VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATIZOVANÉ TESTOVÁNÍ ZABUDOVANÝCH WEBOVÝCH APLIKACÍ

AUTOMATED TESTING OF EMBEDDED WEB APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ONDŘEJ DUFEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Dufek Ondřej**
Program: Informační technologie
Název: **Automatizované testování zabudovaných webových aplikací**
Automated Testing of Embedded Web Applications
Kategorie: Web

Zadání:

1. Prostudujte principy tvorby uživatelského rozhraní zabudovaných aplikací pomocí webových technologií.
2. Seznamte se s existujícími nástroji pro testování webových aplikací, jako např. Selenium.
3. Po konzultaci s vedoucím navrhněte architekturu obecného řešení pro testování zabudovaných aplikací s webovým rozhraním.
4. Implementujte navržené řešení pomocí vhodných technologií.
5. Otestujte vytvořený systém na vhodné aplikaci.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Swicegood, T.: Programming Node.js, O´Reilly, 2012
- Žára, O.: JavaScript - Programátorské techniky a webové technologie, Computer Press, 2015

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Burget Radek, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 21. října 2019

Abstrakt

Tato práce je zaměřena na automatické testování zabudovaných webových aplikací, konkrétně na jejich uživatelská rozhraní. Problematiku jsem řešil na již připraveném rozhraní. Použil jsem automatizační nástroj Selenium. Vytvořené řešení ulehčuje vývoj automatických testů zvoleného produktu Ray3 a umožňuje regresně zkontrolovat novou verzi produktu s minimální časovou investicí. Výsledné řešení umožní zajistit větší kvalitu výrobku a ulehčit práci manuálním testerům.

Abstract

This thesis focuses on automatic testing of embedded web applications, specifically on their user interfaces. I solved this problem on an already implemented user interface. I designed and implemented an automated test environment based on the Selenium tool. The created solution facilitates the development of automatic tests of the selected product and enables regression testing of a new version of the product with a minimum time cost. The resulting solution will ensure greater product quality and make work easier for manual testers.

Klíčová slova

Selenium, DOM, automatizované testování, zabudované webové aplikace

Keywords

Selenium, DOM, automated testing, embedded web applications

Citace

DUFEK, Ondřej. *Automatizované testování zabudovaných webových aplikací.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Burget, Ph.D.

Automatizované testování zabudovaných webových aplikací

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Ondřej Dufek
3. června 2020

Poděkování

Chtěl bych poděkovat Ing. Radku Burgetovi, Ph.D. za vedení a ochotu při řešení této práce. Dále bych chtěl poděkovat firmě RACOM s.r.o. za poskytnutí testovací stanice a jejím pracovníkům za odbornou pomoc.

Obsah

Kapitola 1

Úvod

V dnešní době jsou webová rozhraní hodně populární, protože jsou pro cílového uživatele dobře pochopitelná a přehledná. Je to tedy věc, kterou by každý rozšířený, uživatelsky přívětivý produkt měl mít. A samozřejmě by se měl snažit ji mít co nejvíce spolehlivou.

U zabudovaných rozhraní bývá často velký počet vzájemně navázaných konfiguračních možností. Například u stanic, které spolu navzájem komunikují přes vzduch, může nastat obrovské množství kombinací frekvencí, pásem a ostatních věcí, které zajistí, že se stanice spojí a budou spolu bezproblémově komunikovat.

U takových komplexnějších zařízení nastává často problém s jejich otestováním. Ručními testy se v takovém případě může pokrýt vždy jen nepatrná základní část konfigurace. Proto je velmi výhodné vytvořit automatické testy, které zvládnou zkontrolovat mnohem více možností konfigurace a ušetří práci člověka. Automatizace je v tomto století velmi populární a čím dál tím víc vyžadovaná záležitost napříč odvětvími.

Proto se tato práce zabývá možnou automatizací webových aplikačních rozhraní nad zabudovanými systémy. Testuje se již hotová stanice zapůjčená od firmy Racom s.r.o. se stále vyvíjeným uživatelským webovým rozhraním. Jako automatizační nástroj pro práci s webovými rozhraními je použit Selenium Webdriver nad jazykem Python3, kde se využije i framework Pytest sloužící pro jednotkové testování. Automatizovat se bude předem daný testovací předpis, který slouží jako regresní a funkční test.

Kapitola 2 se zabývá základními principy tvorby webových uživatelských rozhraní nad zabudovanými systémy. Probírá se zde, z čeho se rozhraní skládají a jaké jsou jejich vlastnosti a používané architektury. Pomocí jakých nástrojů lze takové rozhraní vytvořit a jak se k němu poté přistupuje. Nakonec je zmíněna i stanice s rozhraním, na kterém se výsledné řešení vyvíjelo.

V kapitole 3 jsou představeny různé způsoby testování systémů (primárně tedy webových rozhraní) a jejich základní rysy, možnosti použití a výhody a nevýhody jednotlivých způsobů. Poté jsou vysvětleny nejčastěji používané styly a metody testování webových rozhraní a kdy je jejich použití vhodné.

Jednotlivé nástroje, které mohou usnadnit vývoj programu, který automaticky testuje web, a jejich frameworky a produkty se vyskytují v kapitole 4. Jde o nástroje, které jsou přizpůsobené ke komunikaci s webovým rozhraním a provádění operací nad jeho prvky.

Navržené řešení a architektura výsledného programu je probrána v kapitole 5. Zde jsou probrány i prvotní požadavky a cíle projektu, jeho vstupy a výstupy. Poté je zde představena a popsána navržená architektura.

V kapitole 6 se nejdříve probírá, které nástroje byly k naplnění původního očekávání použity, pod jakým jazykem a frameworkem program vznikal a hlavní způsob komunikace s

rozhraním testované stanice Ray3. Poté jsou popsány typické implementační případy, které tvořily hlavní jádro celého programu, a práce s daným rozhraním. Zmiňují se tu vstupy a výstupy výsledného programu a jejich charakteristika. Zmíněno je i spouštění paralelních testů a obecné záležitosti automatizace nad webovými prohlížeči.

Kapitola 7 vysvětluje průběh a proces jak průběžného, tak konečného testování. Je zde rozebráno vlastní testování a pomoc od externích testerů a způsob naší komunikace a vyhodnocování chyb. Je zde i zmíněná práce s gitem a jeho nástrojem issuetracker, který se při testování velmi používal. Celkově jsou zde popsány různé faktory působící na testování a způsoby vyhodnocování chybných stavů.

Kapitola 2

Uživatelská rozhraní zabudovaných webových aplikací

Zabudovaná aplikace je nějaký software, který je nastálo vložen do jednoúčelového zařízení.

Toto jednoúčelové zařízení se vyznačuje tím, že má svůj specifický účel, kterému je podřízen jeho rozsah funkcí a využití. Uživatel zařízení využívá pouze k jednomu účelu, který je pro ono zařízení typický. Je neefektivní ho upravovat na jinou činnost. Příkladem můžou být hodinky, trouba nebo složitější router či radiomodem. Má to tu výhodu, že zdroje nejsou takový problém a je to relativně jednoduché pro vývoj. Protikladem jednoúčelového zařízení je zařízení univerzální, které je už náročnější na vývoj a zdroje, protože je hodně komplexní a složité. U těchto typů zařízení si uživatel může vybrat, k čemu ho bude používat, a nainstalovat si tam patřičnou aplikaci. Příklad takového zařízení jsou třeba osobní počítače nebo moderní chytré telefony.

U zabudovaných aplikací pak záleží, jestli potřebují operační systém, nebo nikoli. Třeba v minulém příkladu trouba OS nepotřebuje, protože má jen pár funkcí a tlačítek pro obsluhu. To z ní dělá relativně jednoduché zařízení a OS by tu byl zbytečný luxus.

Naopak třeba u routeru už je OS vhodný, protože má komplexní provázané nastavení. Pracuje na mnoha úkolech naráz a musí vyhodnocovat údaje, stavy, zdroje atd. v reálném čase.

V jednoúčelových zařízeních je tedy většinou nějaké jedno uživatelské rozhraní a to umožňuje uživateli aplikaci ovládat. [24]

2.1 Tvorba webového rozhraní

V současné době není tvoření běžných webových rozhraní nijak zvlášť náročné a je to známé téma. Pořád vznikají nové weby, pro jejichž tvorbu je připraveno mnoho programovacích jazyků, a ještě více nápomocných frameworků, které programátorům ulehčují práci se základními problémy při vývoji webových stránek.

2.1.1 Webové technologie v souvislosti s uživatelským rozhraním

Uživatelská rozhraní můžou mít mnoho podob. Můžou to být například tlačítka na ovladači, ale v moderní době se s příchodem internetu a webových technologií začaly čím dál tím více objevovat webová uživatelská rozhraní.

Hodně produktů a aplikací se přesunuje čistě na softwarové zpracování a tím, že je nyní internet dostupný téměř odkudkoli, začaly se aplikace přesouvat spíš do webových prostor,

což uživatelům extrémně ulehčuje přístup atd. Pro uživatele by to mělo být i nejlehčí, co se týče použití.

2.1.2 Jazyky a frameworky

Webové rozhraní může programátor vytvořit pomocí mnoha programovacích jazyků. V základu jde hlavně o poměrně známé jazyky jako je HTML, CSS, PHP, JavaScript, Java, Python a další.

HTML

Je značkovací jazyk, který tvoří hlavní strukturu a logické rozložení stránky. HTML je charakterizováno značkami a jejich atributy, ze kterých se skládá celý kód. Je hodně jednoduchý a intuitivní. Bohužel pro vytvoření průměrné webové stránky zdaleka nestačí použití HTML.

CSS

Popisuje, jak se mají zobrazit elementy, které jsou popsány pomocí značek v HTML. Také je to docela jednoduchý a intuitivní jazyk, díky kterému můžeme proměnit základní webové rozhraní v uživatelsky přívětivé.

Na ulehčení tvorby CSS je mnoho dobrých frameworků, které jsou lehké na naučení a díky nim se dá za krátký čas udělat dostačující základ. Jedním z příkladů těchto frameworků je Bootstrap.

JavaScript

Jeden z nejpoužívanějších programovacích jazyků vůbec. Používá se hlavně ke tvorbě interaktivních prvků na stránce. Můžou to být například tlačítka, animace, efekty atd. Interakce s uživateli je důležitá kvůli získávání dat a informací o uživateli a následném zobrazení obsahu. Proto má JavaScript určitě své místo ve webech.

Nejznámější a nejpoužívanější frameworky na práci s JS je Angular a React. Nicméně určitě by se dalo najít mnoho dalších frameworků, ze kterých si každý může vybrat ten nejlépe vyhovující.

Java

Další z nejpopulárnějších programovacích jazyků. Java má rozsáhlou komunitu a tím, jak je to používaný jazyk, na něm vzniká mnoho frameworků na mnoho možných použití. Ani weby nejsou výjimkou.

Na webové uživatelské rozhraní se zaměřuje například framework Vaadin, který vám umožní psát celý web jen v tomto frameworku bez použití ostatních jazyků.[25] Samozřejmě ale je opět spousta možností.

Python

Poslední z třetice aktuálně nejpoužívanějších programovacích jazyků. Platí zde totéž jako u Javy. Díky vysoké používanosti tohoto jazyka, je zde také mnoho frameworků, které jsou připraveny vám ulehčit práci a díky kterým máte možnost si pouze se znalostí Pythonu vytvořit design stránek.

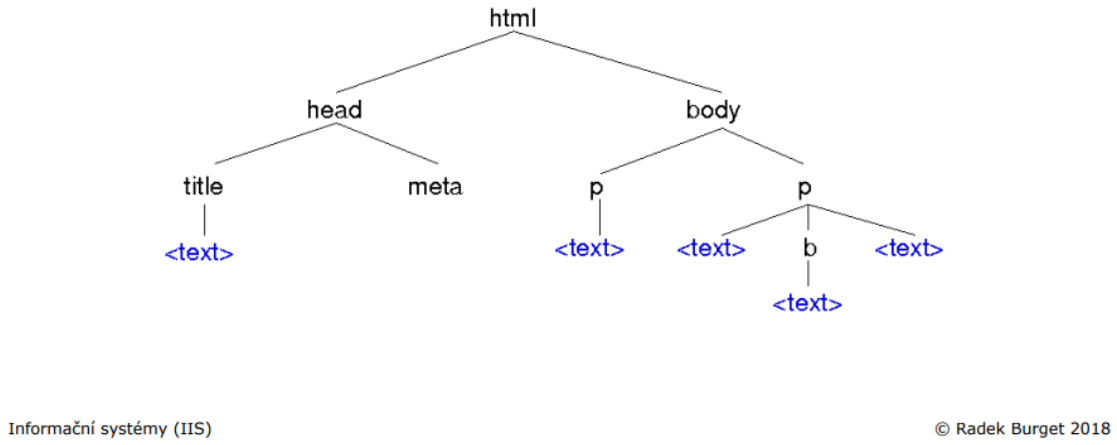
Příklady nejpoužívanějších frameworků pro Python jsou Django, CherryPy a Bottle. [17]

2.1.3 Elementy

Při manuálním testování se řeší hlavně intuitivní proklikávání webu. Na stránce elementy vidíme a jsme schopni poznat, co máme s jakým prvkem dělat a kde ho máme hledat i bez jeho logického označení a názvu.

Nicméně, pokud chceme používat skriptovací nástroje a nemáme zapojenou umělou inteligenci, program nic intuitivně nevytuší a ani neví, kde má prvek hledat. Musíme mu tedy nějakým způsobem s tímto pomoci.

Na webovém rozhraní se tedy logicky orientujeme pomocí elementů, které vznikají už na vrstvě HTML. Všechny elementy na stránce tvoří jeden provázaný strom podle závislostí.



Obrázek 2.1: Strom HTML elementů [2]

Při testování funkčnosti webové aplikace se tedy nejčastěji kontroluje požadovaná funkčnost jednotlivých prvků (elementů) na stránce.

2.1.4 Označení a přístup k elementům

K tomu, abychom mohli automatické testy psát a obecně elementy na webu logicky rozlišovat, potřebujeme každému elementu přiřadit jedinečné označení. S tímto identifikátorem poté pracujeme v našich automatických testech, kde tímto umožníme našemu nástroji nad elementy provádět určité činnosti. Element nemusí mít žádné speciální označení, ale je dobrým zvykem je označovat pro budoucí možné využití. Označit se dá mnoha způsoby popsanými níže.

K elementům na stránce poté přistupujeme podle toho, jaké označovací atributy máme k dispozici. Ne všechny jsou vždy použité. Zde hodně záleží na vývojáři daného webového rozhraní.

K elementům můžeme přistoupit pomocí:

- ID - Jedná se o jednoznačný identifikátor elementu v rámci dokumentu. Například: `<form id="loginForm">`

- Name - Stejný přístup jako u ID, akorát má tvar `<form name="loginForm">"`
- XPath - Toto je mocný nástroj. Je to jazyk, který popisuje buď relativní nebo absolutní cestu k uzlu. Absolutní popis vystihuje cestu od kořenu až po daný element. Př: `/html/body/form[1]`. Takto můžeme jedinečně najít nějaký element, aniž bychom ho museli popisovat. Má to ovšem i negativní stránku. Když se změní rozložení webu, tyto absolutní cesty mohou přestat fungovat a rozbít nám tím část programu. Relativní popis propojuje absolutní cestu s nějakým dalším atributem. Př: `//form[input/@name='username']`.
- Tag name - Je název tagu daného elementu. Například u `<h1>Welcome</h1>` by to bylo 'h1'. Zde ovšem nemůžeme moc dobře zajistit jedinečnost.
- Class name - Slouží pro sjednocení CSS stylů, proto taky asi nebude jedinečný. Př: `<a class="content">`
- CSS selector - Speciálním popisem se k elementu dostaneme pomocí ostatních atributů. Příklad s minulým class name: `a.content`

2.2 Přístup k webovému rozhraní

K webovému rozhraní se dostaneme přes nějaký webový prohlížeč. Ač se to na první pohled nemusí zdát, na prohlížeči také záleží. Různé prohlížeče mohou dodržovat různé normy a zásady, a tak se může stát, že se na různých prohlížečích jedno webové rozhraní bude v určitých aspektech zobrazovat různě. Proto je vhodné si při vývoji a testování zjistit, pro které prohlížeče chceme rozhraní optimalizovat. Jinými slovy, jaké prohlížeče naši uživatelé nejvíce používají. Obecně jsou nejpoužívanější prohlížeče Chrome, Firefox a Safari.

Stanice s rozhraním může být umístěna jak na lokální síti a přímo spojená s počítačem nebo může být umístěná někde ve viditelné internetové síti a k tomu je potřeba přístup k internetu nebo k té dané síti.

Pak už potřebujeme jenom její adresu a můžeme se k ní připojit. Komunikace se většinou uskutečňuje přes protokoly HTTP nebo HTTPS.

2.2.1 HTTP

Je aplikační protokol, který se označuje číslem 80. Využívá transportního protokolu TCP pro zaručený přenos dat. Jeho hlavním smyslem je přenos hypertextových souborů ve formátech HTML, XML atd. Funguje na principu, že klient pošle nějaký požadavek a server mu na něj pošle odpověď. V tomto případě jsme klient my a server je ta daná stanice s webovým rozhraním. Používá rozšíření MIME, čímž může posílat kterýkoli soubor.[16]

Mezi dvě nejčastěji používané metody se řadí GET a POST.

2.2.2 HTTPS

Je další a tentokrát modernější aplikační protokol označovaný pod číslem 443. Opět využívá transportního protokolu stejně jako předchozí HTTP a je určen také ke komunikaci a přenosu dat.

Od HTTP se liší hlavně tím, že je zabezpečený. Je to vlastně protokol HTTP, ale spojený s protokolem TLS (Transport Layer Security). [3]

Poskytuje navíc:

- Šifrování
- Ověření
- Integritu dat

V současné době je zabezpečení dat velkým tématem, a proto je HTTPS mnohem používanějším a stal se takřka samozřejmostí.

2.2.3 Architektury

Aktuálně stojí za zmínku dvě základní architektury načítání obsahu serveru, které fungují trochu odlišným způsobem. [1]

Renderování na serveru

Dříve hodně používaná architektura. Má docela jednoduchý princip. Když klient pošle požadavek, že chce načíst některou stránku, server mu ji celou vyrenderuje a přes HTTP mu pošle celý HTML kód.

Tohle řešení funguje a je i poměrně rychlé, ale ne vždy může být nejefektivnější. V případě, že chceme načíst jen nějaká data ze stránky, nebo nějakou její část, tento způsob nám znova vyrenderuje celou stránku. Všechno CSS, HTML i data, což je zbytečné. Tento problém řeší renderování u klienta.

Renderování na straně klienta

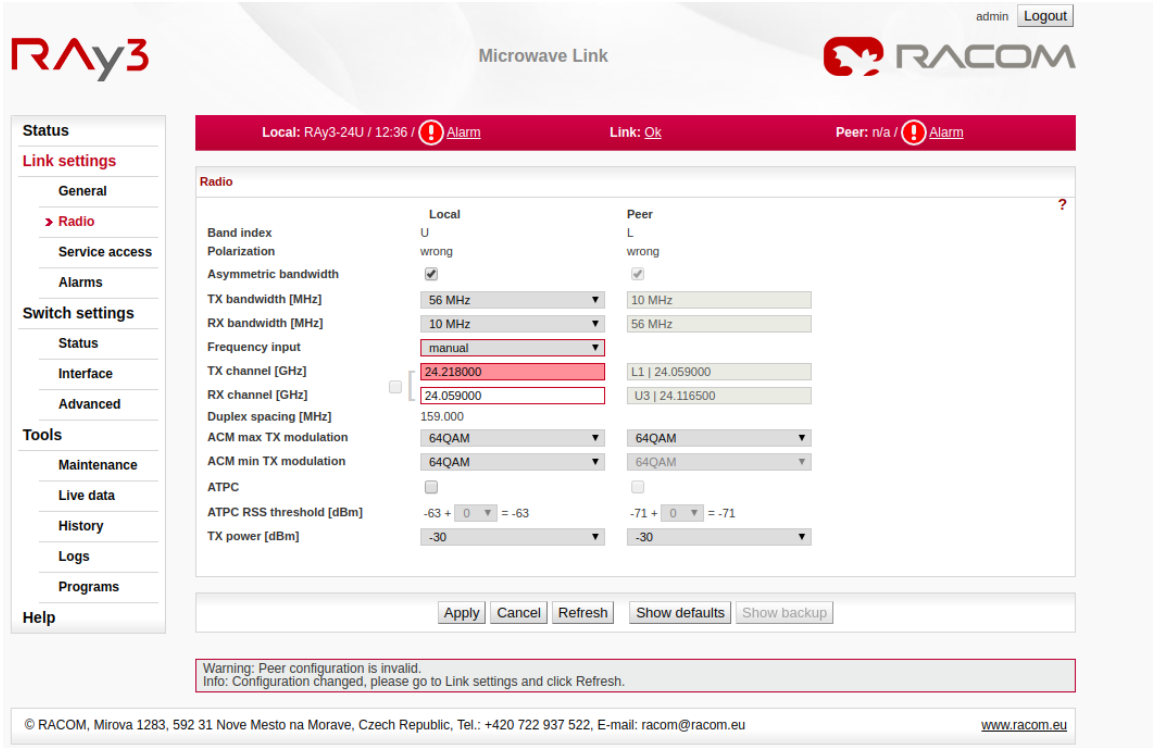
Tato architektura je v současné době populární, protože dokáže proces načítání hodně zefektivnit. Funguje tak, že když klient zašle požadavek, serverové API zašle požadovaná data pomocí datového formátu (například JSON) a klient si sám data zpracuje a následně z nich vytvoří DOM strom. Proto je výsledný přenos a načtení obsahu klienta mnohem rychlejší a všechna potřebná data jsou samozřejmě poskytnuta.

2.3 Stanice Ray3

Praktická část této práce se vyvíjela na stanici Ray3. Je to point-to-point mikrovlnný spoj. Má klasické webové grafické rozhraní, možnost komunikace přes ssh a telnet a v neposlední řadě ovládání přes mobilní zařízení, kde má také grafické rozhraní v aplikaci. Tato práce se bude zabývat především jejím webovým rozhraním. Sám jsem jeho rozhraní nevyvíjel, ale stanice mi byly zapůjčeny i se současně vyvíjeným rozhraním od firmy RACOM s.r.o.

Spoj stanic slouží k vzájemné vysokorychlostní komunikaci na delší vzdálenosti. Produkt Ray se aktivně využívá již přes 10 let po celém světě. Software stanice je dělaný na operačním systému Linux. Je to kvalitní, spolehlivý a rychlý systém.

Webové rozhraní stanice je napsáno netradičně v programovacím jazyce C++. Konkrétně byl použit framework Webtoolkit. Ten umožňuje vytvořit monolitickou aplikaci, která obsahuje jak server, tak i backend. Je možné vytvořit plné dynamické uživatelské rozhraní, nebo použít statické části webu zapsané jako šablony ve formátu XHTML. Formátování webu se řeší pomocí standardního CSS.



Obrázek 2.2: Ray3 webové rozhraní

Kapitola 3

Testování zabudovaných systémů

Existuje mnoho způsobů a metod testování s různými cíli. Výběr se odvíjí od mnoha faktorů. Je dobré se ptát na otázky jako:

- Jaký je náš testovací produkt?
- Co má být testováno?
- Jak mají být testy rozsáhlé?
- Jaký je rozpočet na testování?
- Doba vývoje produktu
- Updatování produktu

Samozřejmě je toho více, ale od těchto základů se bude odvíjet, jestli budeme dělat testy rozsáhlé a opakované. Nebo jestli stačí jen krátké a strohé. Jestli řešit funkčnost nebo odolnost a tak dále.

Zde budou probrány jednotlivé možnosti testování (především webových rozhraní), jako je hlavně manuální a automatizované testování a s tím související testovací předpisy.

V případě stylů testování webových aplikací jsou asi nejrozšířenější funkční testy, a to regresní a smoke.

3.1 Manuální testování

Manuální neboli ruční testování, jak už z názvu vyplývá, je způsob ověření kvality a funkčnosti nějakého produktu jen za pomoci schopností a činností člověka. Je to nejzákladnější a nejpoužívanější metoda a je to první krok způsobu testování čehokoli.

Princip

Obecně oproti automatizovanému testování vyžaduje manuální způsob testerovu plnou pozornost a schopnost vyhodnocování situací. Principem je zde průchod produktem (v našem případě webovým rozhraním), kde člověk zkouší zadávat různé vstupy a očekává patřičné výstupy. Když se objeví jiný výstup, než byl očekávaný, jde o chybu. Kvůli tomu jediné, co tester potřebuje vědět, je dobrá znalost onoho testovaného produktu. Musí znát jeho funkčnost a musí být schopen rozeznat požadované výstupy od těch nechtěných. Není nutné znát

produkt do hloubky. Například není nutné znát kód nebo způsob programování. Stačí jen povrchní znalosti, aby tester věděl, jaké reakce má od zařízení čekat.

Většinou je zvykem manuálního testování prvně zkontrolovat produkt, jak se říká, od A do Z. Na příkladu našeho webového rozhraní je to od úvodní stránky pokračujíce první stránkou menu až k té poslední stránce v úvodním menu. Přičemž je potřeba na každé stránce zkontrolovat funkčnost všech elementů a kombinací nad nimi, což nemusí být rychlá práce. Ideální je vyzkoušet všude racionální vstupy a zkontrolovat patřičné výstupy a také zkusit nesmyslné vstupy a zkontrolovat s tím spojené chybové hlášky, popřípadě jiné ošetření systému proti nemožným vstupům.

Pokud už jsou opravené nějaké známé chyby z předchozích testování, je možné prvně otestovat ty. Aby se ušetřil čas a programátoři se mohli hned pustit do dalšího opravování produktu.

Dále pak přichází na řadu náhodné testování a kombinace různých stránek a elementů. Nestandardní vstupy, pokusy atd.

Výhody

Výhodou, proč je manuální testování tak rozšířené, je, že je hodně jednoduché a intuitivní. Zvládne to takřka každý. Tester na to nepotřebuje žádné vysoké vzdělání. Nemusí být ani moc technicky zdatný. Samozřejmě, že čím lepší bude mít tester technickou znalost, bude lépe a intuitivněji hledat chyby. Ale obecně je to docela nenáročná činnost. To je dost výhoda, protože v případě například nemoci onoho testera, se dá místo snadno nahradit, nebo jak to často bývá, si to vyzkouší sám programátor.

Další věcí je taky krátkodobá rychlost. Když potřebujeme něco jednorázově otestovat anebo vývoj nebude tak dlouhý na to, aby se ubíral čas na psaní automatických testů, je to ideální volba, protože je to za relativní chvíli hotové.

V neposlední řadě se musí přiznat, že člověk je stále ještě chytřejší a svým způsobem vnímavější než počítač nebo i umělá inteligence, a to je zde velkou výhodou, která manuální testování bude vždy vyžadovat, dokud se toto stanovisko nezmění. I když bude automat kvalitně zpracovaný, může obsahovat chyby. Je to tak, protože ho vytvářel člověk a chybovat je lidské. S tím souvisí i grafický vzhled stránek. Zatím nejsou tak dobré nástroje, kterým bychom mohli naplno svěřit grafický vzhled webu. Vychází to z faktu, že program nemá takový nadhled a schopnosti jako člověk.

A také i když už budeme automatizovat, tak některé části mohou být na automatizace zbytečně náročné a ohromně neefektivní, takže vyjde mnohem lépe, když je přenecháte manuálnímu testování. Například firma, která začíná s automatizací a nemá tolik zdrojů, nebude automatizovat fyzické úkony (stisk tlačítka, ..).

Nevýhody

Vše má své pro a proti. I když je manuální testování tolik používané a v některých chvílích nejspíš i nezbytné, má svoje problémy.

Jedním z nich je určitě neefektivita při dlouhodobém vývoji. Dejme tomu, že se webová aplikace vyvíjí třeba 3 roky. Je zde započítán hlavní vývoj a následná podpora a oprava chyb. Nová verze softwaru může vycházet klidně každé dva týdny a občas bude třeba něco otestovat dříve. A když bude aplikace hodně složitá, bude to stát testera hodně opakující se práce. Tester bude nutně provádět určité úkony pokaždé stejně, protože musí zkontrolovat vše. To poté zabírá testerův čas, a tedy i zdroje firmy.

Dále tester nemůže nikdy otestovat vše. Jednak se stále opakujícím se testováním upadá testerova pozornost, a tudíž i výkon, ale navíc když uvážíme složitou aplikaci, tester nikdy nemůže zkusit sám všechny možné kombinace atd. Například když si vezmeme nějaký radiový komunikátor, který má přijímač a vysílač, oba mají na výběr mezi mnoha šířkami pásem a ještě více frekvencemi, je každému jasné, že člověk nemůže otestovat všechny tyto kombinace. Bylo by to nanejvýš neefektivní. Proto zde člověk zkontroluje pouze pár náhodných kombinací a bude doufat, že ostatní fungují také. A zde je ideální příležitost pro automat.

Člověk, jak víme, nemůže pořád jen pracovat. Když je třeba vydaná nová verze a musí ji kontrolovat člověk, může to trvat i několik dní. Program tento problém nemá a není takový problém automatický test pouštět přes noc. To může urychlit samotné testování.

Shrnutí

Z rozboru výše nám vyplývá, že manuální testování může být výhodné hlavně při jednoduchých produktech, při nedostatku zdrojů firmy nebo když tester není programátorsky zdatný.

Naopak při dlouhodobém vývoji a prostoru technicky zdatných testerů je všeobecně výhodné alespoň opakované části testů zautomatizovat.

3.2 Předpis

Vždy je dobré, pokud se jedná o nějakou rutinní činnost, která je spíše neměnná, vytvořit si nějaký předpis jako pomocnou berličku.

Příklad může být dlouhodobé manuální testování. Když máme nějaké rozhraní, chceme otestovat všechny stránky a jejich obsah. Když je to ale komplexní web nebo aplikace, nejspíše budeme muset kvůli časovým důvodům prvně zkontrolovat ty nejdůležitější věci. Proto je dobré si vytvořit předpis, kterého se vždy budeme držet, abychom nějakou důležitou věc náhodou nevynechali.

Předpis může být sestaven například z věcí, které víte, že vaši zákazníci nebo vaši uživatelé nejčastěji používají. Pokud budete mít například školní stránky, je důležitější přihlášení, přehled známek a aktuality výuky než například možnosti změny barvy jednotlivých předmětů atd.

Testovací předpis může být užitečný taky v případě, že bude daný tester nemocný nebo jakkoli nemohoucí svou činnost vykonat, a tak ho bude muset při akutních situacích zastoupit někdo jiný. V případě, kdy náhradník s daným produktem nemá žádné zkušenosti a může jít klidně i o jednodušší věc, ho předpis rychle a efektivně provede celým hlavním procesem. Když je předpis dobře napsaný, měl by kdokoli být schopen produkt otestovat aspoň po základní stránce a v relativně krátkém čase. A to je velká výhoda.

Vytváření předpisů

Předpis by měl většinou vytvářet nějaký zkušený tester, který je dobře obeznámen s funkčností produktu a má ho "osahaný", tzn. že ho několikrát v praxi testoval a zná jeho vstupy a požadované výstupy.

Většinou samotný tester nestačí, protože předpisy by měly být napsané od začátku vývoje produktu a v této fázi ještě tester nemá dost času a znalostí se s produktem natolik

seznámit. Proto je vhodné, aby tester úzce spolupracoval s vývojářem daného produktu, který by mu poskytl přesné informace a který by k testování taky určitě měl co říct.

Ideálně by měl být předpis kvalitně napsaný. Kvůli příkladům výše bychom měli klást důraz, aby byl správně sestavený, srozumitelně a konkrétně popsaný a aktualizovaný. S tím, jak se vyvíjí produkt, měly by se vyvíjet i testy.

3.3 Automatizace

S pokrokem technologií zaznamenáváme v současných obdobích globální nárůst automatizace. Není to jen případ testování webových aplikací. Jak si můžete všimnout, napříč všemi odvětvími po celém světě je trend a tendence zapojit co nejvíc automatizace do celého procesu.

Proč se tomu tak děje? Je to způsobeno našim vývojem. Žijeme v době, kdy se naše společnost a tím i produkty a služby mění nejvíce za celou historii lidstva. Současně také v aktuální době vzniká nejvíce nových produktů a služeb. Je tomu umožněno proto, že lidé začali čím dál tím víc zapojovat automatizaci do procesu. Před několika stoletími všichni lidé měli rutinní práce. Dobrý příklad může být zemědělství. Tehdy muselo být několik lidí, aby zaseli, vypěstovali a sklidili úrodu na poli. V dnešní době na to stačí jeden člověk s automatizovaným strojem, který to celé zvládne za zlomek času a nemá s tím skoro žádnou práci. A je jen otázka času, kdy už ten automatizovaný stroj nebude muset člověk ani řídit. Tento proces se stane plně automatizovaný a člověk už s tím nebude mít žádnou práci. Současně se lidstvu daří rutinní činnosti velmi dobře automatizovat a s postupem času to půjde ještě mnohem lépe. Proto už ve světě čím dál tím víc zanikají tyto opakované činnosti, nahrazují je automaty a lidé mají tím pádem mnohem více prostoru na to být kreativní.

Automatizace obecně je proto velmi aktuální pojem. Řeší se ve většině odvětví a je to teď jeden z ukazatelů a cest k pokroku.

Ukazatele výhodnosti automatizace

V současné době můžeme poznat prostor a výhodnost automatizace u činností, které:

- Opakují se pořád dokola
- Jsou jednoduché, zvládne je většina lidí
- Mění se jen občas
- Pracuje na nich hodně lidí
- Jsou vyžadovány často

Podle těchto ukazatelů jsme schopni poznat, že je vhodné zautomatizovat danou činnost. Čím více a čím silněji jsou ukazatelé zastoupeny, tím větší je výhoda a efektivita automatizace.

Je dost možné, že v dalším vývoji automatizace se dané ukazatele změní a půjdou automatizovat i více kreativní činnosti.

Automatizace testů webu

Informační průmysl není v této oblasti výjimkou. Automatizace je čím dál tím víc vyžadovaná i zde. Jak už bylo výše řečeno, je vhodná na rutinní činnosti. A tím zčásti testování

může být. Výše zmíněný testovací předpis je nejlepším příkladem. Pří prozkoumání obsahuje hodně dříve řečených ukazatelů. Je to stále opakovaná činnost, která se moc nemění a nemusí být nějak odborně náročná.

Další faktorem je, že manuální testování není sice moc znalostně náročné, ale v komplexních produktech může být poměrně rozsáhlé. Musíme vzít v úvahu, že s každým dalším propojeným elementem na stránce se manuální testování exponenciálně ztěžuje a prodlužuje. Nastává stále více kombinací a možností, jak stránku otestovat. Samozřejmě, že nejspíš není v našich silách otestovat vždy úplně vše, ale stále platí, že nám jde o to, otestovat z produktu co nejvíce. A tento problém opět nahrává automatizaci.

Framework nad rozhraním

Pokud se automatizuje nějaký komplexnější a složitější produkt, kde neplánujeme zautomatizovat jen malou část, přijde mi velmi výhodné nedělat program na jedno použití. Abych to blíže vysvětlil, tak pokud máme tušení, že se testovacích programů bude vytvářet nad jedním rozhraním více, je dobré vytvořit obecně praktickou a pomocnou knihovnu funkcí, která se dá opakovaně použít. Tento framework je dělán nad daným testovaným rozhraním a vytvoří se v něm různé funkce stále opakovaných a využívaných úkonů na webu. Například funkce pro otevření hlavní stránky, přihlášení atd.

Sám framework se může vyvíjet spolu se samotným hlavním testovacím programem, protože nechceme dělat ve frameworku funkce, které potom reálně nepoužijeme. Je ale dobré většinu funkcionality dávat do frameworku a z něj až poté skládat a popřípadě lehce doplňovat výsledný testovací program.

Když už jednou máme tento náš vytvořený framework pro dané rozhraní, kterého velká část nejspíše vznikne po prvním testovacím automatu, je jednoduché z něj poskládat nějaký další testovací program, protože už máme připravené funkce onoho rozhraní.

Framework je tedy skvělá příležitost, jak si ulehčit hodně práce znovupoužitelností při programování více testů.

3.4 Funkční testy

Používají se na otestování celkové funkčnosti implementace a na to, že aplikace dělá to, co po ní zákazník chtěl. Jsou to black-box testy. Musíme přesně vědět, jak aplikace bude reagovat na naše vstupy, ale už nemusíme znát kód a přesné fungování uvnitř systému, Jen zkoušíme, že pomocí našeho nastavení konfigurace systém opravdu provede to, co po něm požadujeme. V ideálním případě by se měly otestovat všechny funkce aplikace.

Funkční testy zadávají určité vstupy a očekávají určité výstupy. Tímto způsobem můžeme poznat, jestli aplikace skutečně dělá to, co by dělat měla a funguje správně. Proto jsou tyto testy hodně používané a odhalí mnoho chyb. [26] [5]

Existuje mnoho způsobů funkčního testování. Zde se budou probírat hlavně dvě nejpoužívanější metody v testování zabudovaných webových aplikací, a to jsou regresní testy a smoke testy.

3.4.1 Regresní testy

Při vývoji produktu by se mělo zajistit, aby v nových verzích fungovala již hotová implementace. Jinak vývoj nemusí postupovat dopředu a zákazníci můžou mít s příchodem verze ještě více problémů než předtím. A to nikdo nechce.

Tento styl testování je proto založený na opakování stejných testů s příchodem každé nové verze produktu. Prakticky se napíše testovací předpis a ten se zkontroluje pokaždé, když se systém vyvine. Testují se části, které už byly úspěšně implementované, protože se může stát, že kvůli komplexnosti systému a návaznosti jeho jednotlivých částí se s novou implementací nějaká na ní závislá část rozbije. Regresní testy by nám tedy měly zajistit bezporuchovost již implementovaných částí.

Jelikož se zabudované aplikace mohou vyvíjet delší dobu, je tato metoda hodně rozšířená a používaná. Regresní testy většinou pokrývají většinu částí rozhraní aplikace, a proto je zde velmi vhodná automatizace. Pokud u systému často vznikají nové verze, je automatizace o to více vhodnější.

Opakem regresních testů jsou testy progresní. U nich se naopak testují jen nově implementované části, u kterých se ještě neví, jestli jsou funkční. [6] [23]

3.4.2 Smoke testy

Tyto testy se provádí při dokončení aplikace a je možné ji spustit. Otestuje se tím základní funkčnost, že je kostra správně implementovaná a může se přejít k hloubkovým testům.

Díky krátkému rozsahu není tak náročné je zautomatizovat. Jedná se zde o testy jako například: "Jde program spustit?"

Tyto testy jsou vhodné i pro menší projekty, které v rámci vývoje nemají tak velký prostor pro testování. [7]

3.5 Nefunkční testy

Slouží pro kontrolu ostatních věcí, které tak úplně nesouvisí s funkčností. Testuje se například o dostupnost aplikace při větším připojení uživatelů naráz, spolehlivost zobrazení, rychlost při určitých zátěžích, nároky na paměť, zabezpečení a mnoho dalších věcí. Jde o jakési výkonové testování, které nám má přiblížit, jak se bude aplikace chovat při určitých situacích. Neřeší se zde již funkčnost celé implementace a různých konfigurací, ale jde o běh aplikace jako takové. Nefunkční testy nám mohou nastínit kvalitu aplikace v těchto oblastech při dalším vývoji. [5] [22]

Kapitola 4

Nástroje pro testování webových aplikací

Testování jako takové je samozřejmě rozšířená problematika, a tak díky tomu vzniklo nespočet možností, jak ušetřit práci testerů a programátorů.

Nástrojů existuje určitě mnoho a je možné si vybírat mezi dobrými nástroji takřka v každém známějším programovacím jazyku. Dobré je, že v dnešní době jsou a stále vznikají všemožné pomocné frameworky a programy, které můžou pomoci jak začínajícím manuálním testerům, tak i zkušenějším programátorům.

Zde se budu zaobírat těmi nejvíce populárními nástroji v současné době, které mají širokou podporu jak od komunity, tak ve škále programovacích jazyků.

4.1 Selenium

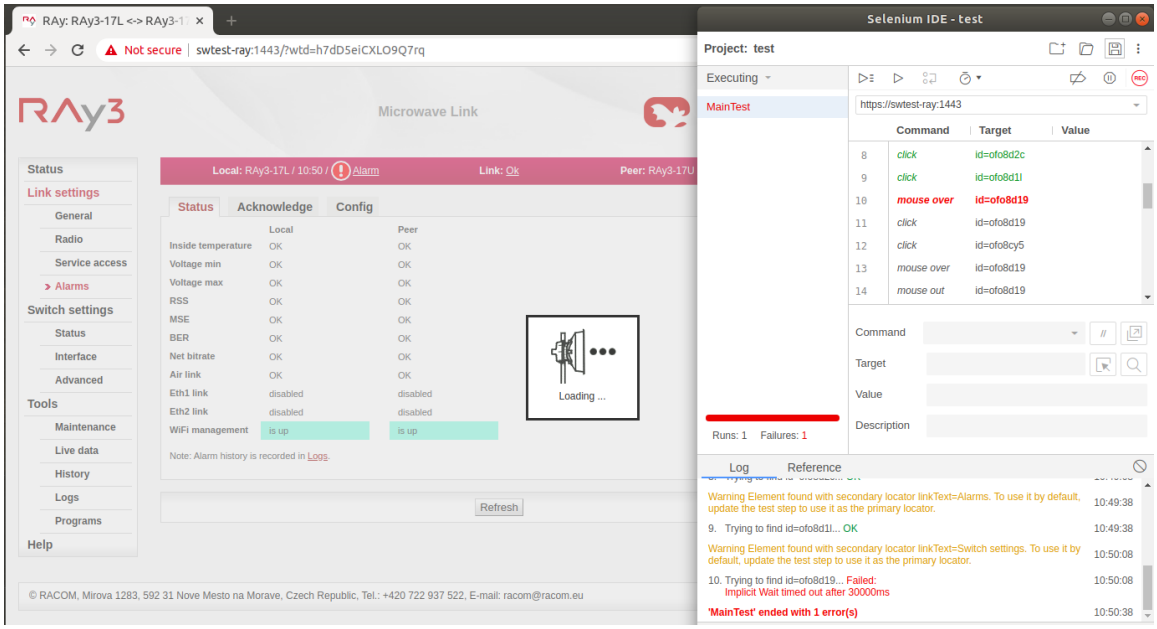
Selenium je open-source nástroj, který je určený k testování webů. Proto se dá použít i na testování webových aplikací. Je to asi nejrozšířenější nástroj pro tento problém a má proto velkou komunitu. Je vyvíjen v jazyce Java, ale použít se dá ve všech možných jazycích a je multiplatformní. Pro jeho zprovoznění musíte mít stáhlé prohlížečové drivery na stroji, kde se testy spouští. Tyto drivery se uloží do cesty v proměnné PATH, aby je Selenium lehce našlo.

4.1.1 Selenium IDE

Je to hodně jednoduchý a základní nástroj. Určený hlavně pro začínající programátory a manuální testery, aby si částečně ulehčili práci. Funguje na principu, že si tester stáhne stejnojmenný prohlížečový doplněk a ten ho provádí celým procesem. IDE recorder je v prohlížeči zapnut a snímá testerovy kroky na webových stránkách. Přesně ukládá činnosti, které dělá tester a skládá z nich automatický test. Poté se recorder ukončí a tester může celý nahraný test spustit. IDE podle nahraných kroků přesně zopakuje celý průběh.

Problém je, že když tester čeká, až se něco načte, recorder to nepozná, a tak se pravděpodobně poté vyskytne problém se spouštěním testů, kde může být průběh moc rychlý nebo jinak nepřesný a způsobí to neúspěch už i u menších automatizovaných testů. Řešením může být úprava nahraných kroků posloupnosti tak, že se jim přidá dodatečný čas, nebo můžeme dokonce manuálně přidat do průběhu další kroky, nicméně už to začíná postrádat nádech jednoduchosti a začíná to být z části programování. Ve výsledku, pokud požadujeme nějaký rozsáhlejší automat, se to stává spíše kontraproduktivní kvůli počtu změn, které musíme

udělat. Nicméně vývoj jde stále dopředu a možná se tento nástroj zanedlouho zlepší natolik, že už ho bude vhodné použít i ve větších a složitějších projektech.



Obrázek 4.1: Selenium IDE náhled

4.1.2 Selenium RC

Někdy se mu přezdívá webdriver 1.0. Selenium Remote Control je možné psát v mnoha jazycích. Například PHP, Ruby, Python, Java, .NET. Funguje na principu serveru a klienta. Server, který je napsán v Javě, komunikuje s klientem přes HTTP protokol. Klient může na serveru spouštět paralelní testy na mnoha různých prohlížečových driverech. [21]

Nyní se nepoužívá kvůli příchodu Selenium Webdriver.

4.1.3 Selenium Webdriver

Nástroj, kde se počítá s vývojářovou znalostí programování.

Je to framework neboli knihovna funkcí, které usnadňují přístup k webovým položkám a akcím. Je asi nejvíce rozšířený ze všech dostupných automatizačních nástrojů a má i nejvíce rozšířenou komunitu, což z něj dělá zajímavého kandidáta.

Funguje primárně na principu prohledávání elementů v DOM stromě, které můžou být označené mnoha jedinečnými způsoby. Např. pomocí identifikátoru, classname, XPath. A ty vyhledá a provede příslušnou akci.

Příklad akcí nad elementy:

- Klikni na element
- Počkej, až se element objeví na stránce
- Vyčti hodnotu elementu
- Zapiš do elementu

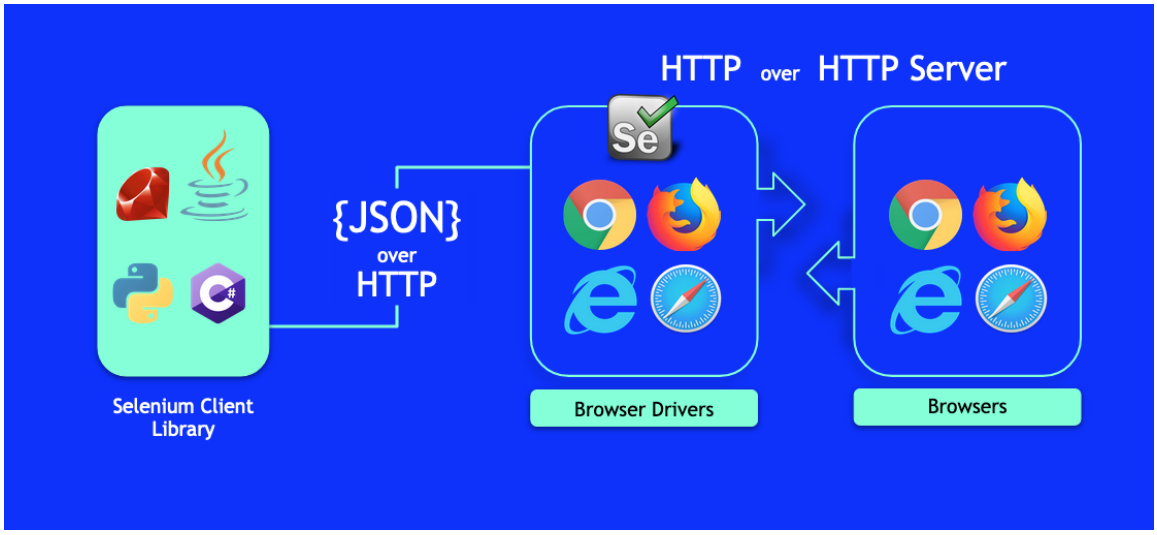
Obrázek 4.2: Architektura Selenium RC [4]

- a mnoho dalších

Nepracuje však pouze s elementy webové stránky, ale se vším možným, s čím se jako uživatelé na webu setkáváme. Je to například:

- Práce s alerty
- Spouštění Javascriptu
- Práce s okny prohlížeče
- Stisk kláves
- a další

Webdriver je možné psát v několika programovacích jazycích. Jako například Ruby, C# a mnoho dalších. Nejzajímavější mně však přišly Python a Java. Mají největší komunitu a také obsahují mnoho frameworků, které mohou pomoci ulehčit práci. U Javy se dá použít například JUnit, TestNG nebo RobotFramework. Oba nabízí pohodlnou a přehlednou možnost psaní testů. Python, jakožto programovací jazyk, nabírá s každým rokem na počtu uživatelů, takže je pravděpodobné, že se budou frameworky stále zdokonalovat a přidávat. Nicméně aktuálně jsou hodně zajímavé Unittest, Pytest a již zmiňovaný RobotFramework.



Obrázek 4.3: Architektura Selenium Webdriver [15]

RobotFramework

RobotFramework má svou vlastní jedinečnou syntaxi. Na rozdíl od ostatních frameworků, které jsou dělány formou knihoven, kde používáte klasickou skladbu jazyka, se zde musíte naučit tuto jeho syntaxi. Ta je hodně primitivní, což může být jak výhoda, tak nevýhoda. Používají se v něm vytvořené funkce, se kterými bohužel nejde moc flexibilně pracovat, ale jejich použití je jednoduché. Může se však spojit s klasickou syntaxí Pythonu nebo Javy. Ovšem poté kód nevypadá tolik přehledně. Další možností je spouštět klasicky psané programy a v nějaké míře spolupracovat s dalšími frameworky.

Převážně je RobotFramework kvůli jeho primitivnosti určen pro začátečníky. Ovšem díky propojování různých frameworků by se dal využít i u zkušenějších jedinců na nějaké základní části testů.

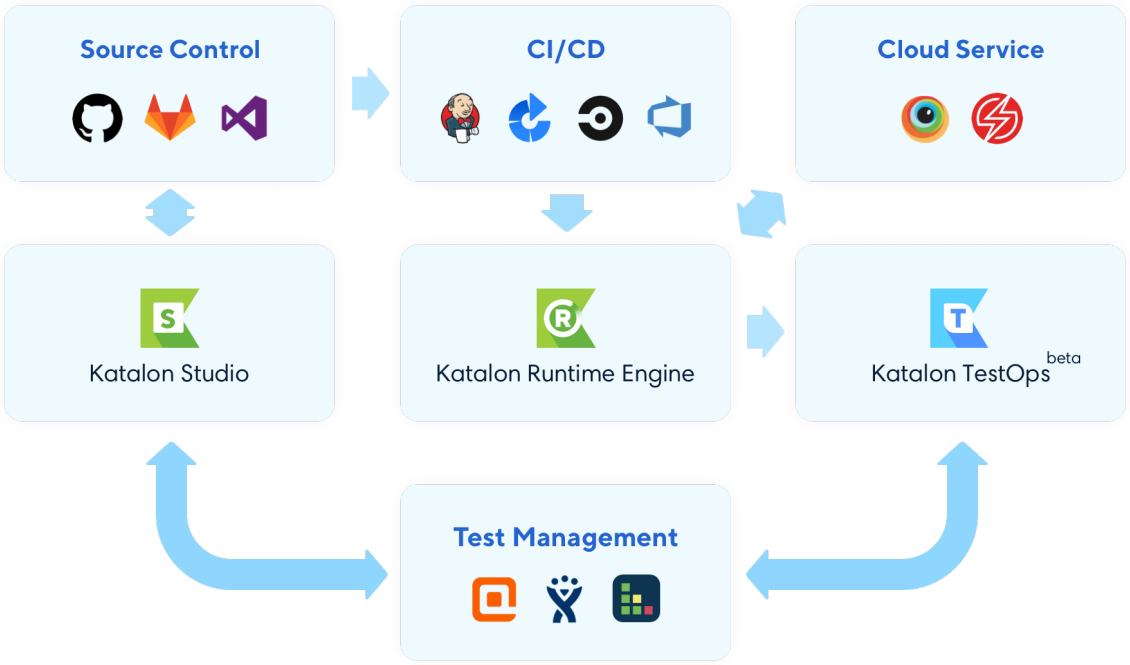
4.2 Katalon

Katalon je řada produktů určených opět pro automatizaci testování. Je poměrně nový a je založený a vystavěný na hodně rozšířených automatizačních knihovnách Selenium a Appium. Je zde pouze od roku 2015, ale má ambice se poučit z ostatních frameworků a vytvořit co možná nejužitečnější nástroj. V roce 2018 bylo zaznamenáno 9% pokrytí podílu na trhu s automatizací UI.[20] Vypadá to tedy, že by se mohlo jednat o zajímavý nástroj do budoucna. Již nyní ho používají velké společnosti jako je například Oracle. V současné době Katalon také pracuje na více typů nástrojů na testování přes IDE až po umělou inteligenci.

Díky Katalonu můžeme automatizovat webové rozhraní i mobilní a desktopové aplikace. [18] [11]

4.2.1 Katalon Studio

Je hlavní produkt z řady Katalon. Je podobný jako Selenium Webdriver. Pro ty méně technicky zdatné má grafické zobrazení a pro více zručnější uživatele je zde skriptování. Je tedy vhodné jak pro začátečníky, tak pro zkušenější programátory.



Obrázek 4.4: Myšlenka propojení katalon produktů [12]

Využívá PageObjectModel. Výhoda je, že grafické prvky aplikačního rozhraní se dají jednou nahrát a dá se s nimi pracovat i mezi různými testy. Pracuje s moderními webovými technologiemi jako je HTML, HTML5, Ajax, JavaScript a Angular. Vzdálené spouštění jde pomocí propojení s nástrojem Docker.

Katalon Studio by mělo být multiplatformní s podporou mobilních operačních systémů iOS i Android. Automatické testy by měly jít spustit na všech běžných prohlížečích, ale i na těch méně používaných jako Internet Explorer a Edge. Stejně jako v Seleniu je tu možnost spustit testy v headless módu, což je prohlížeč bez grafického rozhraní. Headless mód je dostupný v prohlížečích Chrome, Firefox a Safari. Dá se kombinovat i s automaty, které jsou napsané buď ve frameworku Selenium nebo Appium.

Katalon Studio je od začátku freeware s placenou podporou. Díky tomu můžou větší společnosti zaplatit za služby společnosti Katalon LLC a ta jim může poskytnout pomoc v případě problémů s jejich produkty anebo ve školení zaměstnanců. [12] [8]

4.2.2 Katalon Recorder

Funguje podobně jako Selenium IDE, ovšem spolupracuje s nástroji od Katalon LLC. Je to také rozšíření do prohlížečů, které nahrává uživatelovy kroky provedené na testovacím rozhraní a ty následně ukládá a staví z nich onen automatický test.

Bohužel na rozdíl od Katalon Studio nepodporuje Recorder většinu nejpoužívanějších prohlížečů, ale v této době slouží jako rozšíření pouze do Chrome a Firefoxu.

Výhoda je, že dokáže exportovat nahrané testy do jazyků C#, Java a Python. Pokud tedy programátor píše skript v některém z těchto jazyků, dá se propojit s Katalon Studio a ulehčit si tak práci. [10]

4.2.3 Katalon TestOps

Je webová cloudová aplikace sloužící k sbírání testovacích dat a provádění operací nad nimi. Zobrazuje data o testování. Dokáže pracovat s nástroji na zobrazení grafů a ostatní grafické vizualizace, aby byl výsledek testování co nejpřehlednější. Umožňuje testy monitorovat a zasílat upozornění na určité vzniklé situace. Také se pracuje na AI vyhodnocování testů.

Tento produkt se původně jmenoval Katalon Analytics. V roce 2019 se přejmenoval a je poměrně nový. Stále se vyvíjí a testuje, proto je dostupný pouze v Beta verzi. [14]

Měl by umět priorizovat testy a inteligentně je spouštět, aby se důležité testy dostaly na řadu nejdřív. Dále by měl v sobě mít efektivní plánovací mechanizmy, které zajistí maximální využití zdrojů při spouštění testů. Zvládá také paralelní testování.

Umí se spojit se softwary JIRA a Jenkins. Toto nabízí lepší přístupnost a přehlednost při zažitých nástrojích. [13]

Obrázek 4.5: Náhled produktu Katalon TestOps [13]

4.2.4 Katalium

Je to open-source framework pro tvorbu automatických testů. I když hlavní nástroj z Katalon rodiny je pořád Katalon Studio, Katalium je pro ty uživatele, kteří aktuálně vytvářejí testy na Seleniu nebo TestNG frameworku. Pro tyto uživatele by mělo Katalium představovat vhodný přechod do nástrojů rodiny Katalon.

Katalium má zachovávat funkce Selenia a TestNG, ale stavět framework více lehký a užitečný pro širší spektrum uživatelů. Má zajišťovat lepší učící křivku a hladký přechod na nástroje Katalonu dovolující spouštět testy rychleji a efektivněji. Spouštět se tu budou určitě dát i už hotové testy napsané v oněch dvou starších frameworcích.

Součástí Katalia je Katalium Server. Je to část, která má uživatelům nahradit Selenium Grid, což je jakýsi spouštěč testů nad různými prostředími, a zefektivnit tuto práci. [9]

4.3 Ranorex

Tento nástroj na automatizované testy je placený. Každý může využít 30denní zkušební verzi, ale poté už si za využívání musí připlatit. V nabídce Ranorex produktů je Ranorex Studio a Ranorex Webtestit. Placené jsou oba produkty s tím, že více využívaný a hlavní je Ranorex Studio. Hodně světových firem využívá produkty Ranorex k testování svých softwarů.

Tyto nástroje by měly být vhodné pro každého, jak pro začátečníka, tak pro pokročilé vývojáře automatizovaných testů. Umožňují testovat jak weby, tak desktopové i mobilní aplikace. Vývoj testů by měl být intuitivní. Měla by pomoct umělá inteligence. Dobře by mělo jít otestovat i grafické prvky uživatelského rozhraní. Opět je k dispozici Recorder, který může ušetřit hodně práce s hledáním a prací s položkami v rozhraní. [19]

Kapitola 5

Návrh řešení a architektura

Zde bude popsán návrh mého řešení. Požadavky, cíle výsledného programu, uživatelské a implementační vstupy, výstupy a architektura nástroje.

5.1 Požadavky a cíle

Cílem této práce je vytvořit funkční nástroj, který by automatizovaně a regresně testoval nové verze firmwaru stanice Ray3.

Nová verze vychází zhruba každý měsíc a manuálním testerům trvá asi 2 dny, než zkusí podle základního testovacího předpisu otestovat funkčnost aplikačního webového rozhraní a nejsou schopni vyzkoušet mnoho kombinací konfigurace, takže není jisté, že rozhraní funguje za všech okolností přesně tak, jak je žádoucí. Uživatelské rozhraní webové aplikace stanice Ray3 bylo vždy testováno pouze manuálně. Žádné robotické testy webu zde dosud nebyly.

Daný nástroj by tedy řešil problém jak času, tak i počtu vyzkoušených kombinací. Ušetřil by tedy testerovi základní práci a on by se mohl soustředit na otestování něčeho jiného. Například starých opravených chyb, kreativnějšího testování atd.

Automat by implementoval daný základní testovací předpis, který je poměrně neměnný. Mohl by běžet přes noc, dal by se spouštět paralelně a byl by přizpůsoben jak normálnímu běhu se stanicí připojenou lokálně, tak i na speciálním testovacím prostředí serveru.

Cílem je nástroj zprovoznit primárně na operačním systému Linux, který by měl být i hlavním vývojovým prostředím. Nicméně poté by měl být nástroj schopný běhu i na systému Windows, takže by měl být multiplatformní. Jsou dva hlavní prohlížeče, na kterých musí být rozhraní funkční: Firefox a Chrome. Automat by mělo být možno spouštět paralelně, aby se mohlo potenciálně otestovat více stanic v jednom čase.

5.2 Vstupy

Zde bude rozebráno, jaké vstupy žádá program z pohledu implementace a z pohledu uživatele.

Z pohledu implementace nás v návrhu zajímají hlavně použité knihovny, podle čeho bude program implementovaný a návrh, jak budou části provázané. Uživatelské vstupy řeší, jak bude uživatel program používat a co musí zadávat a konfigurovat.

5.2.1 Implementační vstup

Hlavními vstupy, co se týče implementace, jsou manuální testovací předpis a pomocný framework funkcí nad rozhraním.

Předpis

Vstupem programu by byl dříve řečený ověřený a jasně definovaný testovací předpis, který by sloužil jako předloha a vzor implementace. Testovací předpis je základním kamenem manuálního testování webového aplikačního rozhraní stanice Ray3. Je rozdělený na logické části podle rozdělení webového rozhraní. Testují se v něm tedy hlavní funkční základy.

Efektivní zde tedy bylo zautomatizovat daný testovací předpis, protože ten se zkouší pokaždé, když se testuje nová verze stanice. Zbytek už je na kreativitě a zkušenostech testera a tím pádem má proměnlivý průběh. Předpis nicméně zůstává převážně stejný.

Framework

Tím, že je to komplexní produkt, může se očekávat, že automatizace bude pokračovat i po prvním programu a proto jsem se rozhodl, že bude vhodné vytvořit i framework funkcí nad webovým rozhraním dané stanice.

Pomocí onoho frameworku se sestaví průběh testovacího předpisu, a z toho vznikne požadovaný automatický test, který bude v průběhu komunikovat se stanicí a bude zkoušet funkčnost rozhraní.

5.2.2 Uživatelské vstupy

Uživatel by ovládal nástroj klasickým způsobem, a to spuštěním programu z terminálu. Nastavoval by ho buď výčtem parametrů programu anebo pomocí konfiguračního souboru, kde by byla množina všech možných nastavovaných parametrů, pomocí nichž by spouštěl program podle svých potřeb a závislostí jeho systému.

5.3 Výstupy

Jako výstup bude sloužit hlavně výsledný report pro uživatele a dokumentace o implementaci předpisu.

Report

Jako výstup tohoto automatizačního nástroje by byl výsledný report o průběhu testu, jeho částech atd. Report by jasně a přehledně sděloval, které části provedeného testu jsou funkční, které nebyly provedené a proč a které neproběhly podle původního očekávání a kde byla změna nebo případná chyba.

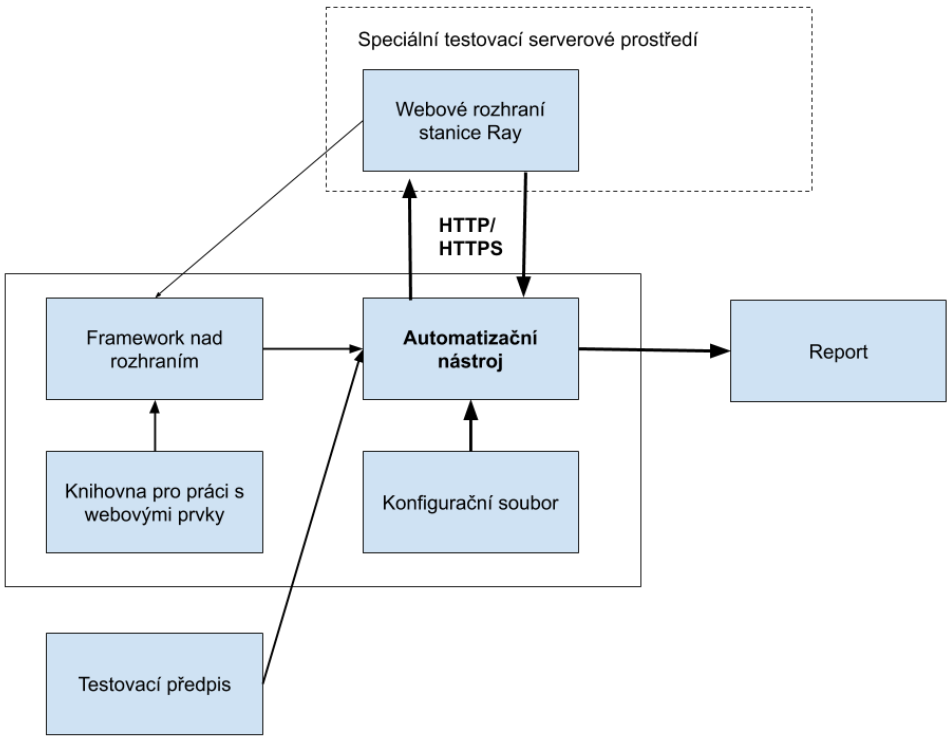
Dokumentace

Vedle celého programu by byla dokumentace ve formě testového předpisu automatu, kde by uživatel jasně viděl, jak program implementuje jednotlivé kroky manuálního předpisu a mohl odhadnout, co nástroj otestuje a co musí ještě dodělat manuálně. Také by z této dokumentace viděl, kde by mohly být potenciální hrozby a chyby automatizace anebo co

je ještě dobré dodělat. Také by zde bylo napsáno, jaké kroky nebyly implementovány a z jakých důvodů.

5.4 Architektura mého automatu

Zde budou detailněji rozebrány jednotlivé části architektury výsledného řešení a jejich návrhy a popisy.



Obrázek 5.1: Schéma mé architektury

Předpis

Aktuálně obsahuje nejzákladnější funkcionalitu a frekventovaně používané položky webu na všech stránkách. Je složen z 22 kroků podle logických a funkčních částí webové aplikace stanice Ray3. Uživatelské rozhraní je jinak velmi komplexní. Je zde mnoho propojených konfigurací a navazujících událostí.

Jsou zde přítomny testy:

- Změny různých konfigurací
- Nastavení uživatelských účtů
- Nastavení a připojení pomocí různých metod

- Fyzický kontakt se stanicí
- Dodržení stejné funkčnosti na více prohlížečích
- Obnovy konfigurací
- A mnoho dalšího

Je jasné, že všechny části předpisu nebude hned možné implementovat (fyzický kontakt, ..), ale bude to sloužit jako dobrá předloha.

Komunikace programu se stanicí

Komunikace bude probíhat přes HTTP nebo HTTPS protokol. Uživatel stanice si může vybrat, přes který protokol ke stanici přistoupí, a v programu to bude také zahrnuté ve výběru.

Konfigurační soubor

Tento soubor bude sloužit uživateli pro výběr různých parametrů. Parametry bude možné zadat i přímo při spouštění v terminálu. Požadované řešení je, aby si uživatel vytvořil pár různých konfiguračních souborů podle různých požadavků a poté si mezi nimi vybíral. Příklad: Uživatel bude chtít otestovat test na prohlížeči Firefox a Chrome. Pokud bude mít dvě stanice k dispozici, bude si moct vytvořit dva konfigurační soubory a s těmi spustit dva programy paralelně.

Výčet možností v konfiguračním souboru:

- Přístup do stanice (HTTP/HTTPS)
- Prohlížeč
- Umístění binárního souboru Firefoxu
- Číslo verze firmwaru a jeho umístění
- SSH klíč stanice a jeho umístění
- Adresa NTP serveru

Mimo konfigurační soubor bude nutné zadat IP adresu a případný port pro přístup k rozhraní. Dále bude možné si vybrat, které části testu chci pustit.

Report

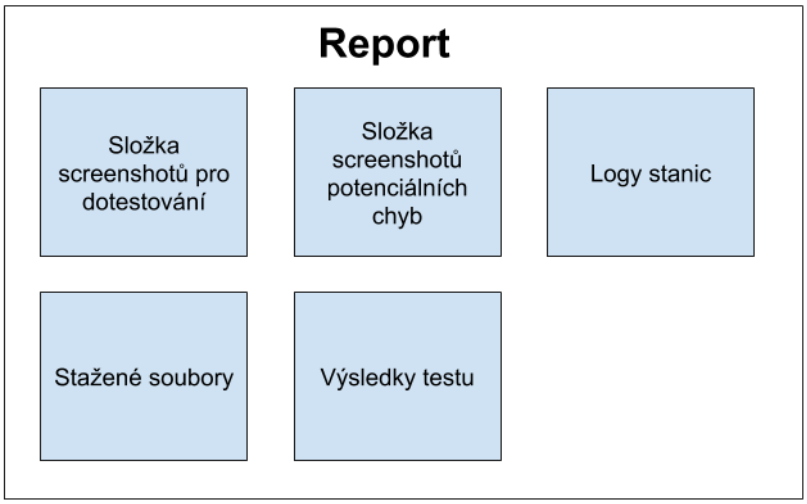
Výsledný program bude testerovi vhodným výstupem hlásit, které části předpisu fungují, které nefungují, které se provedly a které se neprovedly a proč. Cílem je, aby tester spustil program, který mu automaticky otestuje funkcionalitu podle zadaného předpisu a tester po ukončení testu viděl, kde mohou být problémové části a co je zapotřebí otestovat manuálně.

Výsledek se bude zapisovat v čase a přehledným způsobem do terminálu, ale detailnější výpis a další informace se budou ukládat do vytvoření složky reportu.

Report bude obsahovat části:

- Soubor s výsledkem testu

- Složku s pořízenými screenshoty pro otestování problematických částí
- Screenshoty potenciálních chyb testu
- Stáhlé logy ze stanic Ray3 pro kontrolu
- Stažené soubory ze stanice



Obrázek 5.2: Návrh reportu

Framework

Toto je hlavní část řešení, ze které se poté bude automatizovat výsledný testovací předpis a díky této části se v budoucnu budou moci efektivně vytvořit i další automatizační webové nástroje pro dané rozhraní za krátký čas.

Pomocí nějaké knihovny, která usnadňuje komunikaci s webovými prvky, se vytvoří framework nad rozhraním stanice. Bude rozdělen do částí podle jednotlivých stránek rozhraní. Pro každou stránku bude obsahovat množinu funkcí pro práci nad prvky dané stránky, které budou získávat údaje z jednotlivých elementů, budou nad nimi spouštět různé operace a pracovat s jejich daty.

Testovací prostředí

V plánu je spouštět program kdekoli, ale přímo ve firmě Racom vzniklo testovací serverové prostředí, kde se test bude hlavně spouštět. Prostředí je linuxový server s operačním systémem CentOS s grafickým uživatelským rozhraním. Testeři zde budou mít své účty a přístup do množiny nástrojů.

Zde budou stanice připojeny přes switch. Všechny stanice budou mít stejné adresy a přistupovat se na ně bude přes porty.

Komunikace s testery

Při vymýšlení architektury a vůbec plánování návrhu mého automatu jsem byl v kontaktu s týmem manuálních testerů a se samotnými vývojáři stanice Ray3, se kterými jsem konzultoval postup, průběh a implementaci kroků.

S testery jsem primárně řešil skladbu funkcí mého automatu, priority jednotlivých kroků a tak dále. Je to přece jen nástroj pro samotné testery, takže mi přijde důležité se s nimi o tom bavit a vytvářet to na míru testerům. V tomto případě jsou něco jako moji zákazníci. Řešil jsem s nimi dodatečné funkce, styl výstupů a funkčnost a zapojení v jejich testovacím prostředí.

Aktualizace

Počítá se s tím, že při vývoji rozhraní stanice Ray3 se bude muset průběžně aktualizovat i daný framework funkcí a že se bude měnit i testovací předpis, takže se bude aktualizovat i samotný automatický test.

Kapitola 6

Implementace automatického testu

Celá práce byla vytvářená na operačním systému Linux Ubuntu. Všechny použité nástroje by ale měly být multiplatformní, takže by se to mělo bez větších problémů zvládnout vyvíjet i používat i na ostatních systémech jako je Windows nebo MacOS.

6.1 Webové rozhraní stanice a práce s elementy

Webové uživatelské rozhraní stanice Ray3 má prvky normálního webového rozhraní, je tu ovšem jeden automatizační problém. Drtivá většina elementů má automaticky generované atributy jako ID a class. Mají tedy unikátní označení, ale pouze v rámci jedné session. Tzn. že po připojení na rozhraní se atributy vygenerují. Když se otevře nové okno nebo záložka prohlížeče a připojí se na tu samou stanici, bude mít toto rozhraní zase jiné hodnoty atributů.

Nejvíce se generují automatická ID, kde je hodnota atributu sedmimístné náhodné spojení znaků písmen a číslic.

Jedinečný přístup k elementům

Z tohoto důvodu se nedá přistupovat k elementům pomocí atributů, ale byl využit jazyk XPath, kde se používá absolutní cesta k elementu.

K nalezení a popsání XPathu elementu bylo použito rozšíření ChroPath v prohlížečích Chrome a Firefox. Rozšíření funguje tak, že se v prohlížeči na stránce ve vývojářských nástrojích najde element a rozšíření ChroPath nabídne různé popisy podle jedinečnosti přístupu k němu.

6.2 Automatizační nástroj

Bylo vyzkoušeno mnoho nástrojů a jejich variant. Nejvíc v úvahu připadaly ale Selenium a Katalon. Selenium má nejpočetnější komunitu a je nejrozšířenější testovací webový automatizační nástroj. Katalon funguje podobně, ale má ambice být modernější a řešit nedokonalosti o poznání staršího Selenia. Navíc jsou obě open-source.

IDE

Obě mají IDE rozšíření do prohlížeče a možnost nahrávání kroků, což by mohlo být skvělé ulehčení práce. Nicméně IDE nefungují tak dobře a mají hodně nedokonalostí, kvůli kterým

se musí následně opravit i krátký kód. IDE ale připadají v úvahu hlavně z důvodu, že by si program mohli potenciálně lehce dovyvíjet i manuální testeři, kteří se tolik nevyznají v programování. Bohužel ale IDE bylo vyřazeno z výběru, protože by mohlo udělat jen jakousi chatrnou kostru, která by se poté musela opravovat.

IDE často neřešily čekání na elementy na stránkách, nebo měly čekání špatně navržené, proto testy často padaly. Dále také slouží jen jako test webu. Jakmile je potřeba pracovat se soubory atd., bude se kód muset opravit.

Nicméně výhodou je, že se dá jejich kód exportovat. Může se tedy takto potenciálně ulehčit práce, nebo navést nezkušenější programátory a testery. Také se mi líbí, že se dá zvolit, podle čeho bude hledat elementy na stránce (ID, Class, XPath, ...)

IDE jsem ale v práci nepoužil.

Katalon Studio vs. Selenium Webdriver

Po vyřazení IDE zbývají dva hlavní konkurenční nástroje obou frameworků, a to jsou Selenium Webdriver a Katalon Studio.

Jsou to v mnohém hodně podobné nástroje. Srovnával jsem funkčnost obou dvou a v Seleniu nebyly větší problémy s funkčností základních testů, ale Katalon z různých důvodů padal a vyskytly se problémy i v základních příkazech, jako čekání na elementy atd.

Věřím, že to jsou pro Katalon začátky a že časem se vše vyladí a bude fungovat správně ve všech prostředích a případech, což z něho může udělat ještě lepší nástroj, než je Selenium. Ale z důvodu větší jistoty jsem si pro svoji práci zvolil právě Selenium.

6.3 Jazyk a framework

Selenium má podporu mnoha jazyků. Nejzajímavější mně ovšem přišly Python a Java. Oba se řadí mezi nejpoužívanější a nejrozšířenější jazyky světa a mají pro to své důvody. Oba mají širokou komunitu. Oba dva mají mnoho možných frameworků na podporu psaní testovacích programů nebo jednotkových testů.

Nakonec jsem si vybral Python. Má jednoduchou, ale účinnou syntaxi a tento jazyk teď zažívá největší rozkvět a rychle se probojovává na přední příčku v používanosti a díky tomu na jeho bázi vzniká mnoho podpůrných a doplňujících frameworků a knihoven, a to se v budoucnu může hodit. Python nabízí obrovskou škálu možností, je intuitivní a neustále se rychle vyvíjí.

Framework

U Pythonu jsem volil mezi frameworky Unittest, Pytest a Robot Framework. Robot Framework hodně vybočuje. Má svou vlastní syntaxi, která připomíná assembler, a výhoda je, že je jednoduchá, ale na druhou stranu poměrně omezená. Samozřejmě se dá syntaxe spojit i s normální syntaxí Pythonu a ostatními frameworky, ale dělat něco na dvě půlky a pak to spojovat dohromady mi nepřijde výhodné, takže RobotFramework jsem vypustil.

Na výběr byl tedy Unittest a Pytest. Oba dva se využívají pro psaní jednotkových testů. Pracují se stejnou klasickou syntaxí Pythonu a knihovny Selenia, jen nabízí škálovatelnost kroků testů, automaticky zajišťují výstupy testů, mají mnoho rozšiřujících možností a celkově se v nich automatické testy píší lehce a přehledně. Oba umí poskládat kroky testu do různých pořadí podle určitých předpisů.

Unittest je starší a slouží již dlouhou dobu hlavně pro jednotkové testování. Pytest je novější, modernější a má obrovskou škálu možností. Aktuálně na něj začínají přecházet velké společnosti.

Prvně jsem zkoušel Unittest, který mi přišel skvělý a plně dostačující pro ulehčení práce a přehlednosti kódu. Nicméně jsem poté vyzkoušel Pytest a zjistil jsem, že on nabízí v základu o mnoho více možností, ač zatím tedy není tak moc rozšířený.

Zvolil jsem si tedy jazyk Python s frameworkem Pytest.

6.4 Struktura testu

Celý test je škálovaný podle kroků reálného předpisu. Krok předpisu většinou odpovídá třídě v programu, který je dále rozdělený na podkroky (funkce v programu). Forma názvů je test_(číslo kroku)_(název kroku).

Občas se nějaké kroky buď sloučily, nebo rozdělily oproti původnímu předpisu. Je to kvůli skutečnosti, že princip manuálního a automatizovaného testování může být trochu jiný.

Některé části nejsou automatizované. Stalo se tak převážně kvůli ať časově, nebo proveditelně náročnému řešení. Například test fyzických tlačítek, které nejsou přímo součástí webového rozhraní, nicméně se v testu objevují, protože mají s rozhraním funkční spojitost. Tyto části nejsou ve výsledném programu implementované, nicméně jsou zde dané kroky zahrnuty a přeskočeny. Tato skutečnost je výhodná pro testery, kteří budou test reálně využívat, protože ve výsledném reportu vidí, které kroky byly přeskočeny a musí se tedy dodatečně vyzkoušet.

6.5 Struktura frameworku

Framework funkcí nad rozhraním byl rozdělen mezi soubory, kde každý soubor náleží jedné stránce hlavního menu rozhraní. Pro každou takovou stránku byly pak vytvořeny funkce, které pracují s danými elementy oné stránky a které se následně volají v hlavním programu, který je složen pomocí tohoto frameworku.

Jak již bylo řečeno, toto řešení umožňuje programátorovi pouze jednou vytvořit daný framework a pak z něj čerpat ve více výsledných programech.

6.6 Vstupy automatu

U parametrů při spouštění programu uživatel musí zadat IP adresy cílových testovaných stanic. Ve speciálním testovacím prostředí jsou stanice připojeny přes switch, mají stejné přístupové adresy a jen se jim mění porty podle pořadí. To program řeší tak, že stačí místo IP adres napsat pořadí stanice a test se sám připojí.

Pytest parametry

Samotná knihovna Pytest, přes kterou je program spouštěn, má hodně možností výpisu a běhu. Za zmínku zde stojí hlavně výběr jen určitých částí kroků testu. Uživatel tedy nemusí pokaždé spustit celý test, ale může si vybrat jen část, což je hodně výhodné.

Další je parametr na výběr chybového výpisu. Jde si zvolit, jestli psát velmi detailní chybový výpis, nebo například jen stručný. To se muže lišit pro uživatele nebo vývojáře.

Konfigurační soubor

Jako vstup parametrů zde hlavně slouží konfigurační soubor popsaný v návrhu. Je dělaný pomocí struktury JSON, aby se v něm dobře hledalo. Zadávají se zde cesty k důležitým souborům jako přístupové SSH klíče atd. Vše se vybírá textovou podobou, proto jsou v souboru komentáře s nápovědou.

Všechny položky, u kterých to má význam, se testují na správnost výběru. Například jestli uživatel vybral některý z možných prohlížečů nebo jestli existují cesty k souborům.

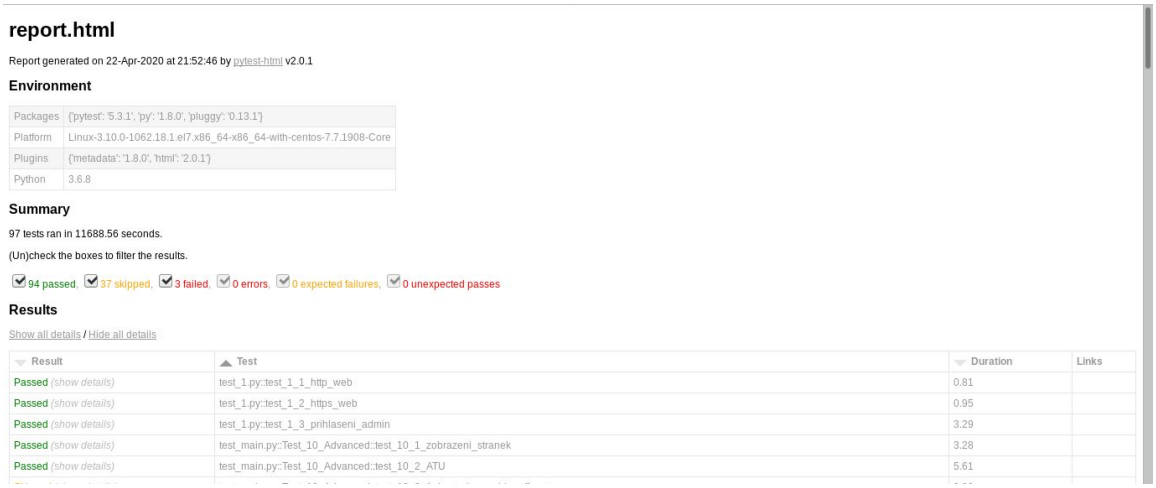
Konfigurační soubor se buď vybírá defaultně, nebo si uživatel v parametru při spouštění programu vybere, jaký chce, aby se umožnilo paralelnímu testování a také se nemusel pořád upravovat jeden.

6.7 Výstupy automatu

Jako výstup zde slouží výpis přímo do terminálu, který zajišťuje knihovna Pytest sama a má také pár možností. Například jak moc chceme mít výpis kroků testu detailní atd.

Jinak se hlavní výsledek ukládá do reportu podle času spuštění testu a IP (portu) stanice, aby mohlo dojít k paralelnímu spouštění a uživatel rychle našel, který test hledá.

Hlavní výsledek programu je také uložen v dalším html souboru, který je vytvořen knihovnou Pytest-html a poskytuje dobrý filtr kroků programu a jejich výpis.



Obrázek 6.1: Report výsledku testu

Screenshoty

Pořizování screenshotů se implementovalo také pomocí knihovny Pytest a ukládají se buď pro následné ruční zkontrolování obrázků, nebo když vznikne někde nepředpokládaná situace, aby tester nebo vývojář poté věděl, v jakém stavu byla stanice, když nastala ona situace.

Logy stanic

Logy se stahují pomocí již existujícího programu testera Petra Juráka a ukládají se při různých příležitostech, aby tester mohl lépe najít příčinu chyby stanice.

6.8 Hledání elementů

Hledaní elementů na stanicích bylo trochu problematičtější. Použil jsem na to knihovnu Selenium, která umí najít element podle mnoha faktorů, bohužel stanice Ray3 si vytváří automaticky ID s každým načtením session, takže hledání podle ID většinou nebylo možné. Hledání bylo uskutečněno nakonec pomocí absolutní XPath, protože nebyly jiné možnosti. Toto je nepříjemná cesta, když vychází nové verze rozhraní a změní se nějaká položka blízko kořene stromu.

6.9 Čekání na načtení elementů

Při průchodu webu nebo například při změně různých elementů se postupně načítají další části webu (tzn. elementy), se kterými můžeme chtít pracovat.

Čekání na načtený element je možné dvěma způsoby: absolutní čekání a dynamické čekání. Při absolutním čekání se vybere určitý počet sekund a po ty program čeká, než udělá další úkon. Dynamické čekání čeká na nějakou událost maximální počet sekund, který mu vývojář zadá, a pokud událost nastane dříve, čekání se ukončí a je proveden další úkon.

Bylo také vypozorováno, že pokud se spustí více testů naráz, nebo programy nemají tolik volných prostředků, tak se odezva a komunikace programu se stanicí o hodně zpomalí. Proto nemůžeme vědět, jak dlouho načítání přesně bude trvat.

Z důvodu postupného načítání a proměnlivého času načtení bylo použito dynamické čekání. Implementováno bylo pomocí knihovny WebDriverWait od Selenia. Je zde na výběr mnoho událostí, na které se má čekat. V programu se většinou používá čekání, dokud daný element není viditelný. Je použit i reverzní přístup. Například program čeká, dokud načítací logo nezmizí.

6.10 Změna položek

Jedna z nejčastějších akcí nad elementy je jejich změna. Tím, že je několik typů elementů (Inputbox, Select, Checkbox, ...), se nedá říct jeden způsob, protože se se všemi pracuje jinak.

Když se element našel, je použita knihovna Selenia, která umí se všemi typy elementů pracovat. U primitivnějších elementů jako například checkboxy jsou jenom udělané funkce, které nastaví checkbox a zkontrolují.

Ale například u Selectu je možností mnoho a Selenium má na to další knihovnu Select, která má mnoho možností usnadňujících práci. U Selectu je taky důležité vybírat jen položky, které jsou povolené. Tuto selekci umožňuje výběr podle atributů class.

6.11 Kontrola položek

Další častá akce nad elementy je vyčítání jejich obsahu. U primitivnějších (checkbox, input, ..) se to dělá pomocí Selenia jednoduše, ale u jiných (Select, ..) to může být trochu složitější.

Z elementu se dá zjistit spoustu věcí. Nejčastěji zjišťuji text, ale také i hodnotu atributu "value", co se hodí například u Selectu.

Po vyčtení obsahu se poté kontroluje podle očekávaných hodnot. Využívá se například pro zjištění, jestli jsou stanice spojené.

6.12 Konfigurace

Na rozhraní se většinou provádí nastavování různých konfigurací a jejich následná kontrola.

Zmínil bych tu konfiguraci rádia a kontrolu, jestli se stanice po přenastavení spojí. Jelikož chceme vyzkoušet mnoho kombinací, jsou zde v cyklech změny mnoha konfiguračních položek. Vždy se rádio nastaví, zkontroluje se uložení, zkontroluje se správné zobrazení položek radia a poté správné údaje na status stránce. Při nastavování sleduje spojení stanic a případně provede spojení sám změnou konfigurace.

6.13 Obnova konfiguračních souborů

V rozhraní se dají zálohovat různé konfigurace do souboru. Ten se stáhne, vyjme se pomocí data a názvu a vloží do reportu daného testu. Odtud se poté použije při obnově konfigurace a poté se zkontrolují položky webu.

Pouze při použití Firefoxu jsou problémy se stáhnutím. Současný driver Firefoxu z nějakých důvodů nezvládá nastavení prohlížeče a stáhnutí souborů.

6.14 Různé přístupy do stanice

Do stanice se mimo webové rozhraní může přistoupit i přes telnet nebo ssh. V rozhraní jsou přepínače všech těchto přístupů. Test zkouší vypnout přístup přes rozhraní všech těchto metod a poté je pomocí patřičných knihoven otestuje. Poté opět zapne a znovu otestuje. Testuje se i přístup přes webové rozhraní u druhé stanice.

6.15 Změny uživatelů

Přístup ke stanici je možný přes účet. Podle toho vám jsou i přidělena různá práva. Test tedy zkusí vytvořit různé typy uživatelů a u nich poté zkouší, jestli mají zobrazeno to, co mají mít, jestli se jde připojit k webovému rozhraní, jestli funguje přístup přes klíč atd.

6.16 Drivery

Jelikož jsou požadované dva prohlížeče Chrome a Firefox, je test udělaný jen pro tyto dvě volby.

Pro komunikaci s prohlížeči je nutné mít jejich drivery. Driver je jakýsi prostředník, který spojuje WebDriver v testu s daným prohlížečem přes proxy. Driver si může každý stáhnout, protože je to open-source. Driver chromu se jmenuje Chromedriver a Firefoxu se jmenuje Geckodriver. Umístění driverů je v linuxové proměnné PATH, kde ji WebDriver najde. Driver se vždy automaticky spojí s prohlížečem a jeho verzí. Pokud se samotný prohlížeč aktualizuje na novou verzi, driver se spouští automaticky s novou verzí.

Problém je, když je více verzí prohlížeče. Proto je v konfiguračním souboru možnost vybrat si umístění verze, aby si popřípadě mohl uživatel zvolit testovanou verzi prohlížeče.

6.17 Paralelní testování

Testy lze spouštět paralelně, jen je nutné mít stanici pro každý test. Proto je vytvořeno testovací prostředí, kde může být několik párů stanic různých typů, které se dají otestovat paralelně za krátký čas hned po vydání nové verze firmwaru.

Jak už bylo řečeno, pro každý test je jeden konfigurační soubor a report se ukládá podle jedinečného jména času a adresy stanice.

Problém může vzniknout při velké zátěži v prostředí. Testy se o hodně zpomalují, takže je dobré počítat s touto situací a dávat větší prostor na čekání na element.

Kapitola 7

Testování

Zde se budu zabývat testováním a ověřováním požadované funkčnosti výsledného řešení. I programy určené k testování softwaru se musí samozřejmě nějak testovat, aby se potvrdila jejich funkčnost. Jen se často netestují stejným způsobem, jako ostatní druhy softwaru. Tím, že toto výsledné řešení jsou jakési jednotkové testy, by mohlo být cyklické vytvářet pro ověření tohoto řešení další jednotkové testy. Vznikl by tím jakýsi "test na test". A takto by to mohlo vznikat teoreticky až do nekonečna.

Další věc je také ta, že tento testovací program není určen pro koncového uživatele, ale jen pro testera nebo vývojáře daného vyvíjeného softwaru. Tím pádem má jen úzké zaměření a využití. Určitě od něj ale očekáváme spolehlivost.

Také s tím souvisí, že samotný jednotkový test nemá tolik možností konfigurace a volby. Je proto otestování jeho správné funkčnosti jednodušší. Jedno jeho spuštění je také velmi dlouhý proces, který zabere několik jednotek hodin.

Výsledný program jsem testoval hlavně sám, protože jsem ho vyvíjel, nicméně mi pomáhali v testování i manuální testeři firmy Racom.

Podařilo se dokončit požadovanou funkčnost programu a bylo to i otestováno několika stranami.

7.1 Testování během vývoje

Jak bylo zmíněno, většinu jsem si testoval sám. K testování bylo samozřejmě potřeba být spojen se stanicemi pro připojení se k rozhraní. Program byl psán v jednotkovém frameworku Pytest, který má možnost chybového výpisu v případě neočekávaných událostí, respektive chybám.

Tím, že je daný program posloupnost jednotkových testů a přímo pomocí nich testuje dané zařízení, se nad ním neprovádí moc konfigurací a průběh je povětšinou stejný. Toto napomáhá k tomu, že se během pár běhů programu mohou zkusit všechny možnosti spuštění. Tím pádem by nemělo nastat nějaké větší množství skrytých chyb.

7.1.1 Přístup ke stanici Ray3

Jak již bylo řečeno, pro spuštění programu je potřeba být spojen s testovanou stanicí, aby mohla probíhat komunikace s rozhraním.

Proto jsem po celou dobu měl k jedné nebo více stanicím přístup. Buď jsem měl stanice fyzicky lokálně spojené, anebo byl přístup přes speciální serverové testovací prostředí.

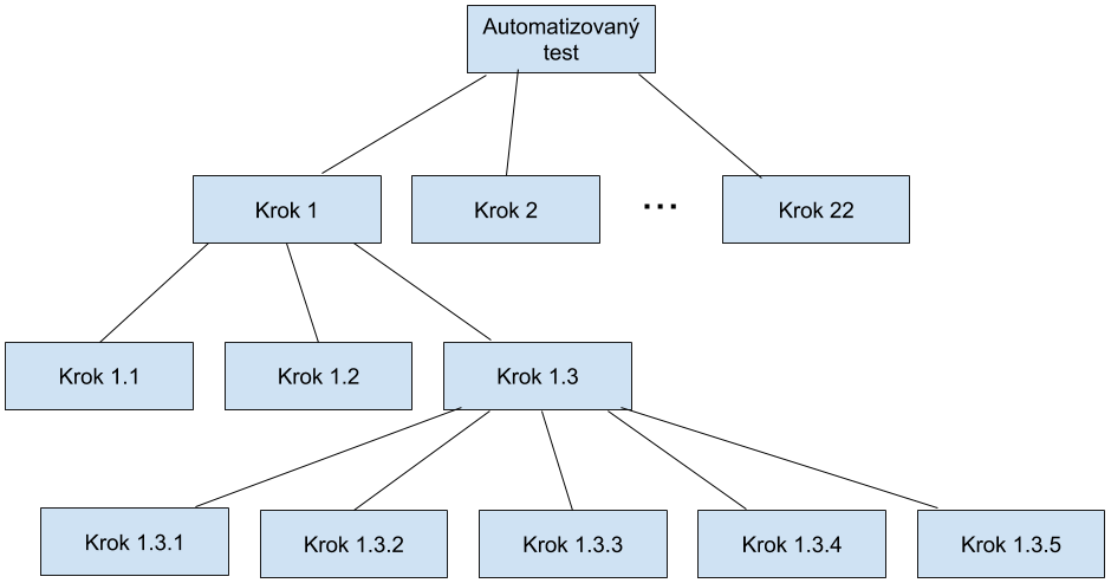
Oba dva přístupy se ukázaly jako odlišně funkční. Je to pravděpodobně kvůli časové prodlevě u jednotlivých typů spojení. Dá se čekat, že lokálně připojené stanice budou komunikovat rychleji než stanice připojené v jiné síti. To mělo výsledný vliv na odezvu a manipulaci s rozhraním.

7.1.2 Pytest

Chyby programu byly odchytávány pomocí chybových hlášení nástroje Pytest. Zde lze nastavit, jak detailní zpráva o chybě nebo neočekávaném stavu má být. Pytest tedy ke každému kroku, co je "FAILED", vypíše do výsledného reportu důvod a pozici v kódu.

7.1.3 Postupné testování

Program byl vyvíjen postupně po částech. Výsledný test má 22 kroků, které jsou většinou složeny z několika menších kroků druhého řádu. Ty pak mohou obsahovat další dílčí kroky, což jsou kroky třetího řádu. Takto vzniká strom, který se nepravidelně větví.



Obrázek 7.1: Strom kroků

Testovalo se rovnou při implementaci jednotlivých částí.

Při testování se tedy spustila vždy nejmenší samostatná část testu, což mohl být jeden krok třetího řádu. Když byl daný krok v pořádku, pokračovalo se jeho sourozencem. Poté, co byly správně implementováni a otestováni všichni sourozenci, mohl se otestovat jejich otec (Krok druhého řádu). Takto se postupovalo, dokud se nedostalo do fáze otestování celého programu.

V případě pozdějšího nalezení chyby se tento postup znovu uplatnil od postiženého místa.

Tímto efektivním způsobem se mohly začít testovat malé části a pomocí rychlého odladování zajistit správnou funkčnost. Nejmenší krok trval většinou v řádu desítek vteřin, takže se bylo možné i podívat na grafické zpracování a výslednou reálnou činnost.

7.1.4 Čas

Program pracuje s časovými údaji. Konkrétně s časem nastaveném ve stanici a s časem systému, na kterém program běží. Od toho se muselo odvíjet i testování.

Rizikové mohlo být spouštění přes půlnoc. I když se používaly datumové knihovny, může se stát, že když se některý krok pracující s časovými údaji spustí v jeden den a dokončí se druhý den, mohl by nastat problém. Další riziko bylo při neaktuálním času nastaveném ve stanici. Stanice vytváří záznamy a soubory v závislosti na čase a program tyto časové údaje poté porovnává.

Zkoušelo se tedy program spustit v různé časy před půlnocí, aby se mohla projevit případná chyba. Dále se zkoušelo nastavit neaktuální čas na stanici a spustit různé kombinace kroků testu, které pracují s časem.

7.1.5 Prohlížeče

Chybovost programu mohou jistou mírou ovlivnit i jednotlivé prohlížeče, proto se test musel zkoušet na Chrome i na prohlížeči Firefox.

Takto byly zkoušené i ostatní možné uživatelské konfigurace. Nicméně už málo z nich měly na výsledný běh velký vliv.

7.1.6 Snímky obrazovky

Hodně přínosná funkce při testování onoho testovacího programu byly vytvářené screenshoty při chybách. Tyto snímky vznikaly vždy, když nastala v běhu programu nečekaná situace. To se ukázalo přínosné jak u spouštění programu v reálném použití, tak při kontrole funkčnosti všech kroků.

Sám Pytest už poměrně přesně popíše, kde a proč chyba vznikla, nicméně neví nic o výsledném rozhraní a k tomu právě sloužily tyto snímky. Šlo o to, že když vám například Pytest vypsal, že nemohl najít nějaký element na stránce, i když podle testu tam měl být, nemusí to vývojáři testu nic říct. Když je ale v tomto případě použit screenshot, jde lehce poznat, v kterém stavu se rozhraní nacházelo, a tím i mnohem lépe určit příčinu problému.

7.1.7 Git

Po celou dobu jsem pracoval s verzovacím systémem Bitbucket. Po implementování a otestování nové verze zde byly vloženy patřičné soubory. Bylo použito i větvení, kde v hlavní větvi byl funkční kód programu a v developerské větvi probíhal vývoj testu.

Hojně používanou funkcí gitu byly issues. Zde jsem zapisoval všechny problémy, nápady a vylepšení, které mě napadly, a dále s nimi pracoval. Více o tom v kapitole 7.2.3.

7.2 Praktické nasazení

Tento projekt byl už od začátku navrhován pro to, že se bude reálně používat v praxi. Plán byl tento automat používat po každé nové verzi firmwaru, aby to ulehčilo práci manuálním testerům, kteří za kvalitu produktu zodpovídají, a aby se zvýšilo množství otestovaných položek a kombinací nad nimi.

Pro ověření mého řešení v praktickém nasazení jsem se obrátil na zaměstnance firmy Racom, která se zabývá vývojem zařízení popsaného v kapitole 2.3. Spolupracoval jsem tedy se dvěma testery a s vývojářem onoho webového rozhraní.

S vývojářem jsem řešil funkční záležitosti v případě nějakých chyb nebo nečekaných vzniklých situací jak na straně mého programu, tak na straně webového rozhraní. Pomohl mi, co se týče funkčnosti onoho rozhraní, aby se mi lépe chápalo a testovalo.

Testeři mi pomáhali testovat jednotlivé funkčnosti a opravené chyby programu. Také mi dávali zpětnou vazbu na chod a zpracování programu.

7.2.1 Komunikace

Komunikace s testery probíhala nejvíce přes Issuetracker v Bitbucketu. Zde se objevoval každý významnější problém či nápad na vylepšení daného automatu jak ze strany testerů, tak ode mě. O detailech funkčnosti, výskytu menších chyb atd. se diskutovalo emailem.

Detailnější postupy a návrhy se řešily buď po telefonu, nebo osobní schůzkou.

7.2.2 Testovací prostředí

Firma Racom pro spouštění a testování tohoto automatu vytvořila ono speciální prostředí, které stačilo jednou zinicializovat, aby tam byly dostupné všechny potřebné nástroje a poté se tam všechny zúčastněné strany vzdáleně přihlašovaly a mohly se stanicemi efektivně a ve stejném prostředí pracovat.

Toto se ukázalo jako dobré řešení, protože byla velká pravděpodobnost, že každý, kdo bude chtít, daný program hned správně spustí, aniž by musel cokoli inicializovat.

Samozřejmě se ale musely testy provádět i na lokálně fyzicky připojených stanicích, protože chybovost mohla být jiná než ve vzdáleném prostředí.

7.2.3 Systém pro sledování chyb

Pro přehled chyb, vylepšení nebo návrhů a jejich postupu se používal Issuetracker od Bitbucketu. Záznamy jsem vytvářel já spolu s testery. Postupně se vytvářely tyto záznamy, které jsem přebíral od nejnaléhavějších po nejméně naléhavé. Issue, na kterém jsem pracoval, jsem otevřel, aby bylo vidět, že je v procesu řešení. K danému záznamu se psaly komentáře od všech zainteresovaných účastníků, probíraly se návrhy, zpětná vazba atd.

Poté co se záznam o nějaké události vyřešil a já jsem ho otestoval, uložil jsem opravu na git, uvědomil testery, kteří mi issue zkontrolovali, a pokud bylo vše správně, issue se zavřelo a pokud ne, tento opravovací proces začal znovu.

Kapitola 8

Závěr

V rámci této práce jsem nejprve nastudoval oblast uživatelských aplikačních webových rozhraní v zabudovaných systémech. Popsal jsem hlavně způsoby, jak rozhraní fungují, jaké mají architektury, ale zaměřil jsem se i na možnosti, jak je tvořit a jaké programovací jazyky se u jejich vývoje vyskytují. Jedno z klíčových témat zde byla práce s elementy webové stránky, protože to je pro výsledné řešení klíčové. Také jsem si zajistil stanici s připraveným rozhraním pro vývoj programu. Tato stanice neměla pevně dané jedinečné atributy svých elementů v rozhraní, proto se musel v řešení použít jazyk XPath pro práci s elementy. Toto není z pohledu automatických testů ideální výchozí situace.

Dále jsem se seznámil s obecným testováním aplikačních rozhraní zabudovaných systémů. Popsal jsem zde jednotlivé způsoby jak manuálního, tak automatizovaného testování a jednotlivé přínosy, výhody a nevýhody těchto způsobů a typické případy použití. K tomu jsem popsal nejpoužívanější metody testování webových rozhraní, a to funkční a nefunkční testování.

Poté jsem prostudoval možné nástroje, které se na tvorbu automatizovaných testů nad webem nejčastěji používají. Nejvíce mě zaujal Selenium Webdriver kvůli široké podpoře a letech dominance v této oblasti. Tento nástroj jsem poté použil i ve vývoji výsledného programu. Nicméně určitě stojí za zmínku i ostatní uvedené nástroje, a to jak pro rychlý nárůst v posledních letech, tak kvůli možnosti propojení automatizace s umělou inteligencí.

V rámci této bakalářské práce jsem vytvořil jak obecný framework nad daným rozhraním, tak i test, který tento framework využívá. Framework má sloužit pro lehký vývoj automatizovaných testů nad rozhraním stanice Ray3 a konkrétní program je vytvořený podle původního testovacího předpisu a má sloužit k ulehčení práce manuálních testerů a lepšímu ověření kvality rozhraní ve formě otestování více možných konfigurací. V práci jsem popsal návrh a architekturu frameworku i programu, kde jsem zmínil očekávané vstupy, výstupy a ostatní charakteristiky, které jsem vybral pro řešení.

Úspěšně jsem implementoval požadovanou funkcionalitu a popsal nejčastěji frekventované řešené situace. Jednalo se převážně o výběr správných nástrojů a programovacího jazyka, kde jsem si zvolil již zmiňovaný Selenium Webdriver nad jazykem Python a framework pro jednotkové testování Pytest pro jeho široké spektrum možností a způsobu použití.

Výsledný framework funkcí jsem testoval přímo při implementaci do jednotlivých částí testovacího programu, jelikož spolu obě hlavní části (tj. framework a test) úzce souvisejí, a proto byl také jejich vývoj prováděn současně. Výsledný testovací program má poměrně přímočaré řešení s jednoduchou konfigurací, proto se poměrně dobře testoval v průběhu implementace po malých částech, popřípadě při ověření funkčnosti většího celku. Pro hlášení o testování a přehledu potenciálních oprav sloužil verzovací systém Bitbucket, přes který

probíhala i komunikace s externími testery o průběhu testováním a dalších možnostech výsledného řešení.

Řešení se podařilo úspěšně implementovat a bylo důkladně otestováno jak mnou, tak i externími testery.

Literatura

[1] Breux, G. *Client-side vs. Server-side vs. Pre-rendering for Web Apps* [online]. 2018 [cit. 2020-04-11]. Dostupné z: https://www.toptal.com/front-end/client-side-vs-server-side-pre-renderings.

[2] Burget, R. *HTML z pohledu vstupu a výstupu IS*. Božetěchova 1/2, 612 00 Brno-Královo Pole: FIT VUT v Brně, 2018.

[3] Google. *Zabezpečení webu protokolem HTTPS* [online]. 2020 [cit. 2020-04-01]. Dostupné z: https://support.google.com/webmasters/answer/6073543?hl=cs.

[4] HelloSelenium. *What is workflow of Selenium RC?* [online]. 2013 [cit. 2020-05-10]. Dostupné z: http://www.helloselenium.com/2013/10/what-is-workflow-of-selenium-rc.html.

[5] Hlava, T. *Funkční a nefunkční testy* [online]. 2011 [cit. 2020-03-21]. Dostupné z: http://testovanisoftwaru.cz/tag/funkcni-testy/.

[6] Hlava, T. *Progresní a regresní testy* [online]. 2011 [cit. 2020-03-21]. Dostupné z: http://testovanisoftwaru.cz/tag/regresni-testy/.

[7] Hlava, T. *Smoke testy* [online]. 2011 [cit. 2020-03-21]. Dostupné z: http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/smoke-testy/.

[8] Joey Y.. *What makes Katalon Studio more powerful than any other Selenium based Frameworks?* [online]. 2017 [cit. 2020-05-17]. Dostupné z: https://www.utest.com/articles/what-makes-katalon-studio-more-powerful-than-any-other-selenium-based-frameworks.

[9] Katalon LLC. *Introducing Katalium: Selenium and TestNG Made Easy* [online]. 2019 [cit. 2020-05-17]. Dostupné z: https://www.katalon.com/resources-center/blog/katalium-introduction/.

[10] Katalon LLC. *Katalon Automation Recorder Quickstart* [online]. 2019 [cit. 2020-05-17]. Dostupné z: https://www.katalon.com/resources-center/blog/katalon-automation-recorder/.

[11] Katalon LLC. *Katalon Docs* [online]. 2019 [cit. 2020-03-21]. Dostupné z: https://docs.katalon.com/katalon-studio/docs/index.html#products.

[12] Katalon LLC. *Katalon Studio* [online]. 2019 [cit. 2020-03-01]. Dostupné z: https://www.katalon.com/katalon-studio/.

[13] Katalon LLC. *Katalon TestOps* [online]. 2019 [cit. 2020-03-21]. Dostupné z: https://www.katalon.com/testops/.

[14] Katalon LLC. *A Makeover for Katalon Brand Identity* [online]. 2019 [cit. 2020-05-17]. Dostupné z: https://www.katalon.com/resources-center/blog/katalon-brand-identity-makeover/.

[15] Kreutzmanbitbar, W. *What is and why do we need selenium framework?* [online]. 2020 [cit. 2020-05-10]. Dostupné z: https://bitbar.com/blog/what-is-and-why-do-we-need-selenium-framework/.

[16] Kučera, F. *Protokol HTTP* [online]. 2011 [cit. 2020-03-31]. Dostupné z: https://www.zdrojak.cz/clanky/protokol-http/.

[17] Malvi, K. *Top 30 Python Web Frameworks to Use in 2020* [online]. 2019 [cit. 2020-03-23]. Dostupné z: https://www.mindinventory.com/blog/best-python-web-frameworks-2019/.

[18] Marketwired. *KMS Technology Releases Katalon Studio, Free Intelligent Test Automation Toolset* [online]. 2016 [cit. 2020-05-17]. Dostupné z: https://finance.yahoo.com/news/kms-technology-releases-katalon-studio-133727741.html.

[19] Ranorex GmbH. *Ranorex* [online]. 2020 [cit. 2020-03-22]. Dostupné z: https://www.ranorex.com/.

[20] SmartBear Software. *The State of Software Testing 2018 Industry Report* [online]. 2018 [cit. 2020-05-17]. Dostupné z: https://smartbear.com/resources/ebooks/state-of-testing-report-2018/.

[21] Software Freedom Conservancy. *Selenium 1 (Selenium RC)* [online]. 2020 [cit. 2020-05-17]. Dostupné z: https://www.selenium.dev/documentation/en/legacy_docs/selenium_rc/.

[22] Softwaretestinghelp. *A Complete Non-Functional Testing Guide For Beginners* [online]. 2020 [cit. 2020-05-17]. Dostupné z: https://www.softwaretestinghelp.com/what-is-non-functional-testing/.

[23] Softwaretestinghelp. *What Is Regression Testing? Definition, Tools, Method, And Example* [online]. 2020 [cit. 2020-05-17]. Dostupné z: https://www.softwaretestinghelp.com/regression-testing-tools-and-methods/.

[24] SUSE. *Embedded Application* [online]. 2020 [cit. 2020-03-22]. Dostupné z: https://susedefines.suse.com/definition/embedded-application/.

[25] Vaadin Ltd. *Vaadin* [online]. 2020 [cit. 2020-03-23]. Dostupné z: https://vaadin.com/.

[26] Zalavadia, S. *Complete Functional Testing Guide With It's Types And Example* [online]. 2020 [cit. 2020-05-17]. Dostupné z: https://www.softwaretestinghelp.com/guide-to-functional-testing/.

Příloha A

Obsah přiloženého paměťového média

```
/
├── bin/ ..... pomocné programy Petra Juráka pro práci s logy stanice
├── bma/ ..... výsledný program
│   ├── lib/ ..... framework funkcí nad webovým rozhraním stanice
│   │   ├── Pages/ ..... jednotlivé soubory funkcí pro každou webovou stránku
│   │   ├── Resources ..... ostatní funkce frameworku
│   │   └── README.md
│   └── test_web_predpis/ ..... automatický test
│       ├── config/ ..... konfigurační soubory
│       │   └── default.cnf
│       ├── reports/ ..... výsledky testů
│       ├── conftest.py ..... inicializace automatického testu
│       ├── key ..... vygenerovaný testovací klíč pro uživatelský účet
│       ├── key.pub ..... vygenerovaný testovací klíč pro uživatelský účet
│       ├── pytest.ini ..... přednastavené parametry spouštění
│       ├── README.md ..... návod pro spuštění automatu a jeho popis
│       ├── test_1.py ..... základní část testu
│       └── test_main.py ..... hlavní tělo testu
├── text_source/ ..... zdrojové soubory pro sestavení textu práce
├── README.md
├── requirements.txt ..... Python3 moduly potřebné pro chod programu
└── text.pdf ..... text práce
```