felmerülő hibákat, a változó környezetnek megfelelően illeszteni a rendszert [9].

### 2.3.1 A tesztelés alapjai

Az emberek által írt kódban biztos, hogy lesz hiba. Főleg, ha egy összetett programról vagy rendszerről van szó. A tesztelés ezeket a hibákat hivatott felfedezni még az üzembe helyezés előtt. Minél előbb fedezzük fel a hibákat, annál több erőforrást spórolhatunk meg. A szoftver tesztek között különböző típusokat különböztethetünk meg,

melyeknek különböző célja van. Léteznek komponens tesztek, melyek a rendszer egyetlen különálló részét analizálják. Az integrációs tesztek a komponensek közötti összeköttetéseket vizsgálják. A kész szoftvert rendszer tesztekkel lehet ellenőrizni. Ennek során általában egy külső cég megnézi, hogy a termék ellátja-e a kívánt funkciókat és megfelel-e a követelményeknek. Az ellenőrző folyamat végén történik meg az átvételi teszt, melynek keretében több különböző tesztet hajtanak végre. Egyrészt alfa tesztet végeznek, melynek során a fejlesztő cég emberei is próbára teszik a programot. Másrészt a felhasználók egy kis része is kipróbálja a terméket a béta teszt keretében. Ezt követően a felhasználók szélesebb körében is tesztelik a szoftvert és a rendszergazdák is átvizsgálják a rendszer helyes működését. Legvégül következik a regressziós teszt, amelyet akkor végeznek, ha valamilyen változtatás, fejlesztés történik a terméken. Ez a rendszer helyes működését vizsgája [9] [10].

Az end-to-end teszt feladata, hogy egy alkalmazás használata során felmerülő folyamatok szabályos működését ellenőrizze az elejétől a végéig. Ennek keretében tesztelik a helyes kommunikációt az adatbázissal és a hálózattal. Vizsgálják, hogy jól funkcionál-e a külső rendszerekkel való együttműködés [11].

### 2.3.2 Automatizált tesztelés

Egy szoftver tesztelését manuálisan és automatizálva is el lehet végezni. A manuális tesztelés során egy embernek kell ezt elvégezni. Például végig kell próbálni, hogy minden funkció működik-e egy termékben. Webes felületek esetben gyakorlatilag végig kell kattintgatni az egész felhasználói felületet. Automatizált tesztelésnél ezt a munkát a gép végzi. Ekkor előre megírt teszteket futtatunk.

A manuális ellenőrzésnek több hátránya is van. Sok időt vesz igénybe és könnyű hibát véteni vagy kifelejteni valamilyen funkciót, mivel monoton munkáról van szó. Vannak teszt típusok, amihez nem használható módszer (pl. terheléses tesztekhez). A manuális tesztelés előnyei közé tartozik, hogy a megváltozott környezethez könnyen alkalmazkodik.

Az automatizált tesztelésnek több előnye van, mint hátránya. Igaz, hogy több időbe kerülhet a tesztek implementálása, de ez hosszú távon megtérül. Szaktudást igényel, de cserébe könnyen karbantartható teszteket kapunk. Sűrűn futtatott teszteknél különösen hasznos, mivel sok időt spórolhatunk meg. Természetesen nem minden esetben

lehetséges vagy előnyös a bevezetése. Gyakran változó funkciók vagy felhasználói felületek esetére nem alkalmasak az automatizált tesztek.

Kisebb projekteknél elegendő lehet a manuális tesztelés is. Nagyobb projekteknél viszont elengedhetetlen az automatizált tesztek bevezetése, amivel növelhetjük a tesztek megbízhatóságát és a tesztelési folyamat sebességét.

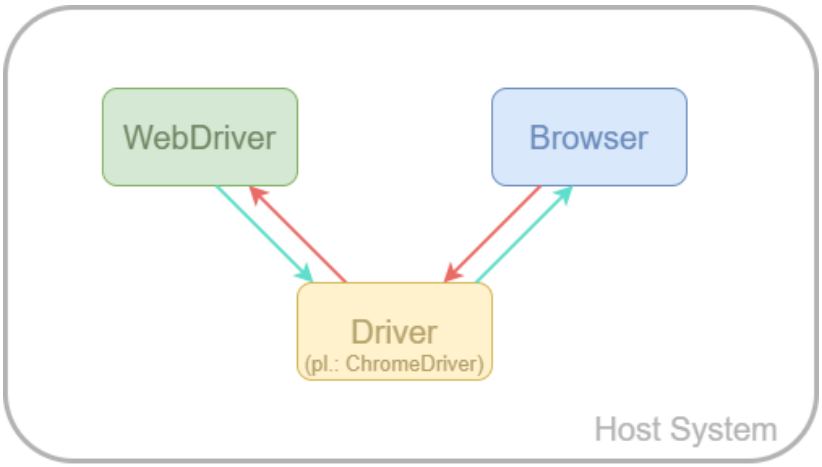
## 2.4 Használt technológiák

Ebben az alfejezetben röviden szeretném bemutatni azokat a technológiákat, amiket webes felületek automatizált teszteléséhez használtam. A Contum Kft. igényeinek megfelelően a teszt keretrendszer vázat és a teszteket Java nyelven implementáltam, ami tökéletesen megfelelt a célra, mivel minden szükséges komponens használatát támogatja.

### 2.4.1 Selenium

Az elérhető webes teszt keretrendszerek közül a Selenium tűnt a legjobb választásnak, mivel minden szükséges feladat elvégzésére alkalmas és ez az egyik legnépszerűbb eszköz, könnyen elsajátítható a használata és rengeteg oktató anyag áll rendelkezésre.

A Selenium 3 egy nyíltforráskódú automatikus tesztelési eszköz webalkalmazásokhoz. A segítségével könnyen imitálhatjuk egy felhasználó tevékenységét webes felületeken. Selenium tesztek futtatása során láthatjuk a böngésző felhasználói felületén zajló eseményeket. A kliens gépről lehet webdriverek segítségével vezérelni a böngészőt, amit a Selenim API-val (Application programming interface – Alkalmazás-programozási felület) tehetünk meg. A Selenium Web Driver egy összetett objektum-orientált API, ami nagyszerű eszköz lehet a manuális tesztelés helyettesítésére. Alkalmazásával a fejlesztés során elkövetett programozási hibák jelentős részét ki lehet szűrni, megfelelő tesztesetek használatával. Az 5. ábra szemlélteti a Selenium működését. A Selenium WebDriver nem közvetlenül a kliens böngészőjével kommunikál, hanem a különböző, böngésző specifikus browser drivereken keresztül (pl. ChromeDriver, GeckoDriver stb). A Selenium alkalmazásához le kell töltenünk egy Selnium Web Drivert és egy Browser Drivert a használni kívánt böngészőnknek megfelelő verziószámmal.




**5. ábra: Selenium [12]**

A Selenium nagy előnye, hogy ingyenes és könnyen használható. Emiatt sokan választják ezt az automatizáló eszközök közül, így számos segédlet, példa és oktató anyag áll rendelkezésünkre. Előnyei közé tartozik még az is, hogy sok különböző technológiával integrálható és használható (pl. Jenkins, Maven, Docker). A Selenium alapvető használatát a Contum Kft. által biztosított Udemy kurzusból tanultam meg [13].

Különböző programnyelveket támogat, többek között: Ruby, Java, Python, C#, JavaScript. Többféle operációs rendszerrel kompatibilis: Microsoft Windows, macOS, Linux. Különféle böngészők használatát támogatja: Firefox, Internet Explorer, Safari, Opera, Chrome, Edge. Mindig érdemes ellenőrizni a támogatott platformok verziószámát a dokumentációjukban, mielőtt elkezdjük a használatát [14].

A Selenium segítségével különböző típusú teszteket is automatizálhatunk. Lehetőség van regressziós tesztek automatikus futtatására, amit általában hiba javítás vagy valamilyen változtatás után szokás elvégezni. Ez ellenőrzi, hogy a rendszerünk még mindig megfelelően működik. Rendszertesztet is készíthetünk a Selenium segítségével, ami átvizsgálja a kész szoftver specifikációban megfogalmazott funkcióinak helyes működését. Ezeken túl integrációs és end-to-end teszteket is készíthetünk Seleniummal [15].

### 2.4.2 TestNG

A TestNG egy tesztelési keretrendszer, aminek sok fejlesztőkörnyezet és plugin támogatja a használatát. Sokféle teszt típus készítését és futtatását teszi lehetővé: egység teszt (unit teszt), integrációs teszt, end-to-end teszt. Párhuzamos futtatást biztosít, ami jelentősen lerövidítheti a tesztek végrehajtásához szükséges idejét. Naplózási funkciót kínál. Paraméterek

megadását is lehetővé teszi, amit a tesztek futásuk során használnak fel. Mindezek beállítását egy XML fájlban adhatjuk meg. A teszteket annotációkkal láthatjuk el (pl. @Regression). Ez olyan, mint egy címke, amit az egyes tesztekhez rendelünk, ezáltal csoportokba rendezhetjük azokat. Több teszthez is kapcsolhatjuk ugyan azt az annotációt és egy teszthez több annotációt is fűzhetünk. Ennek köszönhetően megmondhatjuk, hogy éppen mely annotációhoz tartozó teszteket szeretnénk futtatni. [16]

### 2.4.3 Cucumber

A viselkedés vezérelt fejlesztés (Behaviour-Driven Development - BDD) egy olyan szoftverfejlesztési módszer, ami segíti a kommunikációt a megrendelő, a fejlesztő és a tesztelő csapat között. A Cucumber egy viselkedés vezérelt fejlesztést támogató eszköz. Segítségével olyan teszt forgatókönyveket (scenario) írhatunk Gherkin nyelven, amit minden ember megérthet. Ez a szintaxis kulcsszavakat használ és biztosítja, hogy angol (más nyelveket is támogat) mondatokkal írhatunk meg egy futtatható teszt forgatókönyvet [17].

### 2.4.4 Apache Maven

Az Apache Maven a projektek menedzselésében segítséget nyújtó eszköz. Egységes módszert kínál a projektek felépítésére. Biztosítja a függőségek megosztás különböző projektek között. Használatával nem kell letöltenünk és hozzáadnunk a szükséges JAR fájlokat programunkhoz, elegendő a használni kívánt függőségek felsorolása egy XML fájlban (pom.xml). A felsorolt függőségeket (dependency) a Maven automatikusan letölti számunkra egy központi Repository-ból. Ez különösen hasznos, ha csapatban dolgozunk, mivel nem kell figyelnünk a függőségek egységes verziószámára. A POM (Page Object Model) fájl egyedi információkat tárol a projektünkről (pl. konfigurációs adatok). Ebből kiindulva épülnek fel a Maven projektek. Habár több programozási nyelvvel használható a Maven, leginkább Java projektekhez szokták alkalmazni. A Maven projekteket egyszerűen lehet futtatni parancssorból és különböző fejlesztőkörnyezetekből is. A Maven projektek előnyei közé tartozik, hogy könnyen integrálható Continuous Integration (CI) eszközökkel [18].

### 2.4.5 Docker

A Docker egy operációsrendszer szintű virtualizációt biztosító eszköz. Segítségével különböző környezetekben futtathatunk programokat úgy, hogy nem kell azokat telepíteni a gazdagépre. Segítségével széleskörűen tesztelhetjük a programokat különböző rendszerekben [19].

A Docker konténerek (container) biztosítják a programok futtatásához szükséges környezetet. Egy konténer létrehozásakor kap egy kis részt a gazdagép infrastruktúrájából (processzor, memória stb.). Konténereket képfájlokból (image) lehet létrehozni. Ezek határozzák meg a konténerek szerkezetét. A képfájl felépítését a Dockerfile tartalmazza. A Docker Hub-on előre létrehozott képfájlok vannak, amiket szabadon fel lehet használni.

Egy egyszerű konténer felépítésének folyamata tehát úgy néz ki, hogy először készíteni kell egy Dockerfile-t, majd ennek alapján építünk egy képfájlt. Az elkészült képfájlból létrehozhatunk különböző konténereket, amiben programokat futtathatunk.

# 3 Architektúra

A feladatom egy teszt keretrendszer váz készítése és alkalmazása volt automatikus webes teszteléshez. A váz építés célja, hogy abból kiindulva más projektek teszt keretrendszerét gyorsan és könnyen fel lehessen építeni.

## 3.1 Követelmények

A legalapvetőbb funkcionális elvárás a keretrendszer vázzal szemben a tesztek automatikus futtatása. Egy másik fontos követelmény a tesztek futásának részletes naplózása, mivel a megfelelő formátumú naplózás megkönnyíti a tesztelendő program hibáinak és annak okainak keresését.

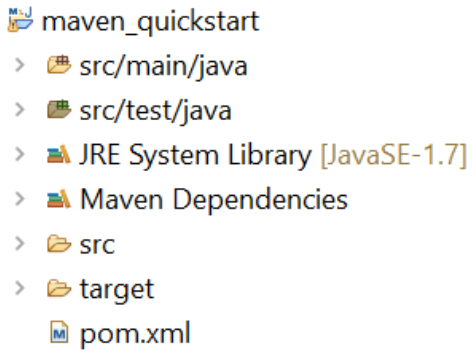
Az újrafelhasználhatóság a leglényegesebb nem funkcionális követelménye. Ennek oka az, hogy ha nem kell minden egyes új projektnél újra kitalálni és felépíteni a tesztkeretrendszer alapjait, időt és erőforrásokat takaríthatunk meg. A másik fontos szempont a bővíthetőség volt a tervezésnél. Mivel minden projekt különböző felépítésű ezért biztosítani kellett, hogy könnyű legyen az új osztályokat és metódusokat bevezetni. A cég számára az is fontos szempont volt, hogy különböző környezetekben is használható legyen ez a keretrendszer.

## 3.2 Selenium keretrendszer szükséges komponensei

### 3.2.1 Maven

Egy automatikus teszt keretrendszer váz alapjául érdemes Maven projektet használni, mivel ez segíti a függőségek kezelését és a projekt felépítését, futtatását. A Maven projektek létrehozásakor lehetőségünk van előre elkészített sablonokból (archetype) kiindulni, melyek megadják a projektünk alapfelépítését. Esetemben elég egy egyszerű *'maven-archetype-quickstart*' sablont használni, aminek a szerkezetét a 6. ábra illusztrálja. Amint az ábra is mutatja, a generált szerkezetbe bele tartozik egy XML fájl is, a pom.xml. Ebben adhatjuk meg a használni kívánt függőségeket és azok konfigurációit, illetve különböző projekt beállításoka. A src/main/java mappába kerülnek a projekt forrás állományok, a src/test/java mappába pedig a teszt forrás fájlok. Az alap konfiguráció szerint a target mappába kerül a projektek Maven build folyamatának minden kimenete (pl. logok, build folyamat végén a projektünkből létrehozott futtatható

JAR fájl). Amennyiben más könyvtárba szeretnénk tenni a kimeneteinket, lehetőségünk van ezt átállítani.




**6. ábra: Maven quickstart sablon felépítése**

### 3.2.2 Test framework - TestNG

A következő fontos komponens egy teszt keretrendszer építése esetében a ’test framework’ kiválasztása, ami lehetővé teszi a tesztek futtatását. Sok megoldás közül választhatunk, de a Seleniumhoz és a Java nyelvhez leggyakoribb választás a JUnit és a TestNG. Ezek közül a TestNG az az eszköz, ami több funkciót biztosít. Magába foglalja a JUnit szolgáltatásait, de kiegészíti azokat.

### 3.2.3 Selenium

#### 3.2.3.1 Web Driver

A böngészők automatizált használatához először le kell töltenünk a megfelelő verzió számú és típusú webdrivert. Majd a kódunkban be kell állítanunk a *system.setProperty()* metódussal minden driver típus elérési útvonalát. Ezekután példányosítanunk kell egy webdrivert a *WebDriver* osztályból. Ehhez mindig meg kell adnunk, hogy milyen browsert szeretnénk használni aktuálisan (Firefox, Chrome stb.). Ezután már átadhatunk a drivernek url-címet a *driver.get(url)* paranccsal és alkalmazhatjuk a korábban ismertetett metódusokat. Ezekre mutat példát az alábbi kódrészlet.

```
System.setProperty("webdriver.chrome.driver", "path of driver");
WebDriver driver=new ChromeDriver();
driver.get("https://www.google.com/");
```

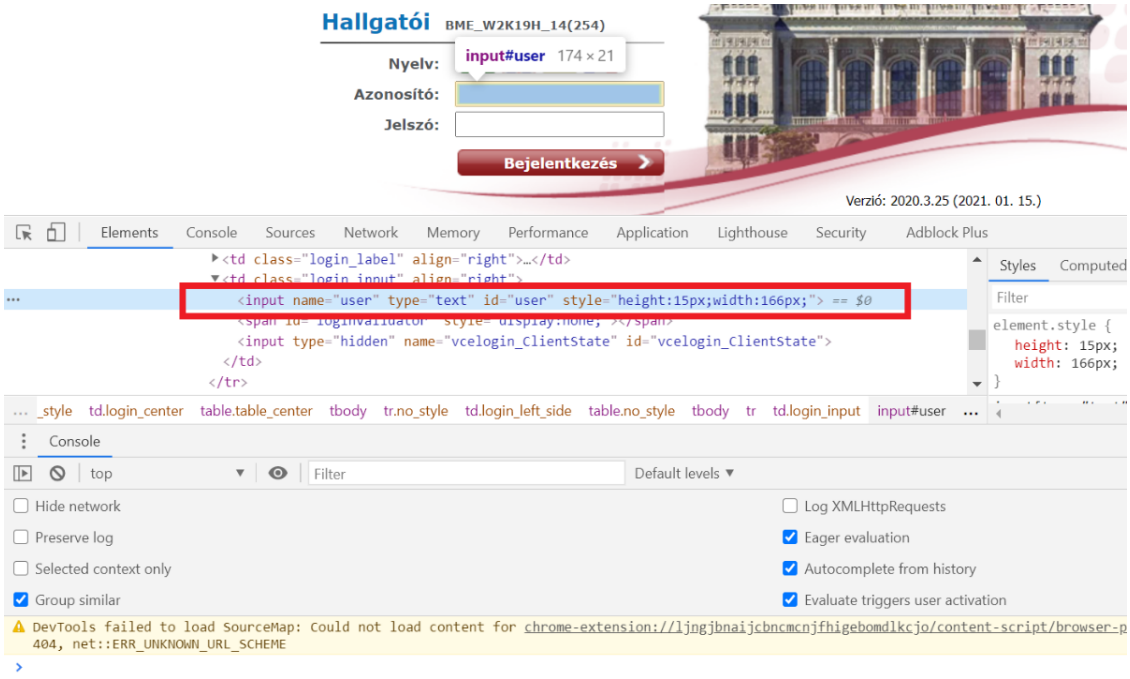
#### 3.2.3.2 WebDriver API

Egy böngészőben megnyitott weboldal web elemekből (*WebElement*) áll. Sok ilyen elemmel interakcióba léphet a felhasználó. Ezekre néhány példa: a szövegdobozba (*TextBox*) beírhat szöveget, a gombra (Button) klikkelhet, a jelölőnégyzetet (*Checkbox*) kipipálhatja. Az ilyen interakciók kódból való elvégzéséhez biztosít a Selenium metódusokat. Ezek közül szeretnék kiemelni és részletezni néhányat, amit gyakran használunk. A *findElement(By)* egy fontos függvény, ezzel azonosíthatunk egy web elemet. Ha megtaláltuk a keresett elemet, utána lekérdezhetjük az adatait, mint például: *isSelected()*, *getText()*. A *findElements(By)* metódus web elemek listájával tér vissza. A *sendKeys()* használatával szerkeszthető tartalmat lehet bevinni szövegmezőkbe a tesztek végrehajtása során. A *click()* metódussal lehet rákattintani az egyes, már lokalizált elemekre.

#### 3.2.3.3 Locator

A web elemek beazonosítása különböző tulajdonságok alapján lehetséges. Ezeket az attribútumokat hívják Locator-oknak, ilyen például: ID, Name, Class, XPath, CSS Selectors. A tulajdonságoknak az értékeit a weboldalak HTML kódjából lehet kiolvasni. A lokátorok (locator) nem minden esetben egyediek. Az ID, a Name és a Class gyakran megegyezik több elemnél is, ezért a gyakorlatban leginkább az XPath-t és a CSS Selector-t szokták használni. Ha ugyanaz az értéke több lokátornak, akkor az első találat elve (first match) érvényesül, vagyis a kutatás az első találat után leáll. A keresés a bal-felső sarokban kezdődik és jobbra lefelé folytatódik. Az ID-t azért sem érdemes használni, mert az oldal frissítésekor megváltozhat [20].

Az 7. ábra szemlélteti egy weboldal elemeinek vizsgálatát. Egy HTML fájl alapvetően tag-ekből épül fel, mint a <html>…</html>, <head> …</head>, <body>…</body>, … . Piros kerettel emeltem ki a képen az Azonosító szövegmező vizsgálatának eredményét. Egy web elemet jobb klikkel tudunk megvizsgálni (inspect). Egy tag-en belül attribútum-érték párok vannak. Sok lokátornak az értékét innen tudjuk kiolvasni. Látható, hogy a *'name'* attribútumnak *'user'* az értéke.




**7. ábra: Web elem vizsgálat**

Vannak olyan attribútumok, melyek értéke teljesen egyedi: XPath, CSS Selector. Ezeknek az értékét több módszerrel is megkaphatjuk. Ezen eljárások azonban változhatnak a használt böngésző típusától függően. Az XPath értékét egyszerűen a keresett elemünknek a HTML kódján való jobb klikk > Copy XPath funkciójával hozzájuthatunk. A Chrome és Firefox böngészőkhöz van egy kiegészítő, ami nagyban megkönnyíti ezt a folyamatot. A 8. ábra mutatja, hogy ez a plugin (Chropath) már kigyűjtötte nekünk a kijelölt elemünkhöz a kívánt lokátortok értékét.


**8. ábra: ChroPath**

Az XPath attribútumból két féle típus létezik: Relative és Absolute. A relatív XPath az elem azonosításához tagname-attribute-value érték hármast használ. Az Absolute XPath viszont a HTML hierarchiában található helyet adja meg. Ez nagyon hasznos tud lenni, ha például egy táblázat elemein akarunk végig iterálni. Firefox böngészőhöz létezik több kiegészítő (FirePath, FireBug), ami ugyan ezt a funkciót látja el. CSS Selector attribútumhoz Chrome böngészőben nincs direkt elérés. Az alsó sávban (tool bar) kereshetjük ki az értékét.

Minden esetben érdemes leellenőrizni a lokátorok értékét, mivel néha hibásak. Ezt a konzol sávban tehetjük meg különböző formátumokban. Lehetőségünk van az XPath kinyerésre manuálisan is, ha ez szükséges. Az alábbi szintaxist követve jutunk hozzá: *//tagName[@attribute='value']*. A korábbi példából kiindulva egy konkrét példa erre: *//input[@name='user']*. A CSS Selectort hasonló módon kaphatjuk meg: *tagName[attribute='value']* szintaxissal.

### 3.2.3.4 Szinkronizálás

Szinkronizációra lehet szükség az automatizálás során, mivel különböző teljesítményű gépeken is futtathatjuk tesztjeinket. Erre jelentenek megoldást a különböző Wait utasítások. Az ImplicitWait metódus segít abban, hogy időt adjuk egy parancs végrehajtására, ha nem találja a Selenium az adott elemet, akkor csak ezután az idő után fog kivételt dobni a program. A FluentWait meghatározza a feltétel várakozásának maximális idejét, valamint a feltétel ellenőrzésének gyakoriságát. Ezenkívül a felhasználó

úgy konfigurálhatja a várakozást, hogy a várakozás során figyelmen kívül hagyja a kivételek bizonyos típusait, például a *NoSuchElementExceptions* elemet az oldalon. Az ExplicitWait lehetővé teszik, hogy a kód leállítsa a program végrehajtását, vagy befagyassza a szálat, amíg az átadott feltétel be nem következik. A feltételt egy bizonyos gyakorisággal hívják meg, amíg a várakozási idő le nem telik. Ez azt jelenti, hogy amíg az állapot hamis értéket ad vissza, addig próbálkozik és vár.

### 3.2.4 Naplózás

Szinte minden alkalmazásban használnak naplózást (logging) a fejlesztési folyamat során. A naplózás lényege, hogy a szoftver futása közben információkat írjon ki. Ez általában fájlba történő írást jelent, hogy az később visszanézhető legyen, de sokszor a konzolra írást használjuk. Több oka is van a naplózás használatának, egyrészt ezáltal ellenőrizhetővé válik a program működésének folyamata, másrészt segítséget nyújt a hibakeresésben. Egy naplóbejegyzés általában ezeket tartalmazza: pontos időpont, naplóbejegyzés szintje (info, error stb.), változók aktuális értéke, osztálynév, forrás, üzenet. Az lenti kódrészlet egy naplóbejegyzést tartalmaz.

```
2021-03-02 11:49:05.069 [main] INFO   resources.base - The driver is
initialized and the window is maximized.
```

#### 3.2.4.1 Log4j 2

Nem minden esetben lehet beépített hibakeresőt (debugger) használni (pl. több szálú és elosztott alkalmazásoknál). Erre nyújt megoldást a Log4j 2, ami egy Java-alapú naplózásra alkalmas eszköz. Különböző módokon lehet konfigurálni a működését. Az egyik egyszerű módszer erre a konfigurációs fájl használata. Különböző kiterjesztésű fájlok is alkalmasak a konfigurációhoz. Én most az XML fájl használatát szeretném kiemelni, mert ezt használtam a tesztkeretrendszer vázhoz. Fontos, hogy a konfigurációs fájlt egy forráskönyvtárban (source folder) legyen, hogy a projekt felépítésekor érvényre juthassanak a beállítások. A Log4j futásidőben konzolra és fájlba is ki tudja írni naplózási adatokat [21].

Többféle naplózási szintet különböztet meg: *trace, debug, info, warn, error, fatal, off* sorrendben. A *trace* jelöli a legapróbb történéseket. Selenium műveleteknél (pl. click, sendkey) a *debug* szintet érdemes használni, vagy olyan kisebb eseményekhez, amik segítik a hibakeresést. Az *info* szintet általában sikeres események naplózására szokás használni, mint amikor betöltött egy oldal vagy sikerült a bejelentkezés. A *warn* jelöli a

figyelmeztetéseket. *Error* esetén még tovább tud futni a program, de a *fatal* akkora hibát jelent, aminek következtében leáll a szoftver működése.

A naplózási beállításokat osztályonként külön-külön lehet változtatni igényeknek megfelelően az XML fájlban. Egyedileg lehet szabályozni a naplózás szintet és típusát. Vagyis, ha két osztályunk van, amiben szeretnénk használni a log4j naplózást, megtehetjük azt, hogy az egyikben csak konzolra íratjuk ki az *error* szint feletti eseményeket, a másik osztályban pedig fájlba és konzolra is kiíratjuk az összes szintet.

Amennyiben fájlban szeretnénk tárolni a naplózási információkat a konfigurációs fájlban meg kell adni azok tárolási helyét a projekten belül. Ezentúl szükség van a naplóbejegyzések mintájának és a fájl maximális méretének megadására is.

### 3.2.5 TestNG Riport

Egy automatizált tesztkeretrendszer hasznos funkciója lehet a riport készítés. Ennek lényege, hogy részletes információt adjon a tesztesetek futtatásáról. Segítségükkel olvashatóbb formában kapjuk meg a tesztesetek lefutásának eredményét. Sok különböző megoldást kifejlesztettek ennek megoldására.

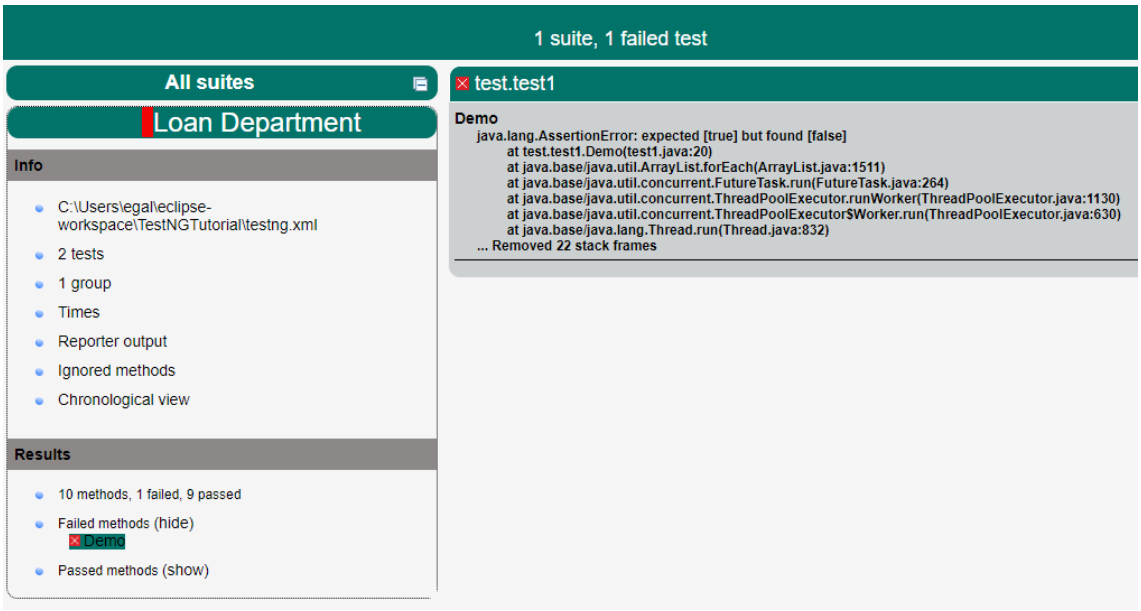
A TestNG által automatikusan generált ripotokat szeretném bemutatni. Két különböző riportot is készít nekünk a TestNG alapértelmezetten. Az egyik az *emailable-report.html* fájl, amely röviden és tömören összegzi a futás eredményét. A 9. ábra egy gyakorló projektem *emailable* riportját szemlélteti. Leírja, hogy miket futtattunk és ezek közül melyik teszt lett sikeres és melyik nem. A sikertelen teszteknél a hibaüzenetet is láthatjuk.

| Test | # Passed | # Skipped | # Retried | # Failed | Time (ms) | Included Groups | Excluded Groups |
|---|---|---|---|---|---|---|---|
| Loan Department | | | | | | | |
| Car Loan | 5 | 0 | 0 | 0 | 83 | | |
| Personal loan | 4 | 0 | 0 | 1 | 85 | | |
| Total | 9 | 0 | 0 | 1 | 168 | | |

| Class | Method | Start | Time (ms) |
|---|---|---|---|
| Loan Department | | | |
| Car Loan — passed | | | |
| test.test3 | MobileLoginCarLoan | 1609787082351 | 4 |
| | MobileSignOutCarLoan | 1609787082378 | 2 |
| | MobileSignOutCarLoan | | |
| | MobileSignOutCarLoan | | |
| | WebloginCarLoan | 1609787082391 | 2 |
| Personal loan — failed | | | |
| test.test1 | Demo | 1609787082350 | 9 |
| Personal loan — passed | | | |
| test.test2 | ploan | 1609787082368 | 3 |
| test.test4 | LoginAPIHomeLoan | 1609787082378 | 3 |
| | MobileLoginHomeLoan | 1609787082387 | 5 |
| | WebloginHomeLoan | 1609787082394 | 8 |

**9. ábra: Emailable Report**

A másik naplózási forma a TestNG részéről az *index.html* fájl. Ez részletesebb információkat ad. Ezáltal megnézhetjük a tesztjeink futásának kronológiai sorrendjét is. Az éppen használt TestNG.xml fájlunkat is láthatjuk. A tesztek futás idejét is összegzi egy táblázatban, amit akár metódus név, futási sorrend vagy futási idő szerint rendezhetünk. Ez hasznos funkció, ha arra vagyunk kíváncsiak, hogy mely tesztjeink futnak hosszú ideig. Van olyan nézet is melyen csak a sikeres vagy éppen a sikertelen tesztjeink láthatók. A 10. ábra az előzőekben bemutatott projekthez tartozó *index* riportot mutatja be.




**10. ábra: Index.html**

## 3.3 Selenium keretrendszer hasznos komponensei

### 3.3.1 Globális környezeti változók

Egy teszt keretrendszer esetében nagyon hasznos, ha a többször, különböző osztályokban is használt környezeti változókat kiszervezzük egy külön osztályba, amiknek elérést globálisan biztosítjuk. Ezt egy .poperties kiterjesztésű fájllal tehetjük. A property fájlban kulcs-érték párok szerepelnek, amik általában a böngésző típusát, webcímeket, felhasználó neveket és jelszavakat tárolnak. Ezeket az értékeket le lehet kérdezni erre készített metódusok segítségével. Property fájlokkal egy helyen tudjuk tárolni ezeket az adatokat, így könnyen karbantartható kódot készíthetünk és elkerüljük a fölösleges kódduplikációt.

### 3.3.2 Globális környezeti változók olvasása

Ahhoz, hogy a property fájlból könnyen lehessen olvasni a változók értékeit érdemes bevezetni egy külön olvasó (Reader) osztályt. Ennek az egyik előnye, hogy itt el lehet végezni egy ellenőrzést a keresett változónk létezésével kapcsolatban. Szerencsés, ha ezt nem a lényegi kódot tartalmazó osztályokban tesszük, mivel így sokkal olvashatóbb és rövidebb teszteket írhatunk és ezzel kiszervezhetjük ezt a felelősséget a SOLID elvek első törvényének (Egy felelősség elve - Single Responsibility Principle)

megfelelően. A másik nagy előnye az olvasó osztály bevezetésének, hogy egyszerűbb kifejezéssel lehet lekérni egy globális változó értékét.

### 3.3.3 Web Driver Manager

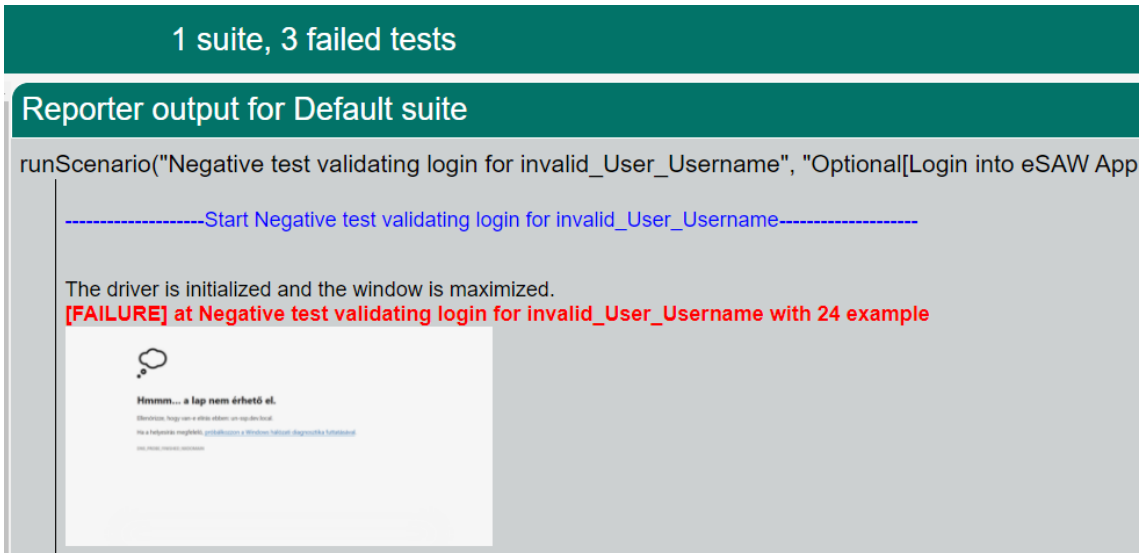
WebDriverManager egy olyan könyvtár (library), amely lehetővé teszi a Selenium WebDriver által megkövetelt illesztőprogramok kezelésének automatizálását. Használatának több előnye is van. Nem kell nekünk letölteni a különböző böngésző drivereket, hanem automatikusan megnézni a gépünkön telepített böngészők verzióját és annak megfelelő driver-t fogja használni. Magától letölti a driver-t, ha az nincs a cache-ében eltárolva. Ez csapat munkánál különösen hasznos, mivel így nem kell mindenkinek figyelni a verzió számokra és az elérési útvonalakra.

### 3.3.4 Page Object Model

A Page Object egy tervezési minta (Design Pattern), ami segíti elkerülni a kódduplikációt, ezzel növelve a karbantarthatóságot. A lényege, hogy egy weboldal tesztelendő elemeit (pl. gomb, szövegmező stb.) a Selnium által biztosított *By* metódussal lekérjük egy-egy változóba és ezeket eltároljuk. Ezekhez a változókhoz írni kell egy metódust (getter), ami visszaadja az adott elemet [22].

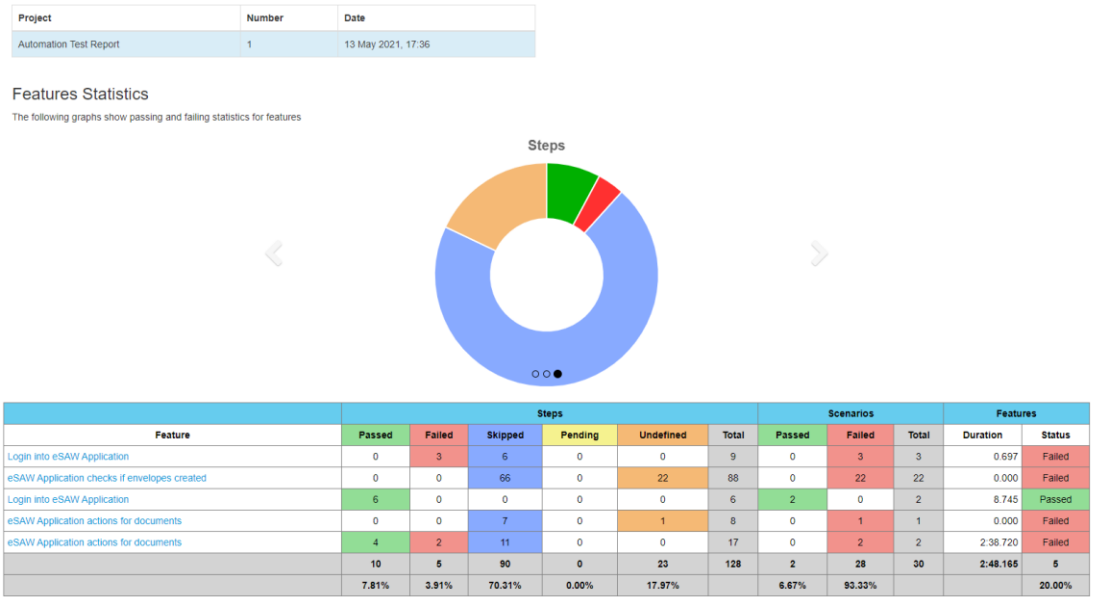
### 3.3.5 Riport

A Reporter Class a TestNG része, ami kiegészíti az általa alapvetően generált riportokat. A log4j működéséhez hasonlóan itt is egyedi információkat írhatunk a tesztesetek sorai közé, amit bele fog tenni a riportokba. Segítségével hiba esetén képernyőképet (screenshot) készíthetünk és ezt beszúrhatjuk a teszteset riportjába a hibaüzenet mellé. Ezt szemlélteti a 11. ábra.




**11. ábra: Reporter Class**

A Cucumber Reporting egy teszt futtatása során létrejött json riportból készít egy hasznos és nagyon részletes html jelentést. Az általános riport információkon kívül különböző statisztikákat készít tesztforgatókönyvekről, tesztesetekről, lépésekről, címkékről. Ezekről mind időbeli adatokat szolgáltat, mint például az átlagos és maximális futási időtartam minden lépésre. Címkék (tag) szerint csoportosított teszteseteket is vizsgálja egy táblázatban. Készít egy összesítést a sikertelen tesztek részleteiről, amiben színekkel jelöli az eredményeket. Pirossal emeli ki a hibás (fail), zölddel a sikeres (pass), kékkel az átugrott (skip), narancssárgával a nem definiált (undefined) lépéseket. A 12. ábra egy statisztikát, a 13. ábra a sikertelen teszteket mutatja.



| Project | Number | Date |
|---|---|---|
| Automation Test Report | 1 | 13 May 2021, 17:36 |

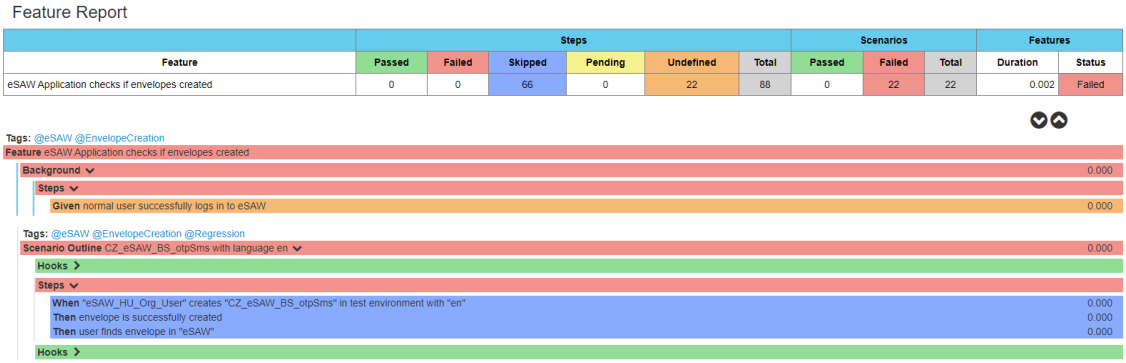
Features Statistics

The following graphs show passing and failing statistics for features

Steps

| | Steps | | | | | | Scenarios | | | Features | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Feature | Passed | Failed | Skipped | Pending | Undefined | Total | Passed | Failed | Total | Duration | Status |
| Login into eSAW Application | 0 | 3 | 6 | 0 | 0 | 9 | 0 | 3 | 3 | 0.697 | Failed |
| eSAW Application checks if envelopes created | 0 | 0 | 66 | 0 | 22 | 88 | 0 | 22 | 22 | 0.000 | Failed |
| Login into eSAW Application | 6 | 0 | 0 | 0 | 0 | 6 | 2 | 0 | 2 | 8.745 | Passed |
| eSAW Application actions for documents | 0 | 0 | 7 | 0 | 1 | 8 | 0 | 1 | 1 | 0.000 | Failed |
| eSAW Application actions for documents | 4 | 2 | 11 | 0 | 0 | 17 | 0 | 2 | 2 | 2:38.720 | Failed |
| | 10 | 5 | 90 | 0 | 23 | 128 | 2 | 28 | 30 | 2:48.165 | 5 |
| | 7.81% | 3.91% | 70.31% | 0.00% | 17.97% | | 6.67% | 93.33% | | | 20.00% |

**12. ábra: cucumber riport - statisztika**




**13. ábra: cucumber riport - sikertelen tesztek**

# 4 Megvalósítás

## 4.1 Használt környezet

Az alábbiakban röviden kifejtem, hogy milyen eszközökre volt szükségem az automatikus teszt keretrendszer váz építéséhez és a tesztek implementálásához. Ezek közül a tesztek automatikus futtatásához elegendő a JDK és a Maven telepítése. A kód központi tárolásához és verzió követéshez GitHub-ot és GitHub Desktop-ot használtam.

### 4.1.1 Eclipse

Ahogy korábban említettem Java nyelvet használtam a tesztkeretrendszer váz implementálásához és alkalmazásához. Az Eclipse fejlesztőkörnyezetet használtam a fejlesztés során, mivel ezt már ismertem és tökéletesen megfelelt a célnak. Az Eclipse segítségével egyszerű a Maven projektek kezelése és támogatja valamennyi technológia használatát, amire szükségem volt.

A tesztek futtatásához szükség van egy JDK-ra (Java Development Kit) is, amiből több verzióval is teszteltem a működést: jdk-15.0.1, jdk-15.0.2, jdk-11.0.10.

Az Eclipse használatához mindössze le kell tölteni a megfelelő oldalról [23] és néhány lépéssel telepíteni. Egyetlen fontos teendő van még ezután, Windows esetén létre kell hozni egy JAVA_HOME környezeti változót és érétkének megadni a korábban letöltött JDK elérési útját.

Végül a fejlesztőkörnyezetbe szükség volt három bővítmény telepítésére. A *Natural 0.9* kiegészítő a Cucumber, az *Eclipse m2e* bővítmény pedig a Maven használatát biztosítja, illetve a *TestNG* kiegészítő, nevéből is egyértelmű, hogy a TestNG alkalmazását támogatja.

### 4.1.2 Maven

A Maven is egy nélkülözhetetlen alkotórész a teszt keretrendszer váz felépítésénél. Ezt pusztán le kell tölteni [24] és kicsomagolni a tömörített fájlt. Használatához JDK szükséges. A JAVA_HOME környezeti változót ellenőrizni kell, hogy jól legyen beállítva a megfelelő működéshez.

### 4.1.3 Selenium

A Selenium használatához le kell tölteni [25] a számunkra megfelelő kliens drivert és kicsomagolni. Ezután a kicsomagolt JAR fájlokat importálni kell az Eclipse függőségei közé.

### 4.1.4 Docker

Első lépésként ellenőriznünk kell, hogy megfelel-e a gazdagép a Docker által előírt követelményeknek [26]. Majd a Docker hivatalos honlapjáról le kell tölteni a Docker Desktop alkalmazást. Telepítés előtt ellenőrizni kell, hogy a virtualizáció engedélyezve van-e rendszerben, ezt a feladatkezelőben tudjuk megnézni. Amennyiben nincs, a BIOS-ban lehet ezt átállítani. Ezentúl a szolgáltatások között engedélyezni kell a Hyper-V szolgáltatást. Ezek után már telepíthetjük a Docker alkalmazást. Érdemes a Docker Hub felületén regisztrálni, mert csak így tudunk nevet adni a képfájloknak, az automatikusan generált ID helyett.

## 4.2 Teszt keretrendszer váz megvalósítása

A teszt keretrendszer vázam a korábbi Architektúra fejezetben tárgyalt komponensekből épült fel és az eSignAnyWhere weboldalra való bejelentkezés folyamatát ellenőrzi. Egy rövid összefoglaló után részletesen bemutatom az elkészült osztályokat, fájlokat és azoknak működését.

### 4.2.1 Skeleton felépítésének folyamata

A szükséges környezeti beállítások elvégzése után az első lépés egy Maven projekt készítése. Ez lehetséges parancssorból és Eclipse fejlesztőkörnyezetben is. Esetemben a váz alapja egy egyszerű *'maven-archetype-quickstart*' típusú sablonból készült Maven projekt. Ez már tartalmaz egy pom.xml fájlt, amibe be kell tenni a szükséges függőségeket. Ezeket a központi Maven Repository tartalmazza, ahol könnyen rá tudunk keresni a nekünk megfelelőre. Első körben ezekre van szükség: selenium-java, webdrivermanager, testng. Ezt később folyamatosan bővíteni kell az igényeknek megfelelően.

A következő lépés a driver inicializálása, amit érdemes a WebDriverManager könyvtárral megtenni egy külön osztályba. Majd a projektet TestNG-vé kell konvertálni. Ezt a projekten való jobb klikk > TestNG > Convert to TestNG lépéssel lehet megtenni.

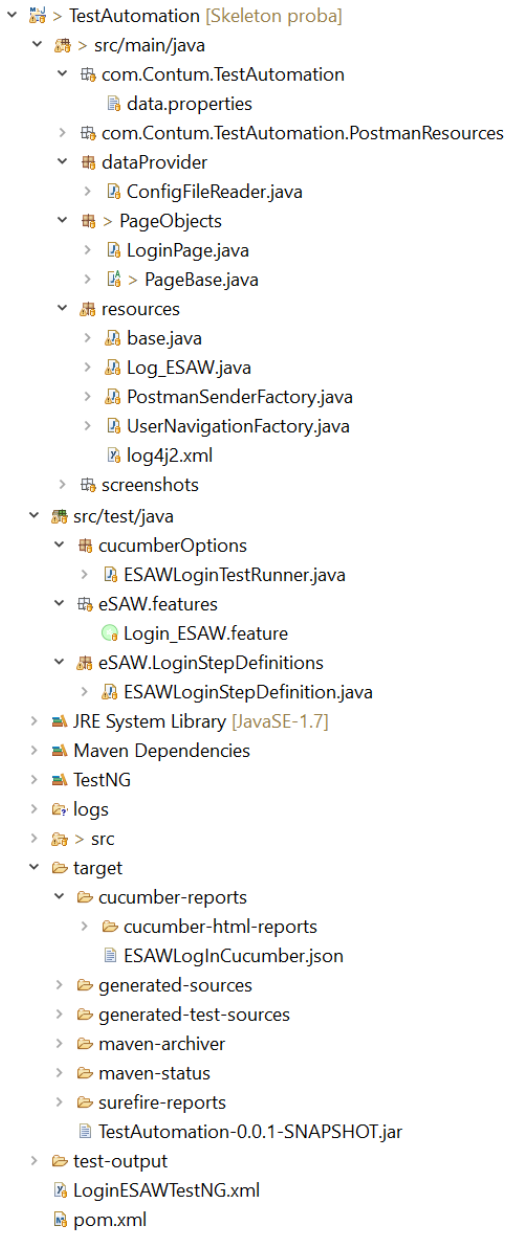
Ennek hatására létre jön egy testng.xml fájl a projekt gyökér könyvtárában, amit később használni fogunk.

Ezután következik a Cucumber-hez kapcsolódó teszt osztályok létrehozása, vagyis a feature fájl, Step Definition és Test Runner hármas, mivel ezek szorosan összefüggenek. Először a feature fájl elkészítése szükséges, mivel ez adja meg, hogy melyik tesztlépések milyen sorrendben fussanak le. Ez alapján készül el a Step Definition fájl, amiben a konkrét java kód kerül. Végül ezt a kettőt összekapcsolja a Test Runner osztály. Ahhoz, hogy végrehajtásra kerüljenek a tesztjeink ezt a TestRunner-t kell TestNG tesztként futtatni. A testng.xml fájlban lehet megadni, hogy melyik TestRunner osztályokat szeretnénk futtatni. Ezt követően érdemes ellenőrizni egy próba teszt futtatásával, hogy működik-e az eddig összeállított struktúránk. Amennyiben sikerültek ezek a lépések készen vagyunk a teszt keretrendszer váz legfontosabb funkcionális részével.

Ezeket követően a projektünk refaktorálása, optimalizálása van hátra. Készíteni kell egy property fájlt, amibe a globális környezeti változókat lehet kiemelni és egy Reader osztályt, ami ezeknek a változóknak az értékét olvassa ki. A web elemeket egy külön osztályba érdemes kiemelni és metódussal biztosítani az elérésüket. A naplózást a log4j biztosítja esetemben. Egy log4j.xml fájl létrehozása után ebben tudjuk konfigurálni az ezzel kapcsolatos beállításokat.

### 4.2.2 Skeleton szerekezete

A 14. ábra ismerteti az elkészült keretrendszer váz felépítését.




**14. ábra: Skeleton szerkezete**

A struktúrát az egyszerűség kedvéért fentről lefelé tárgyalom. A src/main/java mappában a projekt forrásállományok találhatók. A globális környezeti változókat tartalmazó fájlokat (*data.properties*) érdemes külön mappában tárolni, mivel fejlesztés során több ilyen fájlra is szükség lehet. A lenti kódrészlet egy property fájl tartalmát mutatja be.

```
browser=chrome
implicitWait=20
good_User_Username=Alice
good_User_Password=abc123
```

A property fájlból való olvasást a ConfigFileReader osztály biztosítja. Az alábbi példakód egy property fájlból olvasó metódust mutat be, ami az éppen használni kívánt böngésző típusát adja vissza. A keresett változó létezésének ellenőrzését is ez a metódus végzi.

```
public String getBrowserName() {
            String browserName = properties.getProperty("browser");
            if (browserName != null)
                  return browserName;
            else
                  throw new RuntimeException("browserName not specified in
     the Configuration.properties file.");
     }
```

A *PageObjets* package-ben található java osztályok tartalmazzák az ellenőrizni kívánt weboldalak elemeit. Ezek az osztályok a Page Object Model mintára épülnek. Alkalmazásával egy helyre kiszervezzük a web elemeket, így ez növeli a karbantarthatóságot és csökkenti a kódduplikáció mértékét, valamint nagyban segíti az olvashatóságot is. Minden oldalnak külön osztályt érdemes bevezetni, mivel így könnyen átlátható lesz a struktúra. A keretrendszer vázamban két ilyen osztály van: *LoginPage* és *HomePage*. Az alábbi példában a *LoginPage* osztályból látható egy részlet, ami három web elemet (email, jelszó, bejelentkezés gomb) tárol el és mindegyik változóhoz biztosít egy metódust, ami visszaadja az értéküket.

```
By emailField = By.cssSelector("#Email");
By passwordField = By.cssSelector("#Password");
By loginButton = By.cssSelector("#loginButton");

public WebElement loginButton() {
	return driver.findElement(loginButton);
}

public WebElement passwordField() {
	return driver.findElement(passwordField);
}

public WebElement emailField() {
	return driver.findElement(emailField);
	}
```

A driver inicializálása a *base* osztályba kerül. Itt történik a driver konfigurációja és létrehozása. Különböző beállításokat végezhetünk el egy webdriveren. A legalapvetőbb, hogy ki kell választani a böngészőnek megfelelő driver típust (ChromeDriver, GeckoDriver stb.). Be lehet még állítani a headless módú futtatást, ami annyi jelent, hogy nincs felhasználó felület a böngészőkhöz, amit láthatunk a tesztek futása során. Ez hasznos funkció, mert meggyorsítja a tesztek lefutását.

A *Log_ESAW* osztályba kerültek a kiszervezett riporter és naplózó metódusok. Ennek célja, hogy egy metódussal lehessen mindkét funkciót meghívni. Így egy helyen lehet tárolni az összes naplózással kapcsolatos metódusokat. Itt helyet kapott egy képernyőképet készítő metódus, ha hiba merülne fel tesztek futtatása során. Ezeket a képernyőképeket egy külön helyre (*screenshots mappa*) menti a rendszer. Az alábbi kódrészlet egy ilyen metódust mutat be, ami a böngésző bezárásának naplózását végzi mind a riportba, mind a logok-ba.

Ahogy korábban már részletezem, a log4j2.xml fájlban lehet konfigurálni a naplózással kapcsolatos beállításokat. A keretrendszer váz esetében konzolra és fájlba is kiírja az információkat.

```
public void browserIsClosed() {
		Reporter.log("The browser is closed. <br>");
		log.info("The browser is closed.");
		}
```

Nagyon fontos alkotórésze a keretrendszerváznak a pom.xml fájl. A Maven ez alapján építi fel a projektet. A lent látható részlet a pom.xml fájlból, ami a Cucumber függőségét írja le. Ennek köszönhetően lehet használni a Cucumbert. Több ehhez hasonló függőséget tartalmaz a pom.xml fájl.

```
<dependency>
<groupId>io.cucumber</groupId>
<artifactId>cucumber-java</artifactId>
<version>6.9.1</version>
</dependency>
```

A TestNG által biztosított funkciókat egy XML fájlban tudjuk konfigurálni. Ebben kell felsorolni a futtatni kívánt osztályok neveit és itt állíthatjuk be az egyszerre használni kívánt szálak számát is. Paramétereket adhatunk meg, amelyek a tesztek használnak futás közben. A legfontosabb a használni kívánt TestRunner fájl nevének megadása. A *thread-count* változókkal a párhuzamos futtatásnál használt szálak számát állíthatjuk be. Az alábbi kódrészlet a keretrendszer vázban található XML fájl tartalmát mutatja be.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Suite" thread-count="4" data-provider-thread-count="4">
  <test name="Login">
    <classes>
      <class name="cucumberOptions.ESAWLoginTestRunner"/>
    </classes>
  </test>
      </suite>
```

#### 4.2.2.1 Feature fájl

A tesztkeretrendszer vázam következő nélkülözhetetlen komponense a feature file, mely tartalmazza a tesztforgatókönyveket. Egy projekten belül lehet több feature fájl is, ezeket érdemes egy mappában tárolni. Az alábbi példakód a vázban található

*Login_ESAW.feature* fájlt mutatja be. A forgatókönyvek egy funkció tesztelésének lépéseit írják le. Egy feature fájlban több scenario is lehet. A Gherkin szintaxis kulcsszavakat használ egy scenario leírására. A Feature kulcsszó azonosítja a feature fájlt és az összes benne lévő tesztforgatókönyvet. A *Feature* kulcsszó elé, az első sorba lehet tenni egy címkét (tag), ami összefogja az összes forgatókönyvet, ami ebben a fájlban van. Ebben a példakódban ez a *@Example*. A Test Runner fájlban lehet megadni, hogy milyen címkével ellátott teszteket szeretnénk futtatni. Címkéket az összes tesztesethez lehet tenni (pl. *@Login*), így könnyen kiválaszthatjuk pontosan, hogy melyikeket szeretnénk futtatni. A *Scenario Outline* kulcsszó egyedileg azonosítja a tesztforgatókönyveket.

```
@Example
Feature: Example tests

Background:
	Given Browser is initialized

@Login
Scenario Outline: Positive test validating login
	And Navigate to "<loginUrl>" Site
	When User enters "<username>" and "<password>" and logs in
	Then Verify that "<username>" is succesfully logged in to "<url>"
	And close browsers

Examples:
    |loginUrl    |username             |password             |url           |
    |eSAW_Login  |good_User_Username   |good_User_Password   |eSAW_Inbox_Url|
    |eSAW_Login  |good_User_Username_2 |good_User_Password_2 |eSAW_Inbox_Url|

@FailedLogin
Scenario Outline: Negative test validating login
	Given Navigate to "<loginUrl>" Site
	When User enters "<username>" and "<password>" and logs in
	Then Verify that "<username>" cannot log in and notification displayed
	And close browsers

Examples:
	|loginUrl    |username                 |password                 |
	|eSAW_Login  |invalid_User_Username    |invalid_User_Password    |
```

Ebben a példában két *Scenario Outline* van egy feature fájlban: *@Login*, *@FailedLogin*. Egy sima *Scenario*-tól ez annyiban különbözik, hogy ebben változókat adhatunk meg értékekkel, amiket az *Examples* kulcsszóval jelölt táblázatban tárolunk. Ennek köszönhetően újrafelhasználható teszteseket írhatunk. Egy ilyen táblázatnak az első sora a fejléc, ami változók nevét tartalmazza, a többi sorban ezeknek a változóknak

az értéke van, amik a tesztek futtatásához használatosak. Többféle módon adhatunk meg változó értékeket. Az egyik módszer az, hogy a konkrét értékét adjuk meg a változónak (pl. felhasználó névnek beégetünk egy string-et: ”Alice”). Egy másik lehetőség, hogy globális változókat adunk meg, amiket egy külön fájlban tárolunk (pl. *good_User_Name*). A példában szereplő *Login* címkével ellátott tesztforgatókönyv (*Positive test validating login*) kétszer fogja lefuttatni a Cucumber. Először a *good_User_Name,* majd utána a *good_User_Name_2* felhasználónévvel.

A *Given, Then, When* és *And* kulcsszavak különböző lépéseket jelölnek, amiket a Step Definition fájlban deklarálunk. Ott adjuk meg például Java nyelven, hogy mi történjen ezekben a lépésekben pontosan. Ezeket a lépéseket a tesztesetben szereplő sorrendben futtatja a Cucumber fentről lefelé. A *Given* kulcsszóval feltételhez köthetjük a teszt lefutását. A példában ez azt jelenti, hogy először el kell navigálnunk a megfelelő oldalra, hogy utána tudjuk tesztelni a bejelentkezést. *When* kulcsszóval egy eseményt szoktak leírni. A példában ez az, amikor a felhasználó beírja a felhasználónevét és a jelszavát, majd megpróbál bejelentkezni. A *Then* írja le az esemény várható eredményét. Ebben a lépésben lehet összehasonlítani az elvárt és a valós eredményt. A példában itt történik a bejelentkezés érvényességének ellenőrzése. Az *And* kiegészíti az előző lépést. Tehát, ha az előző lépés *Then* kulcsszóval kezdődött, akkor az aktuális sorunkat is úgy kell értelmezni. Ez a könnyebb olvashatóságot és érthetőséget segíti elő.

Amennyiben egy feature fájlban az összes teszteset első *Given* lépése megegyezik, akkor ezt kiemelhetjük a *Background* kulcsszóval. A példában ez a driver inicializálása. Mielőtt elnavigálhatnánk akármelyik oldalra a böngészőben, először létre kell hozni a megfelelően konfigurált drivert [27].

#### 4.2.2.2 Step Definition fájl

A tesztkeretrendszer váz következő fontos alkotórésze az úgynevezett Step Definition fájl. A feature fájlban megírt teszteset lépéseit itt implementáljuk. Ez a fájl mondja meg pontosan, hogy mi fog történni az egyes lépések végrehajtásakor. Amikor a Cucumber megpróbál végrehajtani egy feature fájlban lévő lépést, akkor keres hozzá egy azonos annotációval ellátott metódust a Step Definition fájlban. Emiatt fontos az egyedi és pontos elnevezése a metódusoknak. Ebben segít a Tidy Gherkin nevű Chrome kiegészítő. Bementként oda kell neki adni a feature fájlunk tartalmát és visszaadja a hozzá

tartozó ajánlott Step Defintion fájl tartalmát. Segítségével elkerülhető a metódus nevek elírása és a felesleges gépelés is. Az alábbi kódrészletben egy feature fájlban lévő lépéshez tartozó java metódus látható, ami driver bezárásáért felelős.

```
@And("^close browsers$")
public void close_browsers() throws Throwable {
	driver.quit();
	logger.browserIsClosed();
}
```

Minden lépésnek különböző eredménye lehet. A *Success* azt mutatja, hogy a feature fájlban lévő lépéshez a Cucumber megtalálta a megfelelő nevű metódust és sikeresen le tudta futtatni. Az *Undefined* azt jelenti, hogy az adott lépéshez nem talált megfelelő metódust a Step Definition fájlban. Amennyiben hibát kapunk vagy nem lehet futtatni a tesztet *Fail* eredmény születik. Egy elbukott teszt után következő és azzal összefüggő teszteket a Cucumber nem fogja végrehajtani, ennek eredménye *Skipped*.

Amennyiben a feautre fájlban lévő minden tesztesetben benne van ugyan az a lépés, ezt kiszervezhetjük a hozzátartozó Step Definition fájlba egy úgynevezett Hook segítségével. Tulajdonképpen ugyan az funkciója, mint a *Background* kulcsszónak. Érdemes átgondolni, hogy melyiket használjuk. Amennyiben Hook-ot használunk, az nem fog látszódni a feature fájlban és ez zavaró lehet annak, aki nem ismeri mélyen a forráskódot. Több típusú Hook is létezik. Vannak olyanok, amelyek a tesztesetek előtt vagy után futnak le. Ezeket általában a driver inicializálására és a környezet konfigurálására szokták használni, illetve azok bezárására. Létezik olyan Hook is, ami minden lépés előtt vagy után fut le. Mindegyik Hook-hoz beállíthatunk egy prioritást amennyiben több ugyanolyan típusú Hook-ot használunk, ezáltal meg tudjuk szabni a végrehajtásuk sorrendjét.

#### 4.2.2.3 Test Runner

Ez a fájl típus felelős egy feature fájl és egy Step Definition fájl összekötéséért. A Test Runner egyetlen egy osztályból áll, amit el kell látnunk egy bizonyos *@CucumberOptions* annotációval ahhoz, hogy TestNG tesztként lehessen futtatni. Ez a fájl felelős azért, hogy ténylegesen végrehajtásra kerüljenek a kívánt tesztek. Az alábbi példakódban láthatunk egy nagyon egyszerű beállításokat tartalmazó Test Runner fájlt. A *features* opcióval adhatjuk meg a használni kívánt feature fájl pontos helyét. A glue beállításánál kell megadni a Step Defintion fájl helyét. Lehetőség van a tesztek közül

bizonyos címkékkel ellátott tesztek csoportját futtatni. Ezt a *tags* opció biztosítja, de nem kötelező megadni. Érdemes még beállítani a *monochrome=true* értéket is, mivel ezzel olvashatóbban írja ki a konzolra az adatokat. A *@dataprovider* annotációval és a hozzátartozó metódussal lehet elérni, hogy a tesztek párhuzamosan fussanak le.

```
@CucumberOptions(
          monochrome = true,
          plugin = {"pretty", "json:target/cucumber-
          reports/ESAWLogInCucumber.json"},
          features = {"src/test/java/eSAW/features/Login_ESAW.feature"},
          tags = "@Login",
          glue = {"eSAW.LoginStepDefinitions"}
          )
public class ESAWLoginTestRunner extends AbstractTestNGCucumberTests  {


      @Override
      @DataProvider(parallel = true) public Object[][] scenarios() {
            return super.scenarios(); }

}
```

#### 4.2.2.4 Cucumber Reporting

Gyakorlatban a Cucumber Riporting használata igen egyszerű. Két dologra van szükség. Először a TestRunner fájlba található *@CucumberOptions* beállításokhoz be kell tenni egy új elemet:

```
plugin = {"pretty", "json:target/cucumber-reports/ESAWLogInCucumber.json"}
```

Ez a sor felelős egy json formátumú riport készítéséért, amiből később elkészül a sokkal jobban olvasható html jelentés. Itt meg kell adni, hogy hova kerüljön mentésre ez a fájl a projekten belül, illetve nevet is kell adni neki.

A másik fontos teendő a pom.xml kiegészítése. Az alábbi képen látható, hogy mit kell beilleszteni ide. Fontos, hogy ezt nem a dependencies részhez kell tenni, hanem a plugins blokkhoz. Néhány konfigurációt el kell itt végezni. Meg kell adni egy kimeneti mappát, ahova a html riport kerül és egy bemeneti mappát, amiben a json riport van, hogy ki tudja olvasni az adatokat. A *'**/*Cucumber.json'* kifejezéssel biztosíthatjuk, hogy az összes projektünkben megtalálható Cucumber.json nevű fájlt átalakítsa html jelentéssé.

```
<plugin>
    <!-- https://mvnrepository.com/artifact/net.masterthought/cucumber-repo
    <groupId>net.masterthought</groupId>
    <artifactId>maven-cucumber-reporting</artifactId>
    <version>5.5.0</version>
    <executions>
        <execution>
            <id>execution</id>
            <phase>verify</phase>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <projectName>Automation Test Report</projectName>
                <outputDirectory>target/cucumber-reports</outputDirectory>
                <cucumberOutput>target/cucumber-reports</cucumberOutput>
                <inputDirectory>target</inputDirectory>
                <skippedFails>true</skippedFails>
                <jsonFiles>
                    <param>**/*Cucumber.json</param>
                </jsonFiles>
            </configuration>
        </execution>
    </executions>
</plugin>
```

**15. ábra: Cucumber riport függőség és konfiguráció**

### 4.2.3 Tesztek futtatása

Az elkészült tesztek ezek után készen állnak a futtatásra. Ez parancssorból és fejlesztőkörnyezetből is ugyan úgy végrehajtható. Eclpise használatával ezt úgy tehetjük meg, hogy TestNG.xml fájlon jobb klikkelés után kikeressük a ’Run as TestNG suit’ opciót. Ha lefutott a teszt, a projekten jobb klikkelve megtalálhatjuk a Maven install funkciót. Végül frissíteni kell a projektet és az elkészült riportok is elkészültek.

Parancssorból a következő utasításokkal érhetjük el ugyan ezt az eredményt:

```
mvn clean
mvn test -Dsurefire.suiteXmlFiles=LoginESAWTestNg.xml
mvn install
```

A *mvn clean* paranccsal kitörölhetjük a *target* mappánk tartalmát. A *mvn test* parancs paraméterben megadhatjuk, hogy melyik TestNG.xml fájlt szeretnénk futtatni.

A tesztek lefutása után mindig kapunk egy összesítést a tesztek sikerességével kapcsolatban. Erre mutat példát az alábbi kódrészlet.

```
================================================
Suite
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0
================================================
```

### 4.2.4 Docker

A teszt futtatásához készítettem két konténert. Mindkettő Ubuntu alapú, de az egyiken Chrome böngésző van és az ahhoz szükséges környezet, a másikon pedig Firefox. Az elkészítés folyamatát a Firefox-os konténeren fogom bemutatni, amit parancssorban végeztem el.

Létrehoztam egy kiterjesztés nélküli fájlt ’Dockerfile’ névvel. Fontos a pontos elnevezés, mert a rendszer mindig ezt a nevet fogja keresni az építési folyamathoz. Parancssorban el kell navigálni abba a mappába, ami tartalmazza ezt a fájlt. A *’Create Dockerfile’* parancs hatására elkészül a tényleges Dockerfile, amiből majd felépül a képfájl.

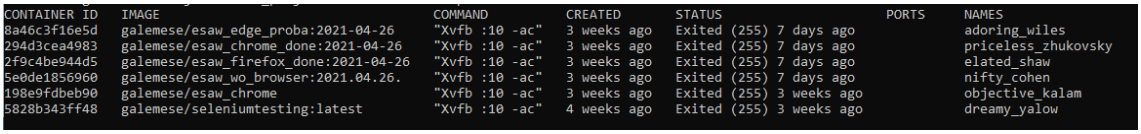
Egy Dockerfile három részre osztható. Az elsőben adhatjuk meg, hogy melyik létező fájlrendszert (base image) szeretnénk használni. Ez lehet ténylegesen csak egy fájlrendszer (pl. alpine) vagy akár egy teljes operációs rendszer is. A második részben olyan parancsokat adhatunk meg, amik még a konténer elindítása előtt lefutnak. A harmadik részben pedig olyan utasításokat definiálhatunk, amik a konténer elindítása után fognak lefutni. Az alábbi képen az általam használt Docker fájl látható.

```
FROM ubuntu
RUN apt-get update
RUN apt -y install default-jre
RUN apt -y install maven
RUN apt -y install firefox xvfb
RUN apt -y install git
ENV DISPLAY :10
```

Az első sor azt jelzi, hogy egy ubuntu rendszerből indultam ki. Szükség van a csomagok frissítésére, ez látható a második sorban. Majd ezután következő sorokban a teszt futtatáshoz szükséges eszközök letöltése látható. Az xvfb (X virtual framebuffer) a grafikus alkalmazások headless módban való futtatásához szükséges.

A következő lépés a képfájl felépítése amit a *'Docker build -t <image_name> .'* paranccsal tettem meg. Az <*image_name*> helyére általában ezt az elnevezési konvenciót szokták használni: *'username/projectname:version'*. Ennek elkészülését a *'docker images'* paranccsal ellenőrizhetjük.

Ezután kell elkészíteni a konténert: *'docker run -ti -d <image_name> Xvfb :10 -ac &'*. Ennek ellenőrzésére a *'docker ps -a'* parancsot lehet használni, ami a létező konténerek listáját adja vissza. Erre mutat példát a 16. ábra.



| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|---|---|---|---|---|---|---|
| 8a46c3f16e5d | galemese/esaw_edge_proba:2021-04-26 | "Xvfb :10 -ac" | 3 weeks ago | Exited (255) 7 days ago | | adoring_wiles |
| 294d3cea4983 | galemese/esaw_chrome_done:2021-04-26 | "Xvfb :10 -ac" | 3 weeks ago | Exited (255) 7 days ago | | priceless_zhukovsky |
| 2f9c4be944d5 | galemese/esaw_firefox_done:2021-04-26 | "Xvfb :10 -ac" | 3 weeks ago | Exited (255) 7 days ago | | elated_shaw |
| 5e0de1856960 | galemese/esaw_wo_browser:2021.04.26. | "Xvfb :10 -ac" | 3 weeks ago | Exited (255) 7 days ago | | nifty_cohen |
| 198e9fdbeb90 | galemese/esaw_chrome | "Xvfb :10 -ac" | 3 weeks ago | Exited (255) 3 weeks ago | | objective_kalam |
| 5828b343ff48 | galemese/seleniumtesting:latest | "Xvfb :10 -ac" | 4 weeks ago | Exited (255) 3 weeks ago | | dreamy_yalow |

**16. ábra: Docker container lista**

A *'docker exec -ti <container_id> bas*h' paranccsal léphetünk be a konténer parancssorába és ennek köszönhetően vezérelhetjük a konténerünket (pl. futtathatunk programokat). Ezután még szükségem volt a keretrendszer váz kódjának klónozására (git clone). Ezek elvégzése után már futtathattam a teszteket a konténerben.

## 4.3 Skeleton alkalmazása egy új alkalmazásra

A tesztkeretrendszer vázból kiindulva építettünk fel egy új alkalmazáshoz tartozó automatizált tesztelési keretrendszert egy kis csapattal a Contum Kft-nél. A váz jó kiindulási alapnak bizonyult a gyakorlatban is. Mivel már minden csapattag értett a Selenium használatához gyorsabban ment tesztek elkészítése. Mivel egy nagyobb projektről volt szó a legfontosabb funkciók ellenőrzéséhez szükséges tesztek néhány hét (4*16 munkanap) alatt készültek el. A teszt keretrendszer optimalizálása még néhány hetet vett igénybe, mivel több számunkra új funkció bevezetését szerettük volna megoldani, hogy gyorsabb legyen a tesztek futási ideje és könnyebb legyen a hibakeresés. Egy elkészült teszt keretrendszer karbantartása lényegesen kevesebb időt vesz igénybe. Ezt a tudást a következő projektben már sokkal gyorsabban lehet majd alkalmazni. Ennek a projektnek a készítése során szerzett fontosabb és érdekesebb gyakorlati tapasztalatokat szeretném részletezni ebben az alfejezetben.

### 4.3.1 Csapat munka

Csapatmunka esetén lényeges, hogy előre egyeztessék a résztvevők az elnevezési konvenciókat az egységesség miatt. A fájlok elnevezését ugyan úgy érdemes szabályozni, mint a változókét és metódusokét. Ennek hiányában sok munkaórába telt mire ezt kijavítottuk. Ha csapatban dolgozunk kiemelten figyelni kell a feladatok kiosztásánál arra, hogy egymástól független részeken dolgozzunk egyidőben. Esetünkben néha előfordult, hogy összefüggő vagy egymásra épülő funkciókon dolgoztunk egyszerre. Ezesetben, ha az alapfunkció logikája, szerkezete megváltozott és közben a ráépülő részen dolgoztunk, azt újra kellet írni, mert konfliktusok születtek GitHub-on az ágak (branch) összevonásánál (merge). Ennek elkerülésére szolgálhat a Scrum módszertan. A napi pár perces megbeszélések reggelente segítenek abban, hogy megismerjük a csapattagok feladatait nagy vonalakban ezzel csökkentve a konfliktusok kialakulását.

Amennyiben egy nagyobb projekthez készítünk teszt keretrendszert fontos előre megtervezni, hogy az alprojekteket vagy modulokat hogyan szervezzük. Általában érdemes a különböző funkciójú fájlokat külön package-be szervezni modulonként. Ez a gyakorlatban annyit jelent, hogy például a feature fájlokat és Step Definition fájlokat külön mappába készítjük.

### 4.3.2 Feature fájl

A tapasztalatok alapján nem érdemes minden tesztforgatókönyvet egy feature fájlba tenni. Egyrészt kevésbé lesz átlátható, másrészt ez nagyon lelassítja a fájl szerkesztését. Egy 150 soros 10 Scenario Outline-t tartalmazó feature fájlban, ha át akarunk írni valamit, az legalább 1-2 percbe telik. Ez laikus szemmel nem tűnhet olyan soknak, de ha ezt akár csak 10 percenként kell megtenni az időnk 10%-át erre pazaroljuk, ezenkívül megakasztja a gondolatmenetet és a tesztkészítő nyugalmi állapotára sincs jó hatással. Gyakorlatilag 4-5 forgatókönyvnél vagy 60-70 sornál nem érdemesebb nagyobb feature fájlokat készíteni.

### 4.3.3 Step Defintion

A Step Defintion fájlok esetében sem érdemes az összes metódust egy nagy fájlban tárolni. Egy 1000 soros fájlban már elég nehéz megtalálni a keresett metódust. Viszont az sem jó, ha nagyon feldaraboljuk ezeket sok kis fájlra. Amennyiben mégis egy nagy Step Definition fájl használatát választjuk, a hosszú és bonyolult metódusokat

mindenképpen érdemes kiszervezni egy külön osztályba. A Step Defintion fájlok méretét csökkenthetjük, ha kiszervezzük a naplózáshoz és riportoláshoz használt metódusokat egy külön osztályba. Ez azért is előnyös, mert ezzel kiszervezünk egy felelősséget a Step Definition-ből.

### 4.3.4 Globális változók

A globális változók kiszervezése külön osztályba nagyon jó megoldásnak bizonyult. A gyakorlatban egyetlen egy globális változó létezésének ellenőrzését és hibakezelését kb. 5 sorból lehet megoldani. Tapasztalataim alapján egy 150 soros Property fájlhoz egy 750 soros olvasó osztály tartozik. Tehát ilyen sok fölösleges kód kerülne a teszt osztályunkba.

### 4.3.5 Page Object Model

A Page Object Model is igazolta létjogosultságát. Használata nélkül így néz ki, ha beírunk egy felhasználónevet és egy jelszót, majd megnyomjuk a bejelentkezés gombot:

```
driver.findElement(By.xpath("//tr[2]/td[1]/table[1]/tbody[1]/tr[1]/td[1]")
     .sendKeys(dataProperties(username));
driver.findElement(By.xpath("//input[@id='pwd']")
     .sendKeys(dataProperties(password));
 driver.findElement(By.xpath("//input[@id='btnSubmit']").click();
```

Page Object használatával, pedig így néz ki ugyan ez a rész:

```
     page.emailField().sendKeys(dataProperties(username));
     page.passwordField().sendKeys(dataProperties(password));
     page.loginButton().click();
```

Mindenképpen megéri tehát a Page Object-ek kigyűjtése. Igaz, hogy jó pár órába telik, mire minden tesztelendő elemet kigyűjtünk, de ez a ráfordított munka nagyon hamar megtérül, mivel javítja a kód karbantarthatóságát. Ha egy elem elérési útja megváltozik az oldalon, ezt csak egy helyen kell javítanunk. Ezen túl érdemes megfontolni egy ős *page* osztály bevezetését, ha a különböző tesztelendő oldalakon (Login page, Home page stb.) van több megegyező elem. Ezzel tovább javíthatjuk a kódunk karbantarthatóságát.

### 4.3.6 Naplózás

Amennyiben lehetőségünk van rá mindenképpen érdemes már a tesztek implementálása közben megírni a naplózó, illetve riportért felelős részeket. Utólag ezt megírni körülményes és hosszadalmas. Ha éppen valaki más kódjába kell a naplózást bevezetni előfordulhat, hogy nem értjük teljesen vagy sokáig tart mire megértjük.

# 5 Értékelés

## 5.1 Mérések

A váz alkalmazásának folyamata közben méréseket végeztem a futás idővel kapcsolatban. Ekkor már minden funkció működött a tesztek elvégzéséhez, de még nem volt optimalizálva és refaktorálva a teszt keretrendszer. A legfontosabb tényező, hogy a párhuzamos futtatás és a headless mód még nem volt bevezetve. A projekt összesen 180 testesetet tartalmazott. Összesen 6 mérést végeztem, melyek eredményét a 1. táblázat mutatja. Az átlagos futás idő 30,23 perc. Az átlagosan előforduló hibák száma 0,5.

| futási idő (perc) | 32,65 | 29,43 | 26,89 | 30,01 | 31,14 | 31,28 |
|---|---|---|---|---|---|---|
| hibák száma | 0 | 1 | 0 | 0 | 2 | 0 |

**1. táblázat: váz alkalmazásának folyamata alatti adatok**

A projekt refaktorálása után ismét megvizsgáltam a szükséges futási időt ugyanazon funkciók tesztelésekor. Átlagosan ugyan azok az eredmények jöttek ki, mint refaktorálás előtt, vagyis az osztályok és metódusok kiszervezése nem csökkentette számottevően a teljes futási idő hosszát.

Ezután a paralel futtatás bevezetését követően is lemértem ugyanezekre a funkciókra (180 teszteset) a futási időt. A 2. táblázat**Error! Reference source not found.** foglalja össze az átlagolt eredményeket. Itt láthatjuk, hogy a szálak számával a hibák száma is növekszik. A 20 szálas futtatást már nem bírták el a hardverek, de már 10 szálnál is nagyon megugrott a hibák előfordulási aránya. Ennél a tesztnél tehát érdemes a 3, 4 vagy 5 szálas végrehajtást választani, mert ezek a leggyorsabbak és legmegbízhatóbbak.

| futási idő (perc) | szálak száma | összes teszteset | hibák száma |
|---|---|---|---|
| 30,11 | 1 | 180 | 0,5 |
| 21,34 | 2 | 180 | 1 |
| 16,50 | 3 | 180 | 1,25 |
| 13,38 | 4 | 180 | 3,25 |
| 13,04 | 5 | 180 | 2 |
| 14,51 | 10 | 180 | 17,75 |
| - | 20 | 180 | - |

**2. táblázat: váz alkalmazása - párhuzamos futtatás**

Ezentúl megvizsgáltam a vázban található bejelentkezési folyamat átlagos futási idejét is. Ebben 3 teszteset található. A headless mód és a párhuzamos szálak időeredményekre tett hatását vizsgáltam. Ezek eredményét a 3. táblázat foglalja össze, a részletes időeredmények a Függelék fejezetben találhatóak. Mivel ezek a tesztek gyorsan lefutnak milliszekundumban mértem az időt.

Az adatokból biztosan megállapíthatjuk, hogy a felhasználói felület nélküli futtatás lerövidíti a végrehajtási időt. Mivel ebben az esetben csupán három teszteset futtatásáról van szó, így a szálak számának növelése is biztosan növeli a gyorsaságot.

| headless | szálak száma | átlagos futási idő(ms) | átlagos hibák száma |
|---|---|---|---|
| false | 3 | 11863,1666 | 0 |
| true | 3 | 7758,6666 | 0 |
| false | 2 | 13221,3300 | 0 |
| true | 2 | 9462,1670 | 0 |
| false | 1 | 17029,6666 | 0 |
| true | 1 | 12227,8333 | 0 |

**3. táblázat: bejelentkezés átlagos adatok**

## 5.2 Tanulságok

Röviden szeretném összefoglalni a keretrendszer váz készítése és alkalmazása során szerzett fontosabb tanulságokat és tapasztalatokat.

Csapat munka esetén a fejlesztési folyamat elején fontos tisztázni az elnevezési konvenciókat és egyéb formai szabályokat, mivel ezzel minimalizálhatjuk az ilyen típusú hibák javítására fordított munkaórák számát és növeli a felépítés átláthatóságát. A Srum módszer használata elősegíti a csoport tagjai között a kommunikációt. A reggeli pár perces daily standup-ok biztosítják, hogy mindenki tisztában legyen nagy vonalakban a többiek feladataival. Figyelni kell a feladatok felosztásánál arra, hogy párhuzamosan ne dolgozzanak többen ugyan azon a részen, mert ez nagy *merge* konfliktust okozhat.

A váz alkalmazása során a tesztelendő alkalmazás még fejlesztés alatt állt, emiatt sokszor nem volt elérhető a tesztfuttatáshoz szükséges céges környezet. Ennek következtében sokszor kellett várnunk az újonnan implementált tesztek ellenőrzésével, ami megnövelte a teljes teszt keretrendszer elkészítéséhez szükséges időt. Ezt esetleg azzal lehet mérsékelni, ha a fejlesztők előre megadnak bizonyos időkereteket, amikor biztosan elérhető lesz a céges környezet.

A Selenium és ehhez kapcsolódó technológiák használatát néhány hét alatt el lehet sajátítani mélyrehatóan. Ezt követően az első nagy projekthez tartozó teszt keretrendszer kiépítése még lassabban megy. Ezt az időt lecsökkenthetjük, ha készítünk egy általános tesztkeretrendszer vázat, amit a későbbi projekteknél felhasználhatunk. Természetesen, ahogy egyre több tapasztalatot szerzünk az automatikus tesztelés terén, egyre gyorsabban és egyre több hasznos funkció beépítésével lehetséges egy teszt keretrendszer felépítése. Egy elkészült teszt keretrendszer karbantartása ehhez képest lényegesen kevesebb erőforrást igényel. Ebben hosszú távú tapasztalatom nincs, de rövid távon heti néhány órát/napot vesz igénybe maximum. A manuális teszteléshez viszonyítva sok esetben ez megtérülő befektetésnek bizonyul. Kooperatív képzésem során elvégeztem egy webalkalmazás elfogadási/átvételi tesztjét (User Acceptance Test) manuálisan. Az összes funkció átvizsgálása 1 napot vett igénybe dokumentálással együtt. Ennek során sokkal kevesebb funkciót kellett ellenőrizni, mint a keretrendszer váz alkalmazása során futtatott tesztek esetén, ami mindössze 15 percig tart. Tehát hatalmas különbség van a tesztek elvégzéshez szükséges időben a manuális és automatikus tesztelés esetén.

Párhuzamos futtatással és felhasználói felület nélkül meggyorsíthatjuk a tesztek lefutását, esetünkben ez megfelezte a futási időt. Riportok segítségével könnyen kiszűrhetjük, hogy mely tesztek tartanak a legtovább és ezeket optimalizálhatjuk. Docker alkalmazásával ellenőrizhetjük, hogy működnek a tesztek különböző környezetekben.

A váz alkalmazása során több hibát is felfedeztünk, amit a tesztelő-fejlesztő csapat nem vett észre. Ezek közül szeretnék részletezni egyet. Egy elektronikus aláírási folyamat elindításának része egy dokumentum feltöltése, amit a felhasználó később aláírhat. Ezt a dokumentumot kódolva továbbítja a Contum-os folyamat vezérlő az eSAW-nak. Ennek hiányában nem lesz mit aláírnia a felhasználónak. A dokumentum feltöltése során a rendszernek ellenőriznie kellett volna, hogy üres-e a dokumentumot tartalmazó tag, de ez nem történt meg.

Továbbfejlesztési lehetőségként több eszköz is rendelkezésünkre áll. Érdemes lenne a sikertelen tesztek automatikus újra futtatását bevezetni, amivel kiszűrhetjük a fals negatív teszteket (pl. váratlan szerver oldali hiba). Ezt például a TestNG IRetryAnalyzer interfész implementálásával lehet megtenni [28]. Ezentúl lehetőségünk van automatikusan indított időzített teszt futtatás beállítására is a Jenkins vagy a GitLab CI/CD segítségével. A Slenium Grid segítségével virtuális vagy igazi távoli gépeken

hajthatjuk végre a Web Driver parancsokat. Könnyedén tesztelhetünk párhuzamosan, különböző böngészőkben.

# 6 Összefoglaló

Ebben a fejezetben szeretném összefoglalni az elvégzett munkámat és az azzal kapcsolatos tapasztalataimat.

Dolgozatomban először megvizsgáltam a távoli aláírásos és távoli azonosítási technológiákat, melyek a szakmai gyakorlatomat biztosító cég profiljához kapcsolódnak. Az utóbbi években egyre jobban elterjedt ezen technológiák használata. Napjainkban már pár perc alatt nyithatunk bankszámlát távolról. A magyar közigazgatásba is bevezették az arcképes azonosítást az online ügyintézés során. Ilyen megoldások segítségével sok erőforrást megspórolhatunk.

Ezután megvizsgáltam egy automatizált teszt keretrendszer általános felépítését és létrehoztam egy tesztkeretrendszer vázat. Ez olyan alapot biztosít, amelyből kiindulva könnyebben felépíthető más projektek teszt keretrendszere is. A váz alkalmazás során tanultakat összefoglaltam és végül mérésekkel vizsgáltam a párhuzamos szálak számának és a headless módú futtatás hatásait. Ezen tényezők gondos megválasztása lényegesen felgyorsítják a tesztek lefutásának idejét és a megbízhatóságot sem csökkentették.

A teljes folyamat során sokat tanultam a Selenium használatáról és az automatikus tesztelésről. Azelején kicsit tartottam ettől a témától, mert nem voltak mély ismereteim ezzel kapcsolatban, de időközben nagyon megkedveltem és rácsodálkoztam, hogy mennyi mindent lehet megoldani automatizált tesztek segítségével. A távoli aláírásos technológiák részletes működésének megismerése is érdekes volt, jó tudni mi történik a háttérben. Megtapasztalhattam a csapat munka előnyeit is a szakmai gyakorlatom során és az önálló probléma megoldó képességemet is fejleszthettem. Összességében örülök, hogy ezt a témát választottam.

# 7 Irodalomjegyzék


[1] *Európai Parlament és Tanács 910/2014/EU rendelete,* 2014.

[2] Európai Parlament és Tanács, „EUR-Lex," [Online]. Available: https://eur-lex.europa.eu/legal-content/HU/LSU/?uri=celex:32014R0910. [Hozzáférés dátuma: 2. május 2021].

[3] *2015. évi CCXXII. törvény az elektronikus ügyintézés és a bizalmi szolgáltatások általános szabályairól,* 2015.

[4] I. Z. Berta, „BERTA - Electronic Signature," [Online]. Available: https://berta.hu/files/Electronic_Signature.pdf. [Hozzáférés dátuma: 3. május 2021].

[5] Z. Hornák, E. Jeges, D. Jancsik, L. Máté, M. Dr Nehéz-Pozsony, G. Tóth, I. Vincze, Á. Kovács, C. Körmöczi és T. Persa, „Távoli személyazonosítási technikák," BME - MIT és Guardware kft, 2005.

[6] F. Hlács, „hwsw," 30. december 2014. [Online]. Available: https://www.hwsw.hu/hirek/53369/chaos-computer-club-ujjlenyomat-biztonsag-hacker.html. [Hozzáférés dátuma: 3. május 2021].

[7] *2017. évi LIII. törvény a pénzmosás és a terrorizmus finanszírozása megelőzéséről és megakadályozásáról,* 2017.

[8] „Namirial," [Online]. Available: https://www.xyzmo.com/. [Hozzáférés dátuma: 20 május 2021].

[9] BME AUT, „Szoftvertechnológia és -technikák 10.előadás - A szoftverfejlesztés fázisai, dokumentáció, tesztelés alapjai," 2020/21 - őszi félév.

[10] L. Ficsor, L. Kovács, G. Kusper és Z. Krizsán, Szoftvertesztelés.

[11] „ToolsQA," [Online]. Available: https://www.toolsqa.com/software-testing/what-does-end-to-end-test-mean/. [Hozzáférés dátuma: 4. május 2021].

[12] „Selenium WebDriver,” [Online]. Available: https://www.selenium.dev/documentation/en/webdriver/understanding_the_components/. [Hozzáférés dátuma: 5. május 2021].

[13] R. Shetty, *Selenium WebDriver with Java -Basics to Advanced+Frameworks*, Udemy.

[14] „Selenium,” [Online]. Available: https://www.selenium.dev/downloads/. [Hozzáférés dátuma: 5. május 2021].

[15] „Selenium - Types of testing,” [Online]. Available: https://www.selenium.dev/documentation/en/introduction/types_of_testing/. [Hozzáférés dátuma: 10 május 2021].

[16] „TestNG,” [Online]. Available: https://testng.org/doc/. [Hozzáférés dátuma: 5. május 2021].

[17] „Cucumber - BDD,” [Online]. Available: https://cucumber.io/docs/bdd/. [Hozzáférés dátuma: 10 május 2021].

[18] „Maven,” [Online]. Available: https://maven.apache.org/. [Hozzáférés dátuma: 5. május 2021].

[19] „BME AUT - Docker,” [Online]. Available: http://bmeaut.github.io/snippets/snippets/0706_DockerBev/. [Hozzáférés dátuma: 20 május 2021].

[20] „Selenium - Web Element,” [Online]. Available: https://www.selenium.dev/documentation/en/webdriver/web_element/. [Hozzáférés dátuma: 5. május 2021].

[21] „Apache - Log4j 2,” [Online]. Available: https://logging.apache.org/log4j/2.x/. [Hozzáférés dátuma: 13 május 2021].

[22] „Selenium - Page Object Model,” [Online]. Available: https://www.selenium.dev/documentation/en/guidelines_and_recommendations/page_object_models/. [Hozzáférés dátuma: 12 május 2021].

[23] „Eclipse,” [Online]. Available: https://www.eclipse.org/downloads/. [Hozzáférés dátuma: 15 május 2021].

[24] „Maven download,” [Online]. Available: https://maven.apache.org/download.cgi.

[25] „Selenium download,” [Online]. Available: https://www.selenium.dev/downloads/. [Hozzáférés dátuma: 15 május 2021].

[26] „Docker,” [Online]. Available: https://docs.docker.com/desktop/. [Hozzáférés dátuma: 20 április 2021].

[27] „Cucumber - Gherkin,” [Online]. Available: https://cucumber.io/docs/gherkin/reference/#background. [Hozzáférés dátuma: 10 május 2021].

[28] I. Z. Dr. Berta, Nagy E-Szignó könyv, Microsec Kft., 2011.

# Köszönetnyilvánítás

Ezúton szeretném megköszönni Kocsis Gábornak és a Contum Kft-nek szakdolgozatomhoz és egész kooperatív képzésem alatt nyújtott segítségét, szakértelmét és türelmét.

Családomnak hálás vagyok az egész képzésem alatt nyújtott támogatásáért, különösen testvéremnek, aki nélkül nem jelentkeztem volna erre a szakra és nem jutottam volna el idáig.

# Függelék

A Mérések alfejezet részletes eredményeit a lenti táblázatok mutatják.

| sorszám | headless | szálak száma | futási idő(ms) | hiba |
|---|---|---|---|---|
| 1 | false | 3 | 11776 | 0 |
| 2 | false | 3 | 12159 | 0 |
| 3 | false | 3 | 11730 | 0 |
| 4 | false | 3 | 11648 | 0 |
| 5 | false | 3 | 12040 | 0 |
| 6 | false | 3 | 11826 | 0 |

| sorszám | headless | szálak száma | futási idő(ms) | hiba |
|---|---|---|---|---|
| 1 | true | 3 | 8 127 | 0 |
| 2 | true | 3 | 7 703 | 0 |
| 3 | true | 3 | 7 890 | 0 |
| 4 | true | 3 | 7 640 | 0 |
| 5 | true | 3 | 7 424 | 0 |
| 6 | true | 3 | 7 768 | 0 |

| sorszám | headless | szálak száma | futási idő(ms) | hiba |
|---|---|---|---|---|
| 1 | false | 2 | 14057 | 0 |
| 2 | false | 2 | 14067 | 0 |
| 3 | false | 2 | 13457 | 0 |
| 4 | false | 2 | 12312 | 0 |
| 5 | false | 2 | 12791 | 0 |
| 6 | false | 2 | 12644 | 0 |

| sorszám | headless | szálak száma | futási idő(ms) | hiba |
|---|---|---|---|---|
| 1 | true | 2 | 9607 | 0 |
| 2 | true | 2 | 9743 | 0 |
| 3 | true | 2 | 9537 | 0 |
| 4 | true | 2 | 9063 | 0 |
| 5 | true | 2 | 9344 | 0 |
| 6 | true | 2 | 9479 | 0 |

| sorszám | headless | szálak száma | futási idő(ms) | hiba |
|---|---|---|---|---|
| 1 | false | 1 | 17409 | 0 |
| 2 | false | 1 | 17356 | 0 |
| 3 | false | 1 | 16282 | 0 |
| 4 | false | 1 | 17122 | 0 |
| 5 | false | 1 | 16807 | 0 |
| 6 | false | 1 | 17202 | 0 |

| sorszám | headless | szálak száma | futási idő(ms) | hiba |
|---|---|---|---|---|
| 1 | true | 1 | 12168 | 0 |
| 2 | true | 1 | 11883 | 0 |
| 3 | true | 1 | 12312 | 0 |
| 4 | true | 1 | 12507 | 0 |
| 5 | true | 1 | 12168 | 0 |
| 6 | true | 1 | 12329 | 0 |