# Disszertáció

Tartalomjegyzék

[Disszertáció 1](#_Toc11222925)

[Bevezető 2](#_Toc11222926)

[A kutatott technológiák bemutatása 3](#_Toc11222927)

[Autómatizált tesztelés keretrendszerek 3](#_Toc11222928)

[Selenium 3](#_Toc11222929)

[Mi a Selenium? 3](#_Toc11222930)

[Miért Selenium? 4](#_Toc11222931)

[Selenium IDE 5](#_Toc11222932)

[Selenium WebDriver 7](#_Toc11222933)

[WebElement-ek és betélyolásuk 10](#_Toc11222934)

[Szakirodalmi tanulmány 13](#_Toc11222935)

[Az automatizált tesztelés tipikus buktatói 15](#_Toc11222936)

[Megvalósított rendszer bemutatása 17](#_Toc11222937)

[A Webalkalmazás: 17](#_Toc11222938)

[Webalkalmazás technológiái 18](#_Toc11222939)

[Webalkalmazás architektúra 18](#_Toc11222940)

[A teszt keretrendszer 20](#_Toc11222941)

[Selenium 21](#_Toc11222942)

[Selenium WebDriver 21](#_Toc11222943)

[Page Object Model 21](#_Toc11222944)

[Cucumber 21](#_Toc11222945)

[JUnit-al való integráció 21](#_Toc11222946)

[Következtetések és tapasztalatok 21](#_Toc11222947)

[Továbbfejlesztési lehetőségek 21](#_Toc11222948)

[Mérések 22](#_Toc11222949)

[Idézett forrásmunkák 22](#_Toc11222950)

Ábrajegyzék

[1. ábra - Selenium arhitektúra Forrás: [3] 4](#_Toc11229282)

[2. ábra - iDatalabs kutatásai szerint több, mint 29000 cég használja a Seleniumot [4] 5](#_Toc11229283)

[3. ábra - Selenium IDE kezdőlap 6](#_Toc11229284)

[4. ábra – Selenium IDE példa teszteset 7](#_Toc11229285)

[5. ábra - A webalkalmazás rétegei 19](#_Toc11229286)

[6. ábra - Egyszerüsített térkép az alkalmazásról 19](#_Toc11229287)

[7. ábra - Teljes oldaltérkép 20](#_Toc11229288)

# 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

## Autómatizált tesztelés keretrendszerek

Előnyök, hátrányok

# Selenium

## Mi a 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 tesztelés 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 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 kell plusz költségekkel számolni. 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 gomb

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 az 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 különösebb 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 éppen

## 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. Itt 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 [5] is kiemel könyvében:

* 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 oldalforrását. 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 betályolásuk

A Selenium tesztek építőkövei a WebElementek és ezek betályolása. Ahogy Avasarala Satya [3] is írja könyvében, 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 betályolhatóak:

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 WebElement típusú searchForm nevű változóban a „search-form” id-val rendelkező formot és összes alatta lévő elemet.

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 elemeznie, 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.

* 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.

Absztrakt

Bevezető – egy mondatban meghatározni, hogy mit akarunk csinálni, miért fontos azt csinálni, miről van szó a dolgozatban. egy oldal terjedelem

Elméleti megalapozás, szakirodalmi tanulmány, itt milyen tipikus hibák vannak, ezeket hogyan próbálták megoldani – milyen rendszerek vannak, (Page Object model, hogy framework szerű legyen)

A rendszer megvalósítása, ahol meg kell legyen fogalmazva a funkcionalitás, amivel szeretnénk foglalkozni. Hogyan épül fel, arhitektúra, mi hogyan lett megvalósítva és miért

Mérések rész – metodológia, eredmények, eredmények tárgyalása, tesztek sorrendje, tesztek prioritizálása, tesztek egységekre bontása

Következtetések – összefoglaló akár táblázatokban, eredmények, következtetések összefoglalása.

# Szakirodalmi tanulmány

#### Mi az, ami létezik a piacon, mennyibe kerül,

Ha automatizált tesztelésre kerül a sor, és a már meglévő technológiák után kutatunk, bajban lehetünk a széles választék miatt. Bár a kör nagyon hamar leszűkíthető, ha tudjuk, hogy mik az igényeink. A piac meglehetősen széles, és minden igényre lehet találni megoldásokat. Van amely üzemeltetéséhez nem szükséges programozói háttértudás, van ahol szükség van programozói tudásra. Vannak, amelyek felhő alapú alkalmazások tesztelésére vannak kifejlesztve, sőt, olyant is találni, amelyek mesterséges intelligenciával vannak felvértezve és elemzik a teszteket. Íme, néhány népszerű 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 hogyan kell használni az egyes funkciókat. Selenium esetén ez a funkció hiányzik. A Cypress-t olyan cégek használják, mint például NASA, KIWI.COM, DHL.

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 kielégíteni az igényeinket, de cask Python nyelven. Ide már kell programozói tudás Előnye, hogy könnyen tanulható, egyszerű esetekben gyorsan használható, támogatja a Gerkhin 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 Gerkhin 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 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, ha automatizált teszteket írunk:

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 ne kézzel kelljen őket futtatni, hanem minden build után autómatikusan lefussanak. És ne akarjuk rögtön, 100%-ban tesztelni az alkalmazást, hanem szépen, fokozatosan adjuk hozzá az új teszteket, melyeket átgondolunk, mert a tesztek igazításának is fenntartási 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. 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 lesz látható a vonzata. 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.

* 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.

# Megvalósított rendszer bemutatása

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

## A Webalkalmazás:

A webalkalmazás a MyLibrary nevet kapta, ami egy egyszerű könyv és író nyilvántartó (ajánló) webalkalmazás, ahol az admin joggal rendelkező felhasználók tudnak hozzáadni új könyveket és 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 egy kapcsolat űrlap segítségével tudják a felhasználók 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 reszponzív out-of-the-box.

Az alkalmazást 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 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.



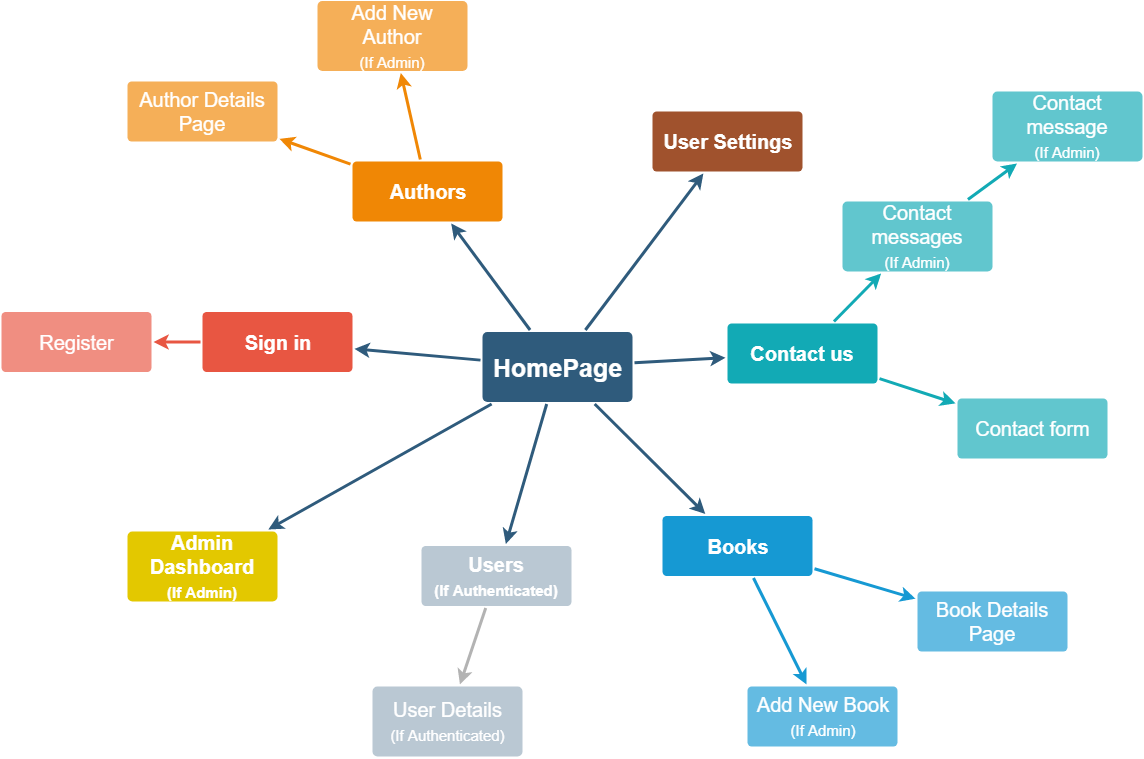
5. ábra - A webalkalmazás rétegei

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 (azért egyszerüsített, mert ebben nincsenek benne azok a funkciók, amik bejelentekzés után vagy jogosultsághoz kötve látszanak) térképét, az alábbi ábra illusztrálja:



6. ábra - Egyszerüsített térkép az alkalmazásról

Néhány oldal láthatósága jogosultsághoz van kötve, ezért a fenti ábrát kiegészítve a teljes alkalmazás térképe így nézne ki:



7. ábra - Teljes oldaltérkép

Mint a fenti ábra is mutatja, néhány funkció jogosultsághoz van kötve. Alapvetően csak két jogosultsági szint van, az Admin és a User. Csak az adminnak van szerkesztési joga (író és könyv hozzáadása, szerkesztése), és ő kezeli a kapcsolati űrlapról beérkező leveleket is.

Dióhéjban ennyi az alkalmazás, egyszerű struktúra, de minden van benne, ami a tesztek és mérések elvégzéséhez szükséges.

## 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 lesz. 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.

### 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 a Selenium 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), 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 tegyük fel más böngészőben szeretnénk futtatni, akkor mind a 10 tesztben meg kell módosítani a származtatást. 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. Ezt az osztályt használom arra, hogy a webdriver függvényeit meghívjam, így központosítva van az, hogy milyen böngészőt használok, és ennek 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

### Cucumber

BDD – Behaviour Driven Development

### JUnit-al való integráció

Mivel fogunk foglalkozni, mik a tipikus hibák

Rendszer arhitektúra

- hibák – leggyakoribb hiba, amivel találkoztam, hogy internetkapcsolat megszakad, vagy tovább tart betölteni az elemet és nem találja meg a selenium

- sebesség

- jmeter - terhelhetőség

- hogyan lehetne tesztelni azt, hogy a meglévő megy-e még ha hozzáadunk újat

- response idő, érdekesség szempontjából processzor idő

- lekérdezések eredményei és időtartama

- eredmények naplózhatósága verziók között, összehasonlítás az előző verziókkal, mi csökken, mi nő, hatékonyság

# Következtetések és tapasztalatok

komparatív összehasonlítás, keresni a piacon

Mikortól érdemes autómatizálni a teszteket?

amikor kis befektetéssel nagyot spórolhatunk

# Továbbfejlesztési lehetőségek

A teszt keretrendszert bevezetni egy continuous integration szoftverbe, hogy deploy után maguktól lefussanak és maguktól küldjenek reportot.

Selenium Grid segítségével párhuzamosan is elvégezhetőek a tesztek, bár ez a továbbfejlesztésben játszana fontos szerepet.

# Mérések

SELENIUM IDE VS WebDriver – ugyanazok a tesztek ezzel – azzal elvégezve, azért nem ide, mert akkor nem lehet a kódba nagyon bele matatni

Böngészők közti különbségek

Egyes feature tesztek fejelesztési ideje

Prioritizálni a teszteket egyszer naivan, aztán optimalizálni, hogy bejelentekzik, letesztel mindent, amit azzal a userrel kell, aztán kilép stb

# 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] | 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]. |