Webtesztelési módszerek tanulmányozása

Tartalom

[Bevezető 3](#_Toc12800778)

[A kutatott technológiák bemutatása 3](#_Toc12800779)

[Selenium 3](#_Toc12800780)

[Mi a Selenium? 3](#_Toc12800781)

[Miért Selenium? 4](#_Toc12800782)

[Selenium IDE 5](#_Toc12800783)

[Selenium WebDriver 7](#_Toc12800784)

[WebElement-ek és betályolásuk 10](#_Toc12800785)

[Szakirodalmi tanulmány 13](#_Toc12800786)

[Autómatizált tesztelési keretrendszerek 13](#_Toc12800787)

[Az automatizált tesztelés tipikus buktatói 15](#_Toc12800788)

[Megvalósított rendszer bemutatása 17](#_Toc12800789)

[A Webalkalmazás: 17](#_Toc12800790)

[Webalkalmazás technológiái 18](#_Toc12800791)

[Webalkalmazás architektúra 18](#_Toc12800792)

[A webalkalmazás funkciói 20](#_Toc12800793)

[A teszt keretrendszer 23](#_Toc12800794)

[Selenium 23](#_Toc12800795)

[Selenium WebDriver 23](#_Toc12800796)

[Page Object Model 24](#_Toc12800797)

[Cucumber 26](#_Toc12800798)

[JUnit-al való integráció 29](#_Toc12800799)

[Teszt keretrendszer arhitektúra 30](#_Toc12800800)

[Mérések 41](#_Toc12800801)

[Következtetések és tapasztalatok 51](#_Toc12800802)

[Továbbfejlesztési lehetőségek 51](#_Toc12800803)

[Idézett forrásmunkák 52](#_Toc12800804)

Ábrajegyzék

[1. ábra - Selenium arhitektúra Forrás: [3] 4](#_Toc12800805)

[2. ábra - iDatalabs kutatásai szerint több, mint 29000 cég használja a Seleniumot [4] 5](#_Toc12800806)

[3. ábra - Selenium IDE kezdőlap 6](#_Toc12800807)

[4. ábra – Selenium IDE példa teszteset 7](#_Toc12800808)

[5. ábra - A webalkalmazás rétegei 18](#_Toc12800809)

[6. ábra - Egyszerüsített térkép az alkalmazás oldalairól 19](#_Toc12800810)

[7. ábra - Teljes oldaltérkép 20](#_Toc12800811)

[8. ábra - Navigációs menü alapesetben 20](#_Toc12800812)

[9. ábra - Navigációs menü User jogosultsággal (If Authenticated) 20](#_Toc12800813)

[10. ábra - Navigációs menü Admin jogosultsággal (If Admin) 20](#_Toc12800814)

[11. ábra - User Settings és Sign out oldalak 22](#_Toc12800815)

[12. ábra - A webalkalmazás oldalai megfeleltetve egy-egy Java objektumnak 26](#_Toc12800816)

[13. ábra - BDD, ahogy TDD-re épül Forrás: [15] 27](#_Toc12800817)

[14. ábra - A teszt keretrendszer rétegei 30](#_Toc12800818)

[15. ábra - JUnit TestRunner.java 31](#_Toc12800819)

[16. ábra - Cucumber Feature fájlok 33](#_Toc12800820)

[17. ábra - Selenium WebDriver könyvtárai 35](#_Toc12800821)

[18. ábra - Feature fájlok java kódjai 36](#_Toc12800822)

[19. ábra - AddNewAuthor.feature - Scenario1 36](#_Toc12800823)

[20. ábra - AddNewAuthorStepDefinition - Scenario1 37](#_Toc12800824)

[21. ábra - Selenium WebDriver implementációja 38](#_Toc12800825)

[22. ábra - WebDriver réteg UML diagramja 41](#_Toc12800826)

[23. ábra - Internet sebessége Forrás: www.speedtest.net 41](#_Toc12800827)

# Bevezető

Probléma: a problémát a munkahelyen fedeztem fel, a jelenlegi munkafolyamatot szeretném optimalizálni autómatizált teszteléssel.   
Röviden a munkám: Product Lifecycle Management szoftvert szabunk testre klienseknek. A szoftver neve Windchill, ami Spring keretrendszerre épülő webes applikáció.   
A probléma pedig ott adódik, hogy a fejlesztéseinket tesztelni kell. Jelenleg kézzel csináljuk ezt. Ez lassú. És valahányszor release-re kerül a sor, újra kell csinálni kézzel minden tesztet, hogy meggyőződjünk róla, hogy minden új funkció az elvártnak megfelelően üzemel és a már meglévő funkciókat nem befolyásolják. Ezt a teszt környezeten egy teszter végzi. Ezen rengeteg idő jelentős részét meg lehetne takarítani, ha a teszteket autómatizálnánk. Plusz, mivel ugyanazt a Windchill-t több projektben is használjuk, ezért a tesztek nagy része újrahasznosítható lenne minimális módosításokkal. A másik probléma pedig az, hogy nem lehet elmenteni és újra futtatni a teszteket, mert ugye azt mindig kézzel csináljuk. Itt jön képbe az autómatizált tesztelés. (Selenium.) Autómatizálhatjuk a teszteket, szimulálhatjuk a végfelhasználó viselkedését és akárhányszor újra futtathatjuk azokat. Ennek meg van az az előnye is, hogy így megbizonyosodhatunk arról, hogy egy már meglévő funkciót nem rontottunk el egy új fejlesztéssel.

A Cucumber pedig azért hasznos választás, mert így a tesztek olvashatóak és szerkeszthetőek lesznek nem csak a fejlesztők, hanem a menedzserek számára is.

PTC Community oldalon publikálhatom a kutatást.

A cég belső portálján is publikálhatom az eredményeket.

Cél: hogy bebizonyítsam az autómatizált tesztek hozzájárulnak a szoftver minőségének javulásához és ezzel időt, energiát és nem utolsó sorban pénzt takaríthatunk meg. Valamint választ adni arra a kérdésre, hogy mikor érdemes autómatizálni a teszeket

# A kutatott technológiák bemutatása

## Selenium

A Selenium egy keretrendszer, melynek segítségével autómatizálhatjuk a webböngészőt. Hogy mit kezd ezzel a hatalommal a felhasználó, az teljesen rajta áll. Első sorban a webes alkalmazások tesztelése céljából történő automatizálására szolgál, de nem merül ki ennyiben. Például az unalmas web alapú adminisztratív teendőket is automatizálni lehet vele. [1]

Nagyon gyakran emlegetik a Seleniumot és a WebDrivert egymás mellett. Aki most találkozik először a kifejezésekkel, bizonyára összezavarodhat, hogy mi a különbség a kettő között. Régen a Selenium és a WebDriver két különálló projekt volt, a lényege mindkettőnek ugyanaz volt: hogy irányíthassuk a böngészőt, csak másképp volt megvalósítva. [2] Később, 2007-ben a két projekt egyesült és kiadták a Selenium 2.0 –t, amely tartalmazta mindkét projektet. [1]



1. ábra - Selenium arhitektúra Forrás: [3]

### Miért Selenium?

Azért döntöttem Selenium mellett, mert figyelembe kell venni, hogy a csapat miben tapasztalt. Javaban van jártasságunk, ezért ésszerű döntés lenne, ha a teszteket is ugyanabban a nyelvben és környezetben tudnánk írni, továbbá a Selenium jól integrálható JUnit-al és Maven-el. A tanulási görbe nem meredek azok számára, akiknek van jártassága Java-ban és objektum orientált programozásban. Szempont volt még a költség is: mivel a Selenium nyílt forráskódú és bárki számára ingyenesen elérhető, ezért erre nem jár plusz költségekkel. Selenium további előnyei közé tartozik, hogy sok böngészőt támogat. Mivel a Windchill hivatalosan az InternetExplorer és a Google Chrome böngészőt támogatja, fontos szempont, hogy a böngészők közötti különbségek ne jelentsenek gondot.

A WebDriver miatt képes a végfelhasználó tevékenységét imitálni a böngészővel, mint például billentyű leütések, drag-and-drop, vissza gombra való kattintás stb. Voltaképpen amit az egér és billentyűzet használatával végre tudok hajtani a böngészőben, ugyanazt el tudom érni, csak kódból.

Plusz pont a Seleniumnak, hogy nagy a felhasználótábora, így jó eséllyel találunk megoldást a gyakori problémákra. Az iDatalabs szerint a Selenium piaci részesedése több, mint 27%. [4]



2. ábra - iDatalabs kutatásai szerint több, mint 29000 cég használja a Seleniumot [4]

### Selenium IDE

A Selenium IDE nem más, mint egy egyszerű bővítmény a böngészőhöz. Célja, hogy programozási tudás nélkül is automatizálhassunk. Használni is egyszerű, mindössze hozzá kell adnunk a Selenium IDE kiterjesztést és az ikonjára kattintva előugrik a kezelőfelülete, amit a lenti „Ábra 2 - Selenium IDE kezdőlap” szemléltet. 

3. ábra - Selenium IDE kezdőlap

Létre hozhatunk projekteket és kezdhetjük az automatizált tesztjeink készítését. Azért nem kell hozzá programozói ismeret, mert amiután beállítottuk a projektet, a Selenium IDE-vel fel tudjuk venni és visszajátszani kölcsönhatásainkat a böngészővel. Ezek lehetnek a teszt eseteink. Például: vegyük alapul a google.com oldalt. Beírom a kereső input-ba, hogy „selenium ide” és megnyomon az ENTER billentyűt, majd rákattintok az első találatra és megnézem, hogy az oldal címe megegyezik-e „Selenium IDE · Open source record and playback test automation for the web”-el? Ezt az egyszerű példát a lenti „Ábra 3 – Selenium IDE példa teszteset” ábrázolja.



4. ábra – Selenium IDE példa teszteset

Előnye, hogy nehéz telepítés nélkül működik és az egyszerű eseteket gyorsan, egyszerűen lehet tesztelni. Egyszerű eset alatt azt a tesztet értem, ami nem több, mint 15 lépésből áll. Egy projekt állhat részekre bontott tesztesetekből, melyeket bármikor újra felhasználhatunk. Hátránya, hogy a bonyolultabb eseteket már nehezen tudjuk szimulálni. Bonyolult teszteset az, ami 15 lépésnél többől áll vagy 5-nél több összetett műveletet tartalmaz. További hátulütő, hogy csak Google Chrome és Mozilla Firefox alá telepíthető a bővítmény.

Önmagában a Selenium IDE édeskevés egy jó autómatizált tesztelési keretrendszer összeállításához, de úgy gondolom, hogy nagy segítség lehet, amikor be kell tájolni egy elemet, vagy ha meg szeretnénk gyorsan mutatni, hogy mi lenne az elvárt működése a teszteknek.

### Selenium WebDriver

Úgy gondolom, hogy a WebDriver a Selenium sikerének igazi kulcsa. Való igaz, hogy a Selenium IDE enged programozói tudás nélkül automatizálni, de ha nagyon hosszú a teszteset, akkor szerintem átláthatatlanná válik. Ebben az esetben érdemes a WebDrivert választani, bár ide már szükségeltetik programozói tudás is, de az itt megírt teszteket futtathatjuk a támogatott böngészőkben, mint pl: Chrome, Firefox, Internet Explorer, Safari, Opera. Általában minden böngésző JavaScript-motorja eltér a többitől, és minden böngésző eltérő módon értelmezi a HTML-címkéket. A WebDriver API úgy irányítja a webböngészőt, ahogyan azt egy felhasználó tenné. Alapértelmezés szerint a FirefoxDriver hozzá van adva a Selenium könyvtárához, de a Chrome, IE és az Opera számára is vannak könyvtárak, melyeket le lehet tölteni. [5]

Vegyünk egy egyszerű tesztesetet, amikor a felhasználó nyit egy böngészőablakot, rákeres a „selenium” kulcsszóra és ellenőrzi, hogy az első találat a Selenium hivatalos oldala-e, majd bezárja a böngészőt. Ez az egyszerű eset a Selenium WebDriver beépített függvényeit használva mindössze nagyjából tíz sor kóddal leírható. Erre az esetre egy naiv implementáció így nézne ki:

**public** **void** isSeleniumTheFirstResult() {

// A webböngésző driver inicializálása, jelen esetben Chrome

WebDriver driver = **new** ChromeDriver();

// www.google.com weboldal betöltése

driver.get("http://www.google.com");

// A kereső input betájolása

WebElement searchField = driver.findElement(By.*name*("q"));

// selenium kulcsszó beírása az input mezőbe

searchField.sendKeys("selenium");

// Enter billentyű “leütése”

searchField.sendKeys(Keys.***RETURN***);

// Az első oldal találatainak listája

List<WebElement> searchResults = driver.findElements(By.*className*("g"));

// Az első találat tárolása a listából

WebElement firstResult = searchResults.get(0);

// Ellenőrzés, hogy az első találat a Selenium hivatalos weboldala-e

**if**(firstResult.getText().contains("https://www.seleniumhq.org/")) {

System.***out***.println("Selenium official page is the first result!");

} **else** {

System.***out***.println("Selenium official page is NOT the 1st result!");

}

// Böngészőablak bezárása

driver.quit();

}

1. példakód - WebDriver naiv implementáció

Aki először találkozik Selenium WebDriver-el és komment nélkül olvasná a kódot, ha nem is értené pontosan, de szerintem sejtené, hogy mit csinálhat a fenti részlet. Úgy gondolom, hogy elég beszédes függvényneveket adtak a Selenium fejlesztői, íme a fontosabbak, melyeket Prashanth Sams is kiemel könyvében [5]:

* get();

A fenti metódus egy URL címet vár, szöveges formában paraméterként, és annyit csinál, hogy a böngészőt az adott címre irányítja.

Használata: driver.get("http://www.google.com");

* click();

Mint ahogy a neve is sugallja, a click() függvény az egérrel való kattintás műveletét végzi, arra a WebElement-re (lennebb lesz szó róla), amelyre meghívtuk.

Használata: driver.findElement(By.locatorType(„path”));

* sendKeys();

A sendKeys() metódus arra hivatott, hogy egy szövegmezőbe beírhassunk, beilleszthessünk szöveget.

Használata: driver.findElement(By.locatorType(„path”));

* clear();

A fenti metódus arra szolgál, hogy egy szövegmező tartalmát kiürítsük.

Használata: driver.findElement(By.locatorType(„path”));

* submit();

A submit() metódus hasonlít a click() metódushoz, azonban ez egy űrlap <form> elküldésére szolgál. Gyakran használt függvény az ENTER billentyű lenyomása helyett.

Használata: driver.findElement(By.locatorType(„path”));

* getTitle();

Mint neve is sugallja, ez a függvény visszatéríti az aktuális oldal címét. Pontosabban a <title></title> HTML elemensek közötti szöveget.

Használata: driver.getTitle();

* getCurrentUrl();

Ez a függvény visszatéríti az aktuális oldal URL címét.

Használata: driver.getCurrentUrl();

* getPageSource();

Ez a funkció visszatéríti a betöltött weblap teljes forrását. Azonban a módosított DOM az aszinkron (Ajax) hívások miatt nem tükröződik néhány böngészőben. Ehelyett visszaadja a korábban betöltött weboldal forrását. A getPageSource() módszer nem ajánlott a JavaScripteket aszinkron módon betöltő weblapok számára.

Használata: driver.getPageSource();

Gyakori példa a használatára, hogy ellenőrizzük tartalmaz-e az oldal egy adott szöveget.

* close();

A close() metódus kilép vagy bezárja az aktív böngészőablakot és befejezi a munkamenetet.

Használata: driver.close();

* quit();

Ez a funkció leállítja a futó illesztőprogramot (driver) és bezárja az összes böngészőablakot az aktív munkamenet befejezésével.

Használata: driver.quit();

### WebElement-ek és betájolásuk

A Selenium tesztek építőkövei a WebElementek és ezek betályolása. Ahogy Avasarala Satya is írja könyvében [3], egy weblap számos különböző HTML elemből épül fel, mint például gombok, linkek, űrlapok, szövegdobozok stb. Ezek az elemek a Seleniumban WebElement objektumként szerepelnek. Nézzünk egy egyszerű regisztrációs oldal HTML kódját:

<!DOCTYPE html>

<html>

<body>

<form id=*"registrationForm"*>

<label>Enter Username: </label>

<input type=*"text"* name=*"Username"* />

<label>Enter Password: </label>

<input type=*"password"* name=*"Password"* />

<label>Repeat Password: </label>

<input type=*"password"* name=*"PasswordRepeat"* />

<input type=*"submit"* value=*"Register"*/>

</form>

</body>

</html>

2. példakód - HTML form

A fenti HTML oldal a <html>,<body>,<form>,<label>,<input> elemekből épül fel, melyek összességében alkotják az egyszerű oldalt. Vegyük például a következő elemet: <label>Enter Username: </label> - itt a <label> a WebElement kezdetét jelzi, az „Enter Username:” a WebElement szövegét alkotja, a </label> pedig az elem befejező tag-je. Hasonlóképpen a <input type="text" name="Username" /> - itt a type és a name a WebElement attribútumai lesznek a tartalmukkal együtt.

Egy weboldal automatizálása többnyire arról szól, hogy ezeket az elemeket megtaláljuk és a felhasználó által végrehajtani kívánt műveleteket végezzük el rajtuk.

Az oldal elemensei (WebElement) többféleképpen betájolhatóak:

1. id alapján – By.id
2. név attribútum alapján – By.name
3. css szelektor alapján – By.cssSelector
4. osztály név alapján – By.className
5. xpath alapján – By.xpath
6. link szövege alapján – By.linkText
7. Részleges link szöveg alapján – By.partialLinkText
8. HTML tag alapján – By.tagName

Példa a használatra:

WebElement searchForm = driver.findElement(By.*id*("search-form"));

3. példakód - findElement() függvény használata

Az összes fentebb felsorolt lokátort hasonlóképpen kell használni. A fenti kódrészlet tárolja searchForm nevű változóban a „search-form” id-val rendelkező formot és összes alatta lévő elemet WebElement típusú objektumban.

Érdemes a lokátorokat a következő prioritás szerint használni: id, name, css, xpath. A cssSelector locator típus jó választás lehet dinamikus weboldalak esetén. Ahogyan azt Prashanth [5] és  Satya [3] ajánlja, ha csak lehet érdemes elkerülni az elemek keresését xpath használatával. Prashanth arra hívja fel a figyelmet, hogy az Internet Explorer gyakran hibásan adja vissza az így lekért elemeket, Satya pedig azért ajánlja, hogy csak a legvégső esetben használjuk, mert lassítja a teszteket, mivel a WebDriver így az egész oldalt ki kell elemezze, hogy megtalálja a keresett elemet.

Ez után már csak azt kell megérteni, hogy a findElement(By.locatorType(„path”)) és a findElements(By.locatorType(„path”)) függvények hogyan működnek:

* findElement();

A fenti függvény visszatérési értéke egy WebElement. Pontosabban az az elem, melynek lokátorát beadtuk paraméterként. Ha a lokátor több elemen is rajta van, akkor az elsőt téríti vissza. Ha egyetlen elem sem a tartalmazza a lokátort, akkor NoSuchElementException-t dob a rendszer.

* findElements();

Nem hiába a többes szám, a findElements() függvény visszatérít egy WebElement-eket tartalmazó listát, amelyekre a paraméterként beadott lokátor stimmel. Amennyiben nem található egy elem sem, üres listát kapunk vissza.

Tárolni nem csak WebElementben lehetséges, ott van például a Select. A Select kifejezetten azért lett implementálva a Seleniumba, hogy lenyíló listát könnyebben kezelhessük általa.

* Select select = new Select(By.locatorType(„path”));

Mint látható, ha a Select konstruktorába beadjuk a lenyíló lista útvonalát, akkor létrehoz nekünk egy objektum reprezentációt belőle. Ez után használhatunk olyan függvényeket, mint selectByVisibleText(), selectByValue(), getOptions() – WebElement listát térít vissza stb.

Ha a fentieket tudjuk és értjük a működésüket, ennyi már elég, hogy Selenium teszteket kezdjünk írni, de ha egy keretrendszert szeretnénk építeni, akkor oda már több szempontot kell figyelembe vennünk. Ezeket bővebben tárgyalom a A teszt keretrendszer című fejezetben.

# Szakirodalmi tanulmány

## Autómatizált tesztelési keretrendszerek

Ha automatizált tesztelésre kerül a sor, és a már létező technológiák után kutatunk, bajban lehetünk a széles választék miatt. Bár a kör nagyon hamar leszűkíthető, ha letisztázzuk, hogy mik az igényeink és miben van elegendő jártasságunk. A piac meglehetősen széles, és minden igényre lehet találni megoldásokat. Vannak olyanok, amelyek üzemeltetéséhez nem szükséges programozói háttértudás és vannak olyanok, amelyekhez kell tudni kódolni. Vannak, amelyek felhő alapú alkalmazások tesztelésére vannak kifejlesztve, sőt, olyant is találni, amelyek mesterséges intelligenciával vannak felvértezve és elemzik a teszteket. Íme, néhány számomra érdekesnek tartott megvalósítás az említett technológiákra a Seleniumon kívül:

1. Cypress [6]

A Cypress egy Seleniumhoz hasonló keretrendszer. Az 1.0-s verziót 2017-ben adták ki, ekkor vált a projekt nyílt forráskódúvá és a Windows operációs rendszer is ekkor lett támogatva. Maga a Cypress Test Runner, mint technológia ingyenes, JavaScript nyelvben lehet írni és futtatni a teszteket. Van egy Cypress Dashboard Service nevű projekt is, ennek segítségével a fenti teszteket futtathatjuk, debuggolhatjuk valós időben a böngészőben, ám ez a szolgáltatás 3 felhasználó és 500 teszteset után fizetős. Tetszik, hogy a weboldalon részletes tutorial van, ahol bemutatják a keretrendszert, videókkal együtt és azt is, hogy hogyan kell használni az egyes funkciókat. Selenium esetén ez a funkció hiányzik. A Cypress-t olyan cégek használják, mint például NASA, KIWI.COM, DHL. Legalábbis a weboldalukon ezt írja.

1. Robot Framework [7]

A Robot Framework egy nyílt forráskódú, Python nyelvre alapuló, platformfüggetlen keretrendszer. Abba a csoportba tartozik, amely használatához nem feltétlenül szükségeltetik haladó programozói tudás, mert ez a keretrendszer úgynevezett kulcsszó – vezérelt szintaxist használ. Ez azt jelenti, hogy elég a megfelelő kulcsszavakat használni ahhoz, hogy irányítsuk a böngésző viselkedését. A kulcsszavak külső könyvtárakból jönnek. Szükség esetén írhatunk magunknak saját kulcsszavakat, ha nincs olyan könyvtár, ami kielégítené az igényeinket, de cask Python nyelven tehetjük meg ezt, persze ide már kell programozói tudás Előnye, hogy könnyen tanulható, egyszerű esetekben gyorsan használható és támogatja a Gherkin nyelvet is. A Robot Framework használatos a Nokia, ABB és Cisco által is.

1. Katalon Studio [8]

A Katalon Studio egy ingyenesen használható, de nem nyílt forráskódú keretrendszer. A rendszer a Seleniumra és Appiumra épül. Úgy kell elképzelni, mintha a Seleniumnak egy grafikus felhasználói felülettel ellátott verzióját kezelnénk. Erőssége, hogy web, mobil és API teszteket is lehet autómatizálni, valamint out-of-the-box lehet integrálni az ismertebb countinuous integration rendszerekkel és jó a jelentés kezelése is, például beállítható, hogy e-mailben küldje el a jelentéseket egy magadott címre a megadott formátumban. Támogatja a BDD-t és Gherkin nyelv szintaxist is. Van lehetőség felhő alapú alkalmazások tesztelésére is. A Katalon stratégiája az, hogy a segítségért (support) és a betanításért (training) kérnek pénzt. 1 évi support egy projekt esetén, 24 órás válaszidővel havi 199 dollárba, vagyis évi 2388 dollárba kerül. Hátrányára szóljon, hogy a felhasználótábora nem túl nagy. Olyan cégek használják, mint például SAP, Clayton homes, TATA.

1. Screenster [9]

A Screenster, mint ahogy neve is sugallja, közeli kapcsolatban áll a képernyővel. Ehhez a keretrendszerhez nem szükséges programozói ismeret, lévén, hogy vedd fel és játszd le elven működik. Az a mottójuk, hogy “Autómatizáld az első tested 5 perc alatt”. Nem szükséges telepíteni, innen is származhat a mottó. A Screenster egy vizuális felhasználói felülettel ellátott teszt automatizálásra alkalmas interfészt biztosít a webes alkalmazásokhoz. A UI teszt rögzítésekor a Screenster lebontja a DOM-ot, és az egyes UI-összetevőket a képernyőn végrehajtott módon egyezteti. Így a tesztelő megerősítheti az egyes oldalakon lévő elemeket, hogy valóban úgy néz-e ki és valóban az-e aminek lennie kell. Nem nyílt forráskódú és csak egy felhasználóig ingyenes, azután már 5 felhasználóig 25 dollár havonta, 20 felhasználóig 45 dollár, korlátlan felhasználó esetén pedig 75 dollár havonta a költsége. Olyan cégek használják, mint Google, Samsung, Verizon.

1. TestCraft [10]

A TestCraft egy 100% kódolást nem igénylő, felhő alapú UI autómatizáló keretrendszer, mely a Seleniumra épül. Programozás helyett itt a tesztelők vizuális felületen hozzák létre a teszt eseteik lépéseit. Úgy van kitalálva a rendszer, hogy mesterséges intelligencia segítse a tesztek működőképességét, és ha változás történik, elvileg autómatikusan adaptálódik hozzá a rendszer. Ingyen próbaverzió elérhető, de az árazásról nem tájékoztat a hivatalos weboldal.

A fenti listában kiemeltem a számomra érdekesebb megvalósításokat, de nem kellene, hogy itt véget érjen, mert még rengeteg alternatíva van a piacon, melyeket hosszan lehetne tárgyalni, de az már nem célja ennek a dolgozatnak.

Mint a fenti 5 említett keretrendszer rövid bemutatása után leszűrhetjük, mindenik rendszer nagyjából ugyanarra képes, de mindenik másképp csinálja. Hogy melyiket érdemes választani, az teljes mértékben az igényeinktől és a pénztárcánktól függ.

## Az automatizált tesztelés tipikus buktatói

Matthew Heusser, aki 20 éves tapasztalattal rendelkezik tesztelés területén, a következő hat pontot említi cikkjében [11], mint a leggyakoribb hibák, amik előjöhetnek, amikor teszteket automatizálunk:

1. Az alkalmazás tesztelése csak a felhasználói felületről

Matthew állítása szerint ha csak a felhasználói felületet használjuk arra, hogy teszteljük az alkalmazás minden funkcionalitását, akkor az előbb- utóbb, ahogy nő a projekt, ahhoz vezet, hogy a tesztek sokáig fognak futni. Ezért az eredményeket nem instant, hanem másnap kapjuk meg. Emiatt inkonzisztencia alakul ki a tesztelés és a fejlesztés között. A probléma megoldására ezért azt ajánlja, hogy ne csak a felhasználói felületen keresztül teszteljünk, hanem írjunk unit és integrációs teszteket is. Úgy gondolom, hogy egy kis projektnél mint például egy webáruház még nem jelentene gondot ez, ám egy nagyobb és összetettebb projektnél érdemes odafigyelni erre a pontra.

1. Build – Test – Deploy munkafolyamat figyelmen kívül hagyása

Matthew azt javasolja, hogy ne kézzel akarjuk lefuttatni az automatizált teszteket, hanem inkább integráljuk azt a munkafolyamatba, hogy minden build után autómatikusan lefussanak. Így mentesíthető az is, hogy esetleg elfelejtsük lefuttatni őket. Ne akarjuk rögtön, 100%-ban tesztelni az alkalmazást, hanem szépen, fokozatosan adjuk hozzá az új teszteket, melyeket átgondolunk, mert a tesztek igazításának és fenntartásának is költségei vannak.

1. Teszt adatok bevitele a felhasználói felületről

Említi még a cikk azt is, hogy ha csak lehet, kerüljük a teszt adatok bevitelét a felhasználói felületről, mert ez lassú folyamat és rengeteg értékes időt elvesz. Úgy érti, hogy egy a teszt adatbázist a felhasználói felületről hozzuk létre. Helyette inkább teszteljük, hogy működik-e a felület egy – két teszt adattal, de magát az adatokat, amik a teszteléshez szükségesek, inkább importáljuk az adatbázisba és minimizáljuk a kézzel történő bevitelt.

1. Tesztek távoltartása a fejlesztéstől

Egy másik probléma, amit Matthew kiemel, hogy a tesztelés általában egy már meglévő rendszeren zajlik, főleg a felhasználói felület tesztelése esetén van ez így. Ezért a tesztek mindig a fejlesztés mögött lesznek. Ennek elkerülése érdekében azt javasolja, hogy vonjuk be a tesztelőket, a programozókat és a „product ownert” is a folyamatba, beszéljék meg és legyen egy közös elképzelésük a rendszerről.

1. Teszt kód Másolása/Beillesztése

Ez a bekezdés arra a keretrendszerekre vonatkozik, melyeket programkódból irányítunk. Matthew arra hívja fel a figyelmet, hogy ha nincs megtervezve előre a teszt rendszer, akkor előbb- utóbb egy nagy spagetti lesz az egész rendszer, átláthatatlan, másolt, duplikált kódrészletekkel, és ha valaki változtatni akar a későbbiek folyamán a kódban, akkor az bizony fogja emlegetni a nevünket. Ennek a kiküszöbölésére találták ki a page-object mintát, mely segít a kód rendszerezésében.

1. Gondolj a tesztekre úgy, mint egy nagy, összefüggő alkalmazásra

Az utolsó pontban Matthew arra hívja fel a figyelmet, hogy érdemes a rendszert úgy kitalálni, hogy ne csak a technikai emberek értsék a teszteket. Valahogyan emberi szemnek olvashatóvá kell tenni a teszteket, ez által a teszt esetek karbantarthatóak lesznek több csoport számára is, nem csak a technikai emberek számára.

A fenti gondolatokkal teljes mértékben egyet értek és még kiegészíteném néhánnyal, amit saját tapasztalataim alapján szűrtem le.

* Megfelelő keretrendszer kiválasztása

Erre azért hívom fel a figyelmet, mert, mint fennebb láthattuk, rengeteg megoldás létezik a piacon teszt automatizálásra. Vegyük figyelembe azt, hogy mi a célunk és azt, hogy miben van esetleg tapasztalatunk, amire építhetjük az új keretrendszer megtanulását.

* A teszteket előre kell tervezni és kigondolni

Minden tesztet előre meg kell tervezni és világos kell legyen az, hogy mit fogunk tesztelni és hogy valóban kell-e az adott teszt. Egyszóval többet ad-e, mint amennyi belefektetett időt elvesz. A tesztekre befektetett idő általában akkor térül meg, ha többször kell lefuttatni őket.

* A tesztrendszer felépítését érdemes előre megtervezni

Ezt azért érdemes betartani, mert ha nincs egy rendszer a tesztjeinknek, akkor hamar átláthatatlan masszává tudja kinőni magát az egész rendszer és ha módosítani kell valahol, akkor összeomolhat. Ebben lehet segítségünkre, ha betartjuk a page-object mintát és élünk az objektum orientált programozás nyújtotta lehetőségekkel.

* Túl sokat akarunk tesztelni

Szerintem nem érdemes 100%-ban lefedni az alkalmazást tesztekkel. Úgy gondolom, hogy bár papíron jól mutat, viszont több energiát emészt fel az, hogy az alkamazás minden apró funkcióját teszteljük, mint amennyi hasznot valóban hoz.

# Megvalósított rendszer bemutatása

A megvalósított rendszer egy web alkalmazás és ennek a webalkalmazásnak a különböző funkcióit tesztelő Selenium keretrendszer Cucumber és JUnit integrációval. Ezen a rendszeren végeztem a méréseket, hogy megtudjam mikortól érdemes automatizálni a tesztjeinket.

## A Webalkalmazás:

A webalkalmazás a MyLibrary nevet kapta, ami egy egyszerű könyv és költő/író nyilvántartó (ajánló) webalkalmazás, ahol az admin joggal rendelkező felhasználók tudnak hozzáadni új könyveket és költőket/írókat a rendszerhez. Minden hozzáadott könyvhöz értelem szerűen rendelhető a megfelelő könyv írója is. Az ötlet annyi, hogy felhasználók ezután tudják értékelni a könyveket és az írókat. Ha új könyvet szeretnének látni a rendszerben, akkor azt egy kapcsolat űrlap segítségével tudják jelezni az adminok számára. Ezen kívül a felhasználók fel tudják venni egymással a kapcsolatot és tudnak véleményt kérni az egyes könyvekről és beszélgetni, vitázni róluk. Mivel a dolgozat célja nem a tökéletes webalkalmazás kifejlesztése volt, hanem az, hogy legyen egy alkalmazás, amin tudunk automatizált teszteket futtatni és kísérletezni rajta, így a fenti ötlet annyira van kivitelezve, hogy elég legyen a tesztek futtatásához és az eredmények leméréséhez.

### Webalkalmazás technológiái

Fejlesztői környezetként az Eclipse IDE volt segítségemre. Az alkalmazás technológiái röviden és tömören a front-end részről a back-end fele haladva sorban:

A front-end dizájnja Bootstrap 4 segítségével lett megvalósítva, ami egy front-end komponens gyűjtemény. Azért szeretem a Bootstrap-et, mert így egy kinézetre összefüggő alkalmazást tudunk építeni és nem kell a formázással tölteni az időt, ráadásul az így írt alkalmazás reszponzív lesz.

Az alkalmazás back-endjét Java nyelven írtam, SpringBoot technológia képezi az alapját. Azért SpringBoot, mert van benne tapasztalatom és mert nagyon kevés konfigurációval lehet Spring alapú webalkalmazásokat építeni.

Továbbá használtam még Thymeleaf szerver oldali sablon motort (template engine) Javahoz, ami abban segített, hogy az adatokat dinamikusan és egyszerűen tudjam a back-endről a front-endre küldeni és megjeleníteni.

Az egyszerűség kedvéért H2 típusú Java SQL memória adatbázist használtam az adatok tárolására. Az adatbázis és a kód közötti kommunikáció JDBC (Java Database Connectivity) és az objektumok perzisztálása JPA (Java Persistence API) segítségével lett megvalósítva.

### Webalkalmazás architektúra

Az alkalmazás MVC dizájn mintára épül. Az alkalmazás több rétegre lett bontva, a View réteg felelős a megjelenítésért, a Controller réteg az irányításért, a Service réteg a logikai műveletek végrehajtásáért, a Repository az adatbázissal való kommunikációért (Entitásokon keresztül) és végül az DB (adatbázis) az adatok tárolásáért, ahogy a lentebbi 5. ábra - A webalkalmazás rétegei is illusztrálja.



5. ábra - A webalkalmazás rétegei

Ha megnyitjuk az alkalmazást, az első oldal a főoldal. Ezt két rész alkotja, az első a legjobb értékeléssel rendelkező könyvek top 3-as listája, a második a legjobb értékeléssel rendelkező szerzők top 3-as listája. A weboldal szerves részét képezi a navigációs sáv, amely mindig látható és segítségével navigálhatunk az oldalak között. Az applikáció egyszerűsített térképét, az alábbi ábra illusztrálja:



6. ábra - Egyszerüsített térkép az alkalmazás oldalairól

Azért egyszerüsített, mert ebben nincsenek benne azok a funkciók, amik bejelentekzés után vagy jogosultsághoz kötve látszanak. Néhány oldal láthatósága jogosultsághoz van kötve, ezért a fenti ábrát kiegészítve a teljes alkalmazás térképe így nézne ki:



7. ábra - Teljes oldaltérkép

### A webalkalmazás funkciói

A Home az alkalmazás fogadó oldala (index). Innen lehet navigálni a navigációs menü segítségével a fent látható oldalak bármelyikére, amelyre jogosultsága van a látogatónak.



8. ábra - Navigációs menü alapesetben



9. ábra - Navigációs menü User jogosultsággal (If Authenticated)



10. ábra - Navigációs menü Admin jogosultsággal (If Admin)

Alapvetően három jogosultsági szint van: látogató, User és Admin. Csak az adminnak van szerkesztési joga (író és könyv hozzáadása, szerkesztése), és ő kezeli a kapcsolati űrlapról beérkező leveleket is. A navigációs menü annak függvényében más, hogy a felhasználónak user vagy admin jogosultsága van, ahogyan azt a 8. ábra, 9. ábra, 10. ábra is illusztrálja.

A Books oldalon a rendszerben tárolt összes könyv listája található, az egyes könyvek borítóképével, szerzőjével, értékelésével, egy rövid, 150 karakteres leírással és az adott könyv részletes információihoz vezető linkel (Book Details Page, 7. ábra). Ezen az oldalon cím szerint lehet keresni a könyvekre. Ugyanezen az oldalon egy „Add New Book” címkével ellátott gomb is megjelenik az admin jogosultsággal rendelkező felhasználóknak, melyre ha rákattint, akkor elnavigálja az Add New Book oldalra, ahol új könyvet tud felvenni a rendszerbe.

A Book Details Page bővebb információt tartalmaz az adott könyvről, pontosabban a könyvről adott teljes leírás itt jelenik meg, valamint innen át lehet navigálni a könyv írójára/ költőjére is. Az alkalmazás továbbfejlesztett változatában itt lehetne a felhasználók hozzászólása a könyvekről.

Az Authors oldal a Books oldalhoz hasonlóan egy listát ad vissza, csak könyvek helyett a költők/ írók listáját tartalmazza, a nevükkel, születési évükkel, értékelésükkel és egy 150 karakteres leírással, valamint egy linkel, ami az adott költő részletes leírásához vezet (Author Details Page).

Az Author Details Page tartalmazza az adott író/költő nevét, születési dátumát, értékelését, és egy link- listát az általa szerkesztett könyvekről, ahová el tudunk navigálni akár. Az alkalmazás továbbfejlesztett változatában itt lehetne a felhasználók hozzászólása az íróról.

Contact us oldal. Ez az oldal látógató és user jogosultságú felhasználóknak egyaránt elérhető. Egy contact form-ot tartalmaz, amin keresztül kapcsolatba tudnak lépni az adminokkal, üzemeltetőkkel. A formon a következő három mező kitöltése kötelező: név, email cím, üzenet. Ha ezt elküldi a felhasználó, akkor az adminok a Contact Messages oldalon tudják elolvasni, törölni és/ vagy válaszolni rá. Ha admin jogosultságú felhasználó van bejelentkezve a rendszerbe, akkor a navigációs menüben alapból Contact Messages menüpont jelenik meg. Lásd 10. ábra.

Contact Messages oldal. A Contact us oldal helyett, ha admin jogosultságú felhasználó van bejelentkezve, ez az oldal jelenik meg. Itt az adminok, mint az előző bekezdésben említettem, tudják kezelni a Contact us oldalról érkező üzeneteket.

Sign in oldal. Ezen az oldalon kereszül van lehetőségük a felhasználóknek bejelentkezni a rendszerbe a helyes e-mail cím és jelszó párossal. Ez az oldal tartalmazza a regisztrációs (Registration) oldalra vezető linket is. Az alkalmazás továbbfejlesztett verziójában lehetőséget adhatnánk a felhasználónak új jelszó generálására, ha esetleg elfelejtené azt.

A előbb említett Registration oldalon van lehetősége a látogatóknak, hogy user jogosultságot szerezhessenek, ezáltal bekerüljenek a rendszerbe és tudjanak hozzászolni, értékelni könyveket, írókat/költőket. Az alkalmazás továbbfejlesztett változatában lehetne ellenőrizni azt is, hogy a felhasználó valóban emberi lény-e, illetve lehetne csak akkor aktiválni a profilt, ha a felhasználó e-mail címben megerősíti a szándékát.

A Users oldal. Ez a funkció csak a regisztrált felhasználók és adminok számára elérhető. Ez az oldal az összes regisztrált felhasználó listáját tartalmazza, azok e-mail címével, nevével, avatarjával és a linkel, ami az adott felhasználó bővebb információs oldalához vezet (User Details Page). Ezen az oldalon lehet email cím és név szerint keresni a felhasználók között (az adminok nem jelennek meg).

User Details Page. Ez az oldal az egyes regisztrált felhasználó adatait mutatja: név, email cím, regisztrációs dátum, rövid önéletrajz (amennyiben ad leírást magáról), és az avatart nagyobb méretben.

User Settings. Ez a funkció szintén csak regisztrált felhasználóknak elérhető, itt tudják állítani a saját magukra vonatkozott információkat, mint név, avatar, nem, rövid bemutatkozó szöveg. Úgy lehet ide jutni, ha a navigációs menüben a bejelentkezett felhasználó e-mail címére kattintunk. Ekkor megjelenik egy lenyíló menü és itt lehet kiválasztani a Settings menüpontot. Lásd az alábbi 11. ábra. Az alkalmazás továbbfejlesztett változatában itt állíthatnák a felhasználók, hogy mely információ jelenjen meg róluk a User Details oldalon.



11. ábra - User Settings és Sign out oldalak

Sign out link. Ez a User Settings menüpont alatt jelenik meg, egy link, melyre ha rákattintunk a Sign in oldalra vezet, egy visszajelző üzenettel, mely szerint a kilépés sikeres volt.

Dióhéjban ennyi az alkalmazás, egyszerű struktúra, de minden van benne, ami a tesztek és mérések elvégzéséhez szükséges. A továbbiakban a teszt keretrendszert fogom tárgyalni.

## A teszt keretrendszer

A már meglévő webalkalmazásra tehát lehet építeni egy teszt keretrendszert. Ez a keretrendszer a Seleniumot, Selenium WebDrivert használja a felhasználói felület automatizálására. A keretrendszer PageObject model dizájn mintát követ, hogy karbantartható és átlátható legyen. A Cucumber integráció azért szükséges, mert azt szerettem volna, ha a tesztek lépéseit megérti egy olyan ember is, aki nem feltétlenül ért a programozáshoz, ugyanis így a tesztek lépései természetes szövegként vannak leírva és a lépések maguk a tesztek dokumentációja is egyben. A JUnit pedig azért került a képbe, mert van tapasztalatom a használatában és mert így az automatizált tesztek futásáról visszajelzést kapunk nem csak a konzolon, hanem a JUnit ablakában is, ahol mutatja a tesztek futási idejét és sebességét is, sőt, az eredményeket elmenthetjük egy generált fájlba a későbbiekben.

### Selenium

### Selenium WebDriver

A WebDriver használata igencsak egyszerű. Származtatunk egy új böngésző drivert, és már használhatjuk is minden funkcióját. A Selenium rengeteg böngészőt támogat, ha szeretnénk használni, akkor le kell tölteni annak a driverét. Ezek ingyenesen elérhetőek, sőt, ha származtatunk egy böngészőt és futtatjuk a tesztet, de a driver nincs hozzá rendelve, akkor a konzolra kiírja az adott driver elérési linkjét. Ez önmagában nem elég a működéshez, mert meg kell mondani a kódnak, hogy hol keresse a Selenium ezt a drivert, például így:

System.*setProperty*("webdriver.chrome.driver","C:\\chromedriver.exe");

4. példakód - böngésző driver beállítása

Röviden ennyiből már használható és automatizálható egy teszt eset. Számomra ez a megoldás nem volt szimpatikus, mert nem szeretném szennyezni a kódot konfigurációval, ezért inkább úgy oldottam meg a driverek elérését, hogy letöltöttem őket, amelyekre szükségem van (chrome, internet explorer, firefox, opera), betettem egy mappába és a mappát beállítottam, mint környezeti változó. Íly módon elhagyható a fenti konfigurációs kódsor.

Ha rákeresünk az interneten a Selenium WebDriver használatára, legtöbb tutorial ennyiről szól. Beállítjuk a WebDrivert, elindítjuk a böngészőt, valami műveletet elvégzünk és aztán bezárjuk azt. Egy teszteset mellett ez rendben is van, de képzeljük el, hogy 10 tesztünk van. Ha minden esetet úgy írnánk meg, mint ahogy azt a 1. példakód –ban mutattam, az átláthatatlansághoz vezetne. Nem beszélve arról, hogy ha más böngészőben szeretnénk futtatni, akkor mind a 10 tesztben meg kell módosítani a böngésző származtatását. Képzeljük el ugyanezt a helyzetet 100 teszteset mellett. Na itt már szükség van absztrakcióra és alapvető objektum orientált programozói tudásra is.

Személy szerint úgy oldottam meg ezt a problémát, hogy létre hoztam egy Browser nevű osztályt, ahol származtattam a drivert és ez az osztály publikus, statikus függvényeket tartalmaz, melyek a driverrel működnek közre. Ezt az osztályt használom arra, hogy a webdriver függvényeit meghívjam, így központosítva van az, hogy milyen böngészőt használok, és az osztálynak annyi lesz a felelőssége, hogy a böngészővel interakcióba lépjen, semmi más.

Ezzel annyit értünk el, hogy a tesztekben nem szerepel a driver definíció. De ettől még nem lettünk megváltva, mert ha így írjuk a teszteket, a sok duplikált kódrészlet nem lesz megszűntetve. Itt jön képbe a Page Object model.

### Page Object Model

A Page Object Model egy nagyon elterjedt tervezési minta (design pattern), amit azért találták ki, hogy a tesztekhez írt kódot fenntarthatóvá, jobban átláthatóvá és könnyebben olvashatóvá tegye. Ezt a tervezési mintát ajánlják a [3], [5] és [12] forrásban is.

Egy weboldalt oldalak alkotnak, amik között tudunk valamilyen formában navigálni. Ha egy modern, úgynevezett „single page application”-ról van szó, azon is alkalmazható ez a minta, mert az alkalmazás ugyanúgy oldalakból áll, csak a betöltődésük dinamikus. A minta lényege (ahogy a nevéből is adódik), hogy az alkalmazást oldalak szerint lebontjuk, és minden oldalnak megfeleltetünk egy saját objektumot. Így minden oldalnak meglesznek a saját felelőssége, funkciói és saját függvényei, elkerülve ezzel a kavarodást és a kód duplikációját.

Az általam írt keretrendszerben is ezt a mintát használtam és a következőképp bontottam fel a webalkalmazás oldalait objektumokká:



12. ábra - A webalkalmazás oldalai megfeleltetve egy-egy Java objektumnak

Az egyes oldalak objektumában aztán létrehoztam azokat a függvényeket, melyek specifikusak az adott oldalra. Például: LoginPage.java-ban van egy olyan függvény, hogy „clickOnRegistrationLink()”. Ez a függvény csak a LoginPage-re jellemző, ugyanis csak ezen az oldalon található a regisztrációs oldalra vezető link. A többi osztályban hasonlóan, az adott oldalra vonatkoztatott egyedi függvények vannak létrehozva.

Ami minden oldalban közös, azt megpróbáltam kiemelni egy Page nevű absztrakt osztályba, amit minden oldal kiterjeszt, kivéve a MenuBar.java. Ezt az osztályt azért hoztam létre, hogy legyen egy osztály, aminek a felelősség a navigációs menüvel való interakció és tartalmaz minden olyan függvényt, amit a navigációs menü irányításához használhatunk. Az oldalaknak megfeleltetett java osztályokról a Teszt keretrendszer arhitektúra címszó alatt fogok bővebben szót ejteni.

### Cucumber

Mint ahogy azt a [13] forrás gyönyörűen leírja, egy szoftver az nem más, mint egy jó öltlet megvalósítása. Legtöbb esetben az ötlet gazdája nem rendelkezik elegendő tudással ahhoz, hogy megvalósítsa önerőből az ötletét, ezért az ötletét kell tudja kommunikálni egy olyan ember felé vagy egy egész csapat felé, aki/akik segít neki a megvalósításban. Ezért kulcsfontosságú szerepet játszik a kommunikáció a két fél között, hogy a kivitelezés helyes legyen. Itt jön képbe a BDD és a Cucumber, mert általa maga a product owner is írhatja az alkalmazást és annak lépéseit.

A Cucumber egy BDD (Behavior Driven Development) megvalósítás, mely lehetővé teszi, hogy természetes nyelven (Gherkin szintaxis) fogalmazzuk meg a tesztjeinket. A Cucumber hivatalos oldala [14] úgy foglalja össze két szóban a keretrendszert, mint „Executable Specifications” vagyis Futtatható Specifikációk. Ugyanis a Cucumber által leírt tesztek nem csak a azok lépéseit tartalmazzák, hanem, mivel bárki számára érthető szintaxist használ (Gherkin), ezért a fájlok a tesztek specifikációi is lesznek egyben.

A BDD, vagyis Behavior Driven Development kiegészítése a TDD-nek (Test Driven Development).



13. ábra - BDD, ahogy TDD-re épül Forrás: [15]

Ahogy a fenti ábra mutatja, a TDD fejlesztési folyamata úgy épül fel, hogy írunk egy tesztet, ami elhasal, aztán megírjuk a szükséges kódot ahhoz, hogy a teszt épp sikeres legyen, refaktorálunk majd újra kezdjük a folyamatot. A probléma ezzel az, hogy itt a teszt van a középpontban és elfelejtődhet a kérdés, hogy valóban szükség van-e arra a tesztre, kódra, funkcióra és vajon hogyan viselkedhet a felhasználó.

A különbség BDD és TDD között inkább a fókuszban rejlik. A BDD nem a tesztre összpontosít, hanem a felhasználó viselkedésére. A Cucumber egy implementációja ennek, mely Gherkin nyelvet használ arra, hogy leírjuk a felhasználó viselkedését és a rendszer válaszát az adott viselkedésre. Ezt a viselkedést Cucumber esetén úgynevezett feature fájlokba írjuk. Erre egy egyszerű példa:

Feature: Home Page Test

I want to test if the home page is working correctly.

Scenario: Can Go To Homepage

Given A web browser

When I enter the webapplication address to the address bar

Then I should be redirected to MyLibrary homepage

5. példakód - Cucumber feature

A fenti kódrészben a Feature, Scenario, Given, When, Then mind a Gherkin nyelv kulcsszavai, melyek azért szükségesek, hogy általuk a Cucumber össze tudja kötni a leírást a kóddal. A kulcsszavakat leszámítva teljesen hétköznapi nyelven van megfogalmazva mindaz, aminek történnie kellene.

Értelmezzük a 5. példakódot:

* A Feature kulcsszó egy szoftver funkció magas szintű leírása. Ez a kódban nem játszik szerepet, inkább a dokumentációban fontos, hogy tudjuk, milyen funkcióval foglalkozik az adott fájl.
* A Scenario kulcsszó egy konkrét eset címe, leírása
* A Given, When, Then a Scenario lépéseit (Step) írják le
  + A Given kulcsszó segítségével definiáljuk a kontextust
  + A When kulcsszó segítségével leírunk egy cselekvést vagy eseményt
  + A Then segítségével leírjuk, hogy mi az elvárt eredmény

A fenti kulcsszavak mellett a projektemben használtam még a következőket:

* Scenario Outline – arra használjuk, hogy ugyanazt a Scenariot többször futtassuk más-más bemeneti értékekkel
* Examples kulcsszó segítségével tudjuk meghatározni a Scenario Outline bementei értékeit táblázatos formában
* And, But – az olvashatóság érdekében kötőszóként használjuk, mint lépés leíró Given, When, Then mellett. lásd a lenti példát:

A lenti példa teljes mértékben a Cucumber hivatalos oldaláról [14] származik és nagyon jól szemlélteti az And és But kulcsszavak jelentőségét:

Amikor több kulcsszó ismétli egymást, így is írhatnánk:

Example: Multiple Givens

Given one thing

Given another thing

Given yet another thing

When I open my eyes

Then I should see something

Then I shouldn't see something else

6. példakód - And, But nélkül Forrás: https://cucumber.io/docs/gherkin/reference/

Vagy olvashatóbbá tehetjük a leírást, ha And és But kulcsszavakat használunk

Example: Multiple Givens

Given one thing

And another thing

And yet another thing

When I open my eyes

Then I should see something

But I shouldn't see something else

7. példakód - And, But használatával Forrás: https://cucumber.io/docs/gherkin/reference/

A kulcsszavak teljes listáját a Cucumber hivatalos oldalának [14] dokumentációjánál lehet elolvasni, ahol nagyon jól leírják és példákkal szemléltetik az egyes Gherkin szavak használatát.

A Cucumber úgy köti össze a java kódot a neki megfelelő feature fájlokkal, és annak lépéseivel, hogy a java függvény fölött megjelenik majd a szöveg, ami az adott lépést tartalmazza. Úgy is mondhatnám, hogy a Given, When, Then, And/But kulcsszavakkal meghatározott mondatok, kódban egy-egy függvény lesznek. Az 5. példakód java függvényei így néznek ki:

@Given("A web browser")

**public** **void** openTheBrowser(){…}

@When("I enter the webapplication address to the address bar")

**public** **void** iEnterTheApplicationUrl(){…}

@Then("I should be redirected to MyLibrary homepage")

**public** **void** iShouldBeRedirectedToMyLibraryHomepage(){…}

8. példakód - Cucumber feature fájlok és java függvények kapcsolata

A fenti kulcsszavak kombinációjából sikerült felépítenem a keretrendszerem tesztjeit. A projektem 11 feature fájlt tartalmaz, melyekben 20 Scenario és 6 Scenario Outline van. Összesen 162 lépést (Given, When, Then, And/But) végez el a rendszer.

Egy nagyon pozitív dolog, hogy ha megírtunk egy lépést (Step), és az ahhoz tartozó kódot is, akkor a Cucumber felismeri, hogy létezik és nem kell újra megírni más feature fájlokban, ha ugyanazt a leírást adjuk meg mindkettőnél.

Összefoglalva: a TDD-vel ellentétben itt nem az a lényeg, hogy a teszt átmenjen, hanem az, hogy a kód azt csinálja, ami a Feature-ben le van írva. Ahogy Catalin Tudose említi a kurzus [16] bevezetőjében: a BDD segít abban, hogy ne csak helyesen csináljuk a dolgokat, hanem hogy a jó dolgokat csináljuk (teszteljük). És az, hogy mi a jó, ezt az ötlet gazdája tudja igazán leírni és elmagyarázni hétköznapi nyelven úgy, hogy érthető legyen a fejlesztők számára is. Ezért lehet nagy érték a Cucumber egy projektben.

### JUnit-al való integráció

A JUnit egy egyszerű teszt keretrendszer javahoz, mely segít a tesztek írásában, futtatásában és az egyes eredmények összehasonlításában (Assertion). Ennyi az egész. Pont azért lett integrálva a keretrendszerbe, hogy ezeket a feladatokat ellássa. Alapszintű jelentést is visszaad a tesztek futási idejével Feature és Scenario szintre lebontva, valamint arról, hogy az összes teszt mennyi ideig futott és hogy hány volt sikeres illetve sikertelen.

### Teszt keretrendszer arhitektúra

Mint ahogy a fentiekben kiderült, a teszt keretrendszer alapjait a Selenium WebDriver alkotja, általa automatizálható lett a böngésző, a tesztek kódjának olvashatóságát, karbantartahóságát a Page Object Model minta alkalmazásával javítottam, a tesztek tartalmát és annak nem technikai személyek által való megértését a Cucumber használata teszi lehetővé és a tesztek futtatásáért, visszajelzéséért a JUnit felelős. Az alábbiakban arról szeretnék beszélni, hogy ezek a technológiák hogyan alkotnak egy nagy teszt automatizálási keretrendszert.

A projekt egy Maven projekt. Nem lenne kötelező Mavent használni, mert a külső könyvtárakat össze lehetne gyűjteni nélküle is. Személy szerint azért részesítem előnyben a Maven projektet, mert egyszerű dependencia és build kezelést tesz lehetővé a használata.

Nagyvonalakban a rendszer így néz ki:



14. ábra - A teszt keretrendszer rétegei

A fenti 14. ábra mutatja a teszt keretrendszerem rétegeit. Lefgelül a JUnit van, hiszen annak segítségével futtatom a teszteket. Alatta a Cucumber, ami tartalmazza a már említett Feature fájlokat, amikben le vannak írva Gherkin szintaxissal a teszt lépések, funkciónként. A következő réteg már a kód rétege, ez több mindent magába foglal, ugye ott a Selenium WebDriver és a PageObject model, ezt később tárgyalom bővebben, az egyszerűség kedvéért az ábrán csak mint Selenium Web Driver szerepel. Ez alatt már csak a konkrét böngésző driver szintje van, ami irányítja a böngészőt, ezzel nekünk csak annyi a dolgunk, hogy a böngésző verziójának megfelelő driver legyen betöltve a rendszerbe.

Ebben a sorrendben bontsuk ki a rétegeket és nézzük, hogy mit hogyan valósítottam meg:

A JUnitot nem módosítottam, mint ahogy fennebb említettem, csak arra használtam, hogy a Cucumber teszteket lefuttassam és az eredményeit megkaphassam, ami kódot kellett írnom ehhez, körülbelül 10 sor.



15. ábra - JUnit TestRunner.java

Mint ahogy a 15. ábra mutatja, létre hoztam egy TestRunner.java osztályt. Ebben van két számunkra érdekes annotáció:

* @RunWith()
  + a fenti annotáció monjda meg a JUnitnak, hogy használjon Cucumbert a futtatáshoz
* @CucumberOptions()
  + mint ahogyan neve is sugallja, ezzel az annotációval tudunk további beállításokat adni a Cucumber számára
  + features="src/test/resources/features" – ezzel a sorral mondom meg a Cucumbernek, hogy hol keresse a feature fájljaimat. Nem lenne erre szükség, ha a feature fájlok a TestRunner osztály mellett lennének, de az átláthatóság érdekében át tettem őket a test/resources/features mappa alá
  + glue= {"stepdefinitions"} – ezzel a paranccsal jelzem a Cucumbernek, hogy a feature fájloknak megfelelő java függvények melyik package-ben vannak
  + plugin = {"json:target/cucumber-report/cucumber.json"} – ezzel a sorral mondom meg, hogy a junit eredményeit mentse ki a target/cucumber-report mappába, cucumber néven és JSON formátumú fájlba
* A @BeforeClass annotációval jelzem a JUnitnak, hogy ennek a függvénynek azelőtt kell lefutnia egyszer, mielőtt a teszteket futtatni kezdené. A mi esetünkben ezen a ponton inicializáljuk a böngésző drivert és nyissuk meg a böngészőablakot.
* Az @AfterClass annotációval jelölt függvény akkor fut le, amikor minden teszt befejezte a futását. Esetünkben arra haszáljuk, hogy bezárjuk a böngészőablakot.

Ennyi a JUnit szintje.

A következő szint a Cucumber szint. Nézzük, hogy hogyan alkottam meg a feature fájlok struktúráját:

A Cucumber feature fájlokat a projekt alatt az src/test/resources/features mappába hoztam létre. Mivel a webalkalmazásnak három jogosultság szintje van, ezért itt is igyekeztem a feature fájlokat eszerint csoportosítani, vagyis az admin jogosultsággal rendelkező felhasználók oldalainak funkcióit az admin mappa alatti feature fájlokkal fedem le, a user jogosultságú felhasználók oldalait az authenticated mappa fedezi, a common mappa pedig azokat a feature fájlokat tartalmazza, melyek megjelennek a látogatóknak. Lásd a lenti 16. ábra.



16. ábra - Cucumber Feature fájlok

Ahogyan az előbbi ábrán is látszik, a feature fájlok az egyes oldalaknak vannak megfeleltetve. A webalkalmazásom összetettségéből kifolyólag elegendő volt oldal szintre lebontani a funkciókat, ám ha egy bonyolultabb rendszerről lenne szó, akkor érdemes lehet tovább bontani a tesztelendő funkciókat (feature). Az általam létrehozott feature fájlok nem fedik le az alkalmazás 100%-át. Úgy gondolom, hogy fölösleges lenne minden apró funkciót külön tesztelni. A feature fájljaim a fő funkciók tesztleírását tartalmazzák, melyek sorrendben:

1. Az admin mappa feature fájlai

* AddNewAuthorPage.feature
  + Scenario1: Admin joggal való bejelentkezés után megjelenik-e az új költő/író hozzáadása funció?
  + Scenario2: Admin joggal való bejelentkezés után működik-e az új költő/író hozzáadása funkció?
* AddNewBookPage.feature
  + Scenario1: Admin joggal való bejelentkezés után megjelenik-e az új könyv hozzáadása funció?
  + Scenario2: Admin joggal való bejelentkezés után működik-e az új könyv hozzáadása funkció?
* AdminDashboardPage.feature
  + Scenario1: Admin joggal való bejelentkezés után megjelenik-e az Admin Dashboard funkció a navigációs menün és kattintásra oda navigál-e?
* ContactMessagesPage.feature
  + Scenario1: Admin joggal való bejelentkezés után megjelenik-e az Contact Messages funkció és kattintásra oda navigál-e?
  + Scenario2: Admin joggal való bejelentkezés után megjelennek-e az üzenetek és ha az első üzent Open Message gombjára kattintunk, megjelenik-e az az üzenet teljes egészében?
  + Scenario3: Admin joggal való bejelentkezés után, ha az első bejövő üzenet nézeténél ha a Delete gombra kattintunk, kitörlődik-e az üzenet?

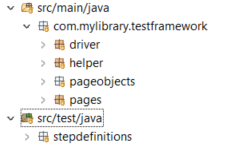
1. Az authenticated mappa feature fájlai

* UsersPage.feature
  + Scenario1: User joggal való bejelentkezés után megjelenik-e a Users funkció és kattintásra oda navigál-e?
  + Scenario2: User joggal való bejelentkezés után a Users funkción tudok-e keresni egy felhasználóra e-mail cím szerint?
  + Scenario3: User joggal való bejelentkezés után a Users funkción tudok-e keresni egy felhasználóra felhasználónév szerint?
  + Scenario4: User joggal való bejelentkezés után a Users funkción tudok-e kiválasztani egy felhasználót, hogy megnézzem az adatait?

1. A common mappa feature fájlai
   * AuthorsPage.feature

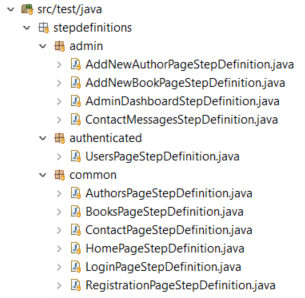
* Scenario1: Ha kiválasztom az első költőt/írót, akkor az ő oldalára írányít-e?
* Scenario2: Tudok keresni egy íróra/költőre a neve alapján? Megjelenik-e a találati listában?
  + BooksPage.feature
* Scenario1: Ha kiválasztom az első könyvet, akkor az első könyv adatainak oldalára írányít-e?
* Scenario2: Tudok keresni egy könyvre a címe alapján és megjelenik a találati listában?
  + ContactPage.feature
* Scenario1: Ha kiválasztom Contact us menüpontot a navigációs listában, a Contact us oldalra navigál-e?
* Scenario2: Ha kitöltöm a contact form-ot és rákattintok a Send Message gonbra, akkor az üzenet sikeresen el lesz-e küldve?
  + HomePage.feature
* Scenario1: Ha beírom az alkalmazás címét a webböngészőbe, betöltődik-e az alkalmazás HomePage oldala?
* Scenario2: Ha a HomePage oldalon a leginkább értékelt könyvre kattintok, annak a könyvnek az adatlapjára navigál-e?
* Scenario3: Ha a HomePage oldalon a leginkább értékelt iróra/költőre kattintok, annak az írónak/költőnek az adatlapjára navigál-e?
  + LoginPage.feature
* Scenario1: Ha beírom a bejelentekzési oldalra a hibás jelszót vagy email címet, dob-e hibaüzenetet?
* Scenario2: Ha a beírom a bejelentekzési oldalra a helyes jelszót és email címet, sikeresen megtörténik-e a bejelentkezés?
  + RegistrationPage.feature
* Scenario1: Oda tudok-e navigálni a regisztrációs oldalra a bejelentkező oldalról?
* Scenario2: Tudok-e regisztrálni egy új felhasználót a rendszerbe?
* Scenario3: Tudok-e kétszer regisztrálni ugyanazzal a felhasználóval?

Mint már említettem, a feature fájlok lépései a tartalmuk által vannak összeköttetésben azzal a java függvénnyel, mely végre hajtja őket, lásd 5. példakód és 8. példakód. Ezek a java függvények a következő szintben vannak, a Selenium WebDriver szintjén. Ez a legösszetettebb szint. Két fő package alkotja: src/test/java/stepdefinitions - itt vannak a feature fájlokkal összeköttetésben álló java függvények és src/main/java/com.mylibrary.testframework – itt pedig a Page Objektek és a többi java osztályok vannak.



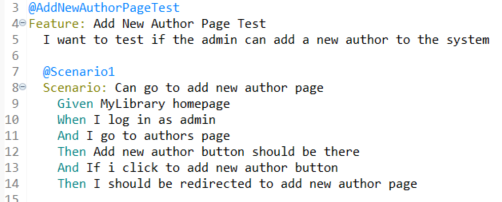
17. ábra - Selenium WebDriver könyvtárai

Azért választottam külön a tesztek lépéseit a src/test/java/stepdefinitions packagebe, mert szerintem logikailag közelebb áll a tesztekhez, mint a WebDriverhez, mivel a feature fájlok tartalmát „tükrözi”, csak ki van egészítve java kóddal. Lássuk először őket:

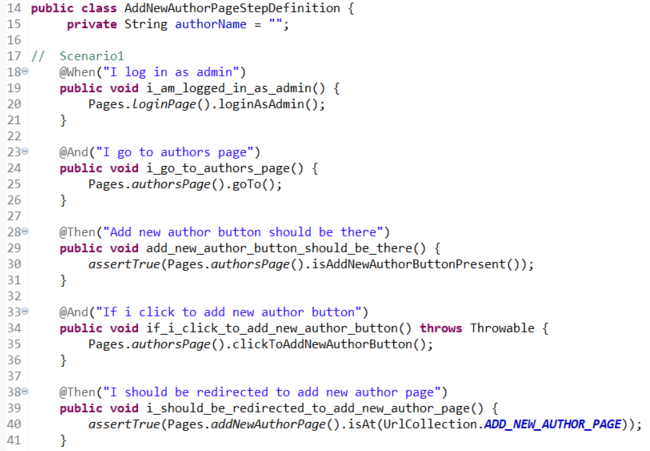


18. ábra - Feature fájlok java kódjai

Ha összevetjük a 16. ábra és 18. ábra tartalmát, felfedezhetjük, hogy a stepdefinitions a feature fájlok teljes mása, csak itt nem Gherkin nyelv van, hanem java kód. Eddig a menedzsment tagjai is átlátták a folyamatot, ám ettől kezdve a fejlesztőké a szerep.



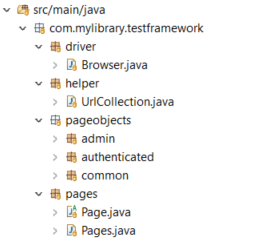
19. ábra - AddNewAuthor.feature - Scenario1



20. ábra - AddNewAuthorStepDefinition - Scenario1

A fenti 19. ábra az AddNewAuthor.feature első Scenarioját tartalmazza Gherkin nyelven fogalmazva, míg a 20. ábra az ennek megfelelő java kódot szemlélteti. Látható, hogy a függvénynevek fölött megjelenik az a sor, ami a feature fájlban meg van fogalmazva, csak néhány módosításon esett át: annotációt kaptak a kulcsszavak és zárójelben, idézőjelek között szerepel a leírás. Észre vehetjük, hogy a Given kulcsszónak nincs párja a java definíciók között. Ez azért van, mert azzal a szöveggel már definiálva volt az a lépés és az annak megfelelő java kód a ContactPage.feature illetve ContactPageStepDefinition.java fájlokban, így ha legközelebb használni szeretném, elég leírni a feature fájlban, a java megfelelőjét kikeresi magától a Cucumber.

A 20. ábra függvényeinek tartalma többnyire egy sor. Ide is írhatnánk már a WebDriver dolgait, de akkor kevésbé lenne átlátható. Hogy mi van a függvényekben, ahhoz bele kell nézzünk az src/main/java/com.mylibrary.testframework package alá:



21. ábra - Selenium WebDriver implementációja

Itt már több dolgot látunk, nézzük szerre, hogy miket tartalmaz:

* driver package
  + Browser.java – ez az osztály felelős a böngésző irányításáért. Ezért is kapta a Browser nevet, valamint ebben az osztályban van példányosítva a böngésző drivere is. Minden függvénye statikus és olyan függvényeket tartalmaz, mint például goTo(String url), ami arra szolgál, hogy a paraméterként kapott url címre navigál, vagy a getTitle(), mely visszatéríti az aktuális oldal title tagjei között szerepelő szöveget, meg hasonló, Selenium WebDriver cím alatt tárgyalt függvények.
* helper package
  + UrlCollection.java – ez az osztály az alkalmazás url gyűjteményét tartalmazza, minden url cím egy public static final String ként szerepel. Például:

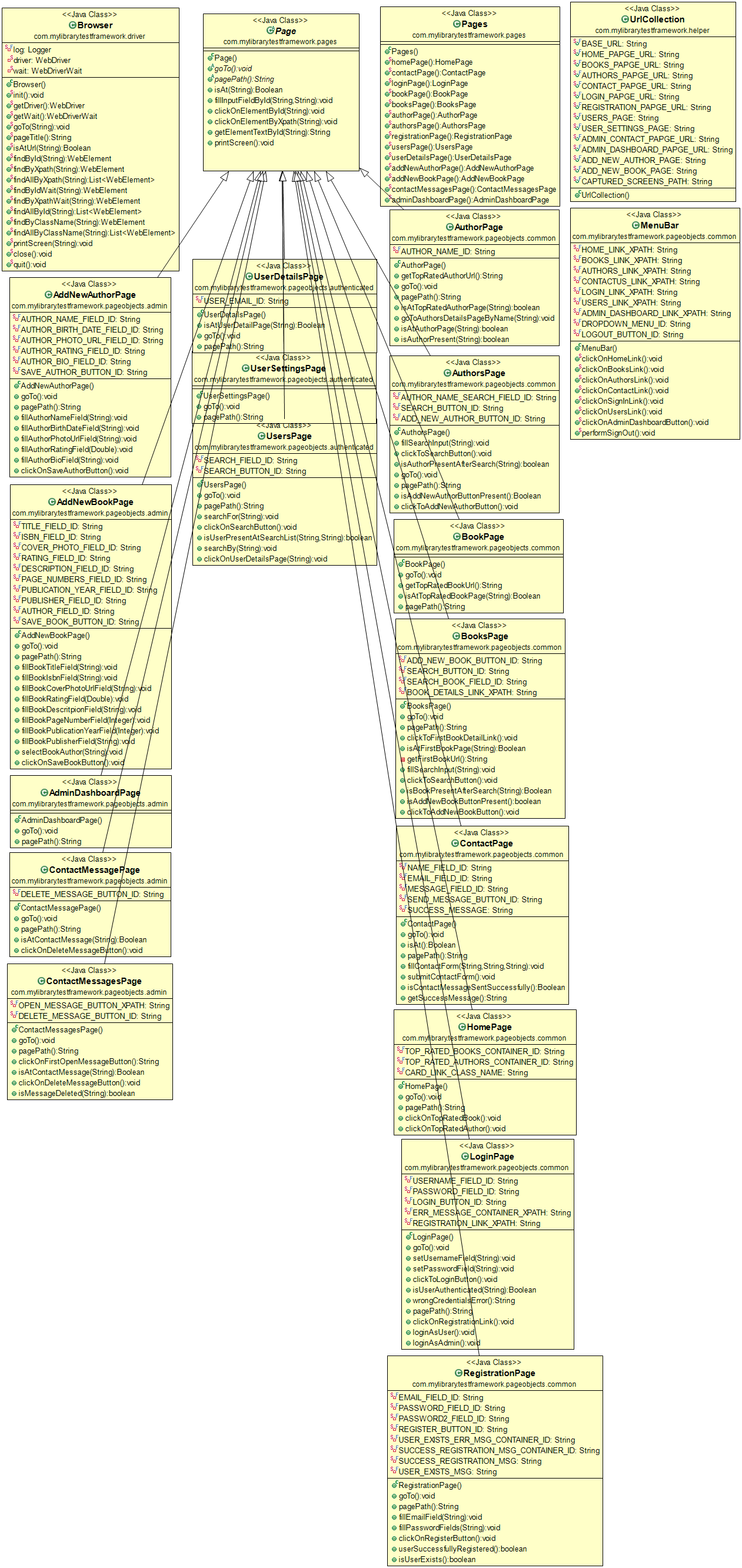
**public** **static** **final** String ***BASE\_URL*** = "http://localhost:8080/";

**public** **static** **final** String ***BOOKS\_PAPGE\_URL*** = ***BASE\_URL*** + "books";

9. példakód - UrlCollection példa

* pages package
  + Page.java – ez egy absztrakt osztály, melybe megpróbáltam kiemelni minden olyan közös tényezőt, amely minden oldalban közös, például, hogy minden oldalnak van egy url címe, minden oldalról lehet készíteni képernyőképet, minden oldalnak van valamilyen WebElementje, melyre rá lehet kattintani vagy input mezője, melyet ki lehet tölteni.
  + Pages.java – az osztály csak publikus statikus függvényeket tartalmaz, melyek vissza adják az egyes pageobjecteket. Azért hoztam létre, hogy mindig ugyanazt a pageobject-et használjam az alkalmazásban, így központosítva az objektumok létrehozását. Ezt a fajta központosítási mintát Factory Pattern – nek nevezik.
* pageobjects package – azért hagytam ezt a legvégére, mert ez a package tartalmazza az alkalmazás page objectjeit, vagyis a webalkalmazás minden oldalának megfeleltetett java objektumot, lásd Page Object Model címsort. A csoportosítási logika ugyanaz, mint a feature fájlok esetében: a jogosultsági szint.
  + admin package
    - AddNewAuthorPage.java
    - AddNewBookPage.java
    - AdminDashboardPage.java
    - ContactMessagePage.java
    - ContactMessagesPage.java
  + authenticated package
    - UserDetailPage.java
    - UserSettingsPage.java
    - UsersPage.java
  + common package
    - AuthorPage.java
    - AuthorsPage.java
    - BookPage.java
    - BooksPage.java
    - ContactPage.java
    - HomePage.java
    - LoginPage.java
    - RegistrationPage.java
    - MenuBar.java

Ha UML diagramként ábrázoljuk, így viszonylnak egymáshoz az osztályok:

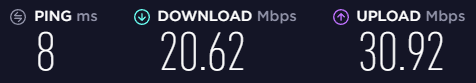


22. ábra - WebDriver réteg UML diagramja

Mint a fenti ábra mutatja, minden \*Page.java a Page.java leszármazottja, ezáltal a gyakori függvényeket megöröklik. Összefoglalva a fenti rétegek alkotják a teszt keretrendszerem egészét.

# Mérések

A méréseket egy Lenovo Y700 laptopon végeztem, 4 magos Intel Core i7 6700HQ processzorral, 8Gb RAM-al, 4Gb-os NVIDIA GeForce GTX 960M videókártyával, 256Gb SSD-vel. Az internet sebesség, melyen végeztem a méréseket Digi Fiberlink 100-as csomag, mely a távoli szerverrel való tesztelés elvégzése idején a következő paraméterekkel, bírt:



23. ábra - Internet sebessége Forrás: [www.speedtest.net](http://www.speedtest.net)

A manuális tesztek eredményei 5 ember átlagolt idejét tartalmazzák, olyan emberek, akik számára nem idegen a webalkalmazás, vagyis nem kell kitalálják, hogy hová kell kattintsanak az egyes funkciók eléréséhez.

A keretrendszer által futtatott mérések eredményei 100 alkalommal való futtatás átlagát mutatják. A teszteket futtattam naiv és optimalizált implementációként is (bővebben a naiv és optimalizált tesztek közötti különbségről beszélek a későbbiekben), a lennebb felsorolt böngészőkre bontva. Ezt úgy oldottam meg, hogy írtam egy szkriptet, ami ciklusban futtatja a keretrendszert. Minden futási alkalommal generáltattam a cucumber.json fájlt az eredményekkel (lásd 15. ábra). Ez után, a kapott eredmény fájlt átdolgoztam egy Cluecumber nevű, Trivago cég által fejlesztett, nyílt forráskódú keretrendszer segítségével (lásd forrás [17]). Ez a keretrendszer html oldalakat generál a fennebb említett cucumber.json fájlból, amit majd böngésző segítségével meg lehet nézni, bár én a keretrendszert arra használtam, hogy a cucumber.json fájlból kinyerjem a számomra fontos inforámciókat. Lévén, hogy a Cluecumberben már implementálva van a beolvasási és mappelési logika, nem kellett megírnom ezt önerőből. Amit a cucumber fájlból kinyertem, az pedig: Feature neve, Scenario neve, Feature fájl futásideje másodpercben, az egyes Scenariok futási ideje másodpercben és a teszteset lefutásának teljes ideje. Ezeket elmentettem egy .csv kiterjesztésű fájlba. Amikor lefutott a 100 teszt eset, az 100 darab .csv fájlt jelentett. Ezeket a fájlokat aztán beolvastam egy általam írt programmal, ami kiszűri és átlagolja a 100 fájlból érkező eredményeket, majd az így keletkezett átlagolt eredményeket kimentettem egy ugyancsak .csv fájlba.

A teszteket lefuttattam Google Chrome (Verzió 75.0.3770.100), Mozilla Firefox (Verzió 67.0.4) és Opera (Verzió 60.0.3255.170) böngészőkben is, úgy, hogy a webalkalmazás helyi szerveren futott (localhost). A mérés kezdete előtt újraindítottam a webalkalmazást, ezáltal az adatbázis mindig kezdeti állapotba került. Internet Explorerrel akadtak gondjaim futtatáskor: nem elemezte helyesen a DOM-ot és nem talált sok elemet, valamint amikor input mezőbe kellett volna írni, akkor 1 karakter begépelése körülbelül 30 másodpercet vett igénybe, pedig a helyes verziójú WebDrivert használtam az adott Internet Explorerhez, így az IE kimaradt a buliból.

A mérésekhez az időt vettem alapul, hogy megtudjam, mennyi belefektetett idő után térülhet meg a munka.

Elsőként a tesztek fejlesztési idejét mértem meg. A fejlesztési időt attól a pillanattól kezdtem mérni, amikor létrehoztam az üres java osztályt, a feature fájlok már meg voltak írva a tesztekkel, és akkor fejeztem be, amikor a teszt sikeresen lefutott. Egy olyan embernek, aki komfortosan mozog javaban, a következőképp alakult az egyes tesztek fejlesztési ideje:

Mint látható, a legegyszerűbb tesztet megírni egy olyannak, aki ismeri a webalkalmazást, körülbelül 10 percbe telik. Ez az egyszerű eset az, amikor azt tesztelem, hogy az adott oldal betöltődik-e? A bonyolultabb esetek 50 és 60 perc munkát vesznek igénybe, például hogy lehet-e hozzáadni új könyvet a rendszerhez, ha admin jogosultsággal jelentkezünk be?

Most, hogy tudjuk, mennyi időt kell áldoztunk a tesztek létrehozására az üres osztálytól a futtathatóig, lássuk ezek mennyi ideig futnak és esetleg mennyi időt spórolhatnak meg számunkra.

Úgy gondolom, hogy manuális tesztelés esetén a böngésző nem, vagy elhanyagolható mértékben befolyásloja az eredményeket. Mint látható, egy olyan felhasználó számára, aki ismeri a rendszert, a legegyszerűbb feladat elvégzése, ami abból áll, hogy beöltődik-e egy adott oldal és a jó oldal töltődik-e be, körülbelül 3-5 másodpercet vesz igénybe, míg a leg időigényesebb feladatok, mint például egy új könyv hozzáadása a rendszerhez, 2 percet is igénybe vehet. És ahhoz, hogy mindent leteszteljen, 12 percbe és 38 másodpercbe telik.

Most lássuk az automatizált teszt keretrendszerünk ugyanezt a feladatot mennyi idő alatt végzi el:

A keretrendszerünk meglehetősen gyorsan végigteszteli az alkalmazást. A legegyszerűbb feladat 0,1 másodpercet vett igénybe számára, míg a legbonyolultabb is csak maximum 2,7 másodpercet. Így összesen Google Chrome esetén 32.6 másodperc, Opera esetén 27,4 másodperc és Mozilla Firefox esetén csak 25.9 másodperc alatt végigzongorázza a teszteket a rendszer. A Firefox teljesített a legjobban, de ez annak is betudható, hogy míg a többiek animációval töltenek ki egy input mezőt (kvázi betűnként írják be az értéket), addig a Firefox csak beilleszti a szöveget úgy, ahogy van. Lássuk feature szerint lebontva, hogyan teljesítettek a böngészők:

Látható, hogy ha automatizáljuk a teszteket, az a fenti esetben megközelítőleg 16x (ha a leglassabb eredményt nézzük) és 20x (ha a legjobb eredményt nézzük) gyorsabban elvégezhető a tesztelési folyamat, mint kézzel. Elég imponáló, de lehet ennél jobban is csinálni. Azért mondom ezt, mert a fenti méréseket naiv teszteken végeztem. Naiv teszt alatt azt értem, hogy nincsenek optimalizálva, újra átgondolva, ezért sok ismétlődést tartalmaznak. Például: Bejelentkezek Admin joggal, elvégzem a dolgomat, kijelentkezek, majd újra bejelentkezek admin joggal, elvégzem a következő dolgot, majd kijelentkezek. Ez redundanciához vezet és lehetne optimalizálni úgy, hogy bejelentkezek, elvégzem az összes admin jogosultsághoz köthető dolgot, majd kijelentkezek, mert nincs értelme a bejelentkezési folyamatot ismételgetni, ha egyszer biztosan működik. Ha eképp optimalizáljuk a teszteket, vajon milyen eredményeket kapunk?

Az előző naiv tesztek 33 Scenariot és 162 lépést tartalmaztak. Így, hogy kivettük a duplikált bejelentkezés – kijelentkezés folyamatot, spóroltunk 10 lépést, vagyis az új tesztek ugyancsak 33 Scenariot tartalmaznak, viszont itt már csak 152 lépést hajtunk végre. Lássuk, hogy ez hogyan befolyásolja a futásidőket:

Mint látható, nagyjából 15 másodpercet nyerünk azzal, hogy a felhasználó nem kell bejelentkezzen többször.

Azzal, hogy nem kell duplikált lépéseket végezzünk, nyertünk legalább 91 másodpercet. Tételezzük fel, hogy a felhasználó elrontja a jelszavát, akkor ennél többet is nyerhetünk.

És ha ugyanezt a rendszerrel végeztetjük, akkor a következőképp alakul a diagram:

Azzal, hogy kicsit újra gondoltuk a teszt folyamatot és kihagytuk a duplikált lépéseket, láthatólag sikerült sprólolni az idővel. Ha összevetjük a manuális és az optimalizált tesztek futási idejét, a következő diagramot kapjuk:

A fenti tesztek mindenikét a lokális szerveren futó alkalmazáson végeztem. Ebben az esetben az internet sebességének elenyésző szerepe van. Nézzük, mennyiben befolyásolja az eredményeket, ha egy távoli alkalmazáson futtatjuk az alkalmazást és úgy végezzük el a teszteket. A távoli alkalmazás futtatásához a Herokut választottam (<https://www.heroku.com/>). Azért választottam ezt, mert ingyenes szervert biztosít, ahová deployolhatom a webapplikációmat egyszerűen és gyorsan, miután bárki számára elérhető lesz. A Heroku-n futó alkalmazás url címe a következő: <https://my-library-example.herokuapp.com/> .

És akkor lássuk az eredményeket:

Várható volt, hogy a távoli szerveren való futtatás tovább fog tartani, és így is lett. Itt is a Firefox viszi a zászlót, aztán az Opera és végül a Chrome, viszont itt már nem olyan nagyok az eredménybeli különbségek, mint amikor lokális szerveren futott a webes applikáció.

# Következtetések és tapasztalatok

A fenti mérések alapján elmondhatjuk, hogy egy alkalmazás autómatizált tesztjeit megírni viszonylag sok időbe telik, még akkor is, ha csak egy egyszerű alkalmazásról van szó. Ha az általam fejlesztett MyLibrary alkalmazást vesszük alapul, annak a tesztjeit megírni összesen 774 percbe telt, vagyis 12 órába és 54 percbe. Ezek a naiv tesztek voltak. Hogy végignézzük és optimalizáljuk a teszteket, további 15 percet vett igénybe. Ez az idő tartalmazza a fejlesztés során fellépő hibákat és azok kiküszöböléseit, amíg működő teszteteket nem kaptam. És ez csak maguk a tesztek megírásának ideje. A keretrendszer kifejlesztésére fordított idő ennél sokkal több. Ennél az egyszerű web alkalmazásnál is több, mint egy hétbe telt. Ez esetben egy vállalati alkalmazásra keretrendszert fejleszteni hónapokba, sőt évekbe is telhet.

A fejlesztés során fellépő leggyakoribb probléma az volt, hogy valamiért nem találta a WebDriver azt az elemet, amit szerettem volna, ezért NoSouchElementException-nal leállt a program. A másik leggyakoribb probléma a böngészők közötti különbségek voltak. Nagyon jól hangzik a Seleniumtól, hogy csak böngésző drivert kell cserélni és a tesztek ugyanúgy működnek, de ez csak részben igaz. Mindenik böngésző másképp értelmezi az oldalakat, ebből akadnak hibák jócskán. Például fent említettem, hogy az Internet Explorernek egy 5 karakterből álló szót 2 és fél percbe telt beírni egy input mezőbe. Vagy a Chrome nagyon jól értelmezte az XPath alapján történő keresést, ám a többi böngésző itt nem remekelt annyira. Így, hogy a webes alkalmazás forráskódjához hozzáférésem volt, sokat egyszerüsített a helyzeten. De ha egy olyan alkamazáshoz kell autómatizált teszteket fejleszteni, ahol nincs hozzáférésünk a kódhoz, sokat bonyolíthat az elemek kiszűrésében, megtalálásában, főleg akkor, ha több böngészőben kell működniük a teszteknek.

Hogy megtudjuk mikortól érdemes automatizálni a tesztjeinket, összegezzük a fenti applikációra vetített időket: A keretrendszer kifejlesztése körülbelül 1 heti munkámba került úgy, hogy aktívan dolgoztam 5 napon át, napi 4 órát. Ez megközelítőleg 20 órai munka. Aztán ehhez még hozzá számolódik a Cucumber fájlok megírásának ideje is. Ez nem olyan sok idő, ha tudjuk, hogy mit kell tesztelni, írni bele. Úgy gondolom, hogy feature fájlonként ide szmolhatunk 40 percet átlagosan. Ez a 20 órai keretrendszerhez hozzátesz még 440 percet (7 órát és 20 percet, mert 11 feature fájlból áll a teszt kollekcióm). Ettől az időtől mentesülhetünk, ha nem mi írjuk a feature fájlokat. Ehhez még hozzá jön a java oldali fejlesztése a feature fájloknak, vagyis amikor működő kóddá áll össze a feature fájlok tartalma. Mindez még további 774 percbe telt, vagyis 12 órába és 56 percbe. Eddig úgy állunk, hogy ha mi kell írjuk a Cucumber feature fájlokat, akkor 1 napba, 16 órába és 25 percbe telik (= 2425 perc), ha nem mi kell írjuk őket, akkor 1 napba, 9órába és 5 percbe telik (= 1985 perc).

Ha a legjobb mért eredményt vesszük figyelembe a fentiek közül (Firefox böngészővel, de távoli szerveren, mert az életszerűbb) akkor megközelítőleg 25 másodperc alatt leteszteltük a rendszert. A manuális teszteléshez képest, ami nagyjából 8 percbe telik (itt sok múlik az emberi tényezőn) nyertünk 7 percet és 30 másodpercet (hagyunk 5 másodperc hibafaktort az ember javára). Ezen feltételek mellett, hogy megtérüljön az automatizált tesztkeretrendszerbe fektetett munka és energia, ahhoz le kell futtatnunk a teszteket legalább 265 alkalommal őket. Ettől kezdve minden csak azon múlik, hogy mennyire sűrűn futtatjuk le a teszteket.

Szóval mikortól érdemes autómatizálni a teszteket? Nos, a fentiekre alapozva elmondhatjuk, hogy ez is nagymértékben az igényektől és a helyzettől függ. Viszont összegezve a tapasztalataimat, elmondhatom, hogy akkor érdemes, amikor egy tesztesetet nagyon sokszor kell ismételni. Akkor is érdemes, amikor nagy az esélye az emberi hibának és sokat kell beviteli mezőkbe írni, mert főleg ez lassítja az embert. Akkor, amikor ugyanazt az esetet több böngészőben is el kell végezni és akkor is, ha túl hosszú időt vesz igénybe egy eset manuálisan való végrehajtása.

# Továbbfejlesztési lehetőségek

Mind a teszt keretrendszert, mind pedig a webes applikációt lehetne továbbfejleszteni.

A webes applikáció esetében lehetne újabb, a Webalkalmazás architektúra címszó alatt említett funkciókkal gazdagítani a rendszert, és megvizsgálni, hogy ahogy nő a rendszer, hogyan hat ki a tesztek futására.

Mivel a képernyőkép készítése funkció már implementálva van a keretrendszerbe, lehetne képernyőképeket készíteni az alkalmazás oldalairól, majd egy képelemző mesterséges intelligencia szoftverrel keresni a vizuális hibák után, mert egy probléma, hogy a vizuális hibákat nem veszi észre a keretrendszer.

Érdekes lenne megvizsgálni azt is, hogy a Selenium Grid, mellyel párhuzamosítani lehet a tesztek futását, hogyan hat az eredményekre, az egyes böngészőket is figyelembe véve.

# Idézett forrásmunkák

|  |  |
| --- | --- |
| [1] | „SeleniumHQ,” SeleniumHQ, [Online]. Available: https://www.seleniumhq.org/. [Hozzáférés dátuma: 02 2019]. |
| [2] | B. Hansen, „Automated Web Testing with Selenium and WebDriver Using Java,” Pluralsight, 22 09 2017. [Online]. Available: https://app.pluralsight.com/library/courses/automated-web-testing-selenium-webdriver-java/table-of-contents. [Hozzáférés dátuma: 01 2019]. |
| [3] | S. Avasarala, Selenium WebDriver Practical Guide, Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing Ltd., 2014. |
| [4] | „iDatalabs,” iDatalabs, [Online]. Available: https://idatalabs.com/tech/products/selenium/. [Hozzáférés dátuma: 05 2019]. |
| [5] | P. Sams, Selenium Essentials, Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing Ltd., 2015. |
| [6] | „Cypress,” Cypress, [Online]. Available: https://docs.cypress.io/. [Hozzáférés dátuma: 01 2019]. |
| [7] | „Robot Framework,” Robot Framework Foundation, [Online]. Available: https://robotframework.org/. [Hozzáférés dátuma: 05 2019]. |
| [8] | „Katalon,” Katalon, [Online]. Available: https://www.katalon.com/. [Hozzáférés dátuma: 05 2019]. |
| [9] | „Screenster,” Screenster, [Online]. Available: https://screenster.io/. [Hozzáférés dátuma: 05 2019]. |
| [10] | „TestCraft,” TestCraft Technologies LTD, [Online]. Available: https://www.testcraft.io/. |
| [11] | M. C. E. D. Matthew Heusser, „TechBeacon,” Micro Focus, 2015 - 2019. [Online]. Available: https://techbeacon.com/app-dev-testing/6-common-test-automation-mistakes-how-avoid-them. [Hozzáférés dátuma: 10 06 2019]. |
| [12] | R. P. MG, Learning Selenium Testing Tools - Third Edition, Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing Ltd., 2015. |
| [13] | M. W. A. H. Seb Rose, The Cucumber for Java Book: Behaviour-Driven Development for Testers and Developers, Pragmatic Bookshelf, 2015. |
| [14] | „Cucumber.io,” Cucumber, [Online]. Available: https://cucumber.io/. [Hozzáférés dátuma: 02 2019]. |
| [15] | V. K. Sharing, „Vikramvi Knowledge Sharing,” 25 08 2018. [Online]. Available: https://vikramviknowledgesharing.wordpress.com/2017/01/01/behavior-driven-development-bdd-cucumber/. [Hozzáférés dátuma: 05 2019]. |
| [16] | C. Tudose, „Java: BDD Fundamentals,” Pluralsight, 11 01 2019. [Online]. Available: https://app.pluralsight.com/library/courses/java-bdd-fundamentals/table-of-contents. [Hozzáférés dátuma: 02 2019]. |
| [17] | Trivago, „Cluecumber,” Trivago, [Online]. Available: https://github.com/trivago/cluecumber-report-plugin. |