VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Automatické testování webových aplikací

Automated testing of web applications

2017 Bc. Jakub Janoška

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Janoška**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Automatické testování webových aplikací
Automated Testing of Web Applications

Jazyk vypracování: čeština

Zásady pro vypracování:

Teoretické zpracování oblasti testování webových aplikací. Analýza vybraných nástrojů pro automatizaci testů a implementace příkladů nad konkrétní webovou aplikací. Vyhodnocení použitých nástrojů.

1. Teoretické zpracování testování softwaru.
2. Analýza vybraných nástrojů.
3. Implementace automatických testů pro konkrétní webovou aplikaci.
4. Porovnání použitých nástrojů.
5. Vyhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Rastislav Kanócz**

Konzultant diplomové práce: Ing. David Ježek, Ph.D.

Datum zadání: 01.09.2015

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky 1D16 OSTRAVA

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. dubna 2017

profiq s.r.o.
Nad Okrouhlíkem 17/2365
182 00 Praha 8 - Libeň
IČ: 24660761 DIČ: CZ24660761

Rád bych poděkoval Ing. Rastislavovi Kanóczovi za jeho odbornou pomoc, cenné rady a velmi vstřícný přístup během tvorby diplomové práce. Také bych chtěl poděkovat Ing. Davidovi Ježkovi, Ph.D. za jeho cenné rady.

Abstrakt

Tato diplomová práce se zabývá automatickým testováním webových aplikací se zaměřením na uživatelské rozhraní pro desktop zařízení. V teoretické části si vysvětlíme, proč je testování potřebné, z jakých aktivit se skládá a jaké existují typy testů. Dále si vysvětlíme pojem webových aplikací či webových stránek a popíšeme si použité technologie. Podrobně si rozebereme problematiku selektorů a ukážeme si několik ukázek zdrojových kódů s automatickými testy. Cílem práce je porovnání vybraných nástrojů pro tvorbu automatických UI testů. Mezi porovnávané nástroje patří WebDriverIO, Sencha Test, TestCafe a Sahi. Nástroje budou porovnány na základě důkladného průzkumu nabízených funkcí a získaných zkušeností v průběhu implementace automatických testů nad aplikací Roundcube a ExtJS Row Editing. Pro porovnání nástrojů je definováno 7 kategorií, kde každá kategorie bude zvlášť vyhodnocena. Závěr práce shrnuje dosažené výsledky a navrhuje možné rozšíření diplomové práce.

Klíčová slova: API, automatické testování, Jasmine, JavaScript, Sahi, Sencha Test, TestCafe, testovací případ, WebDriverIO

Abstract

This master thesis is about automation of web applications with focusing on UI for desktop. In the theoretical part, we will explain why is testing necessary, which activities are usually part of the testing process, and what types of tests exist. Further we will explain concept of web application or web page and we will describe technologies used in the thesis. We will discuss in detail about selectors topic and we will publish a couple of examples with source codes of automated tests. The goal of this thesis is comparison of selected tools and test frameworks for developing automated UI tests. Compared tools are WebDriverIO, Sencha Test, TestCafe and Sahi. The tools will be compared based on a thorough investigation of the offered features and gained experience during the implementation of automated tests for Roundcube and ExtJS Row Editing applications. We will define 7 categories for tools comparison, where every category will be evaluated itself. The conlusion of the thesis summarizes achieved results and suggests possible extensions of diploma thesis.

Key Words: API, automated testing, Jasmine, JavaScript, Sahi, Sencha Test, TestCafe, test case, WebDriverIO

Obsah

Seznam použitých zkratek a symbolů

AJAX	–	Asynchronous JavaScript And XML
API	–	Application Program Interface
ASP	–	Active Server Pages
BDD	–	Behavior Driven Development
CI	–	Continuous Integration
CSS	–	Cascading Style Sheets
DOM	–	Document Object Model
GUI	–	Graphical User Interface
HTML	–	HyperText Markup Language
IMAP	–	Internet Message Access Protocol
ISO	–	The International Organization for Standardization
ISTQB	–	International Software Testing Qualifications Board
J2EE	–	Java 2 Enterprise Edition
JSON	–	JavaScript Object Notation
JVM	–	Java Virtual Machine
MIT	–	Massachusetts Institute of Technology
ST	–	Sencha Test
STC	–	Sencha Test Command
TC	–	Test Case
UI	–	User Interface
VŠB	–	Vysoká Škola Báňská
XML	–	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Pro vytvoření úspěšné, široce rozšířené aplikace nestačí mít pouze dobrý nápad. Velmi zásadním kritériem úspěchu je kvalita produktu. Kvalitu můžeme chápat jako míru spokojenosti zákazníka a shodu vytvořeného softwaru se specifikací produktu. Nyní vyvstává celá řada otázek jako například, jak správně určit míru spokojenosti zákazníka, kdy je vhodný čas a je vůbec potřebné toto zjišťovat? Na tyto otázky se pokusíme odpovědět v průběhu práce.

Jen si uveďme jednoduchý příklad k zamyšlení. V dnešní době téměř každý člověk se dostane do kontaktu s informačními technologiemi. Ať je to při nakupování, hraní her nebo při práci se sofistikovaným softwarem, který má podporovat obchodní proces nebo jakékoliv jiné činnosti. Každý člověk bude velmi frustrován jestli se setká s uživatelsky nepřívětivým ovládáním nebo dokonce neočekávaným či chybným chováním. Takové situace mohou vést k poškození reputace, finančním ztrátám, v krajním případě dokonce ke ztrátám na životech. Z toho příkladu jasně vyplývá, že chyby v software mohou mít obrovský dopad a fatální následky.

Jak už název práce napovídá, budeme se věnovat automatickému testování webových aplikací. Práce obsahuje celkem 9 kapitol, včetně úvodu i závěru. Druhá kapitola se věnuje teoretickému zpracování testování, kde si například popíšeme co je to chyba, rozebereme jednotlivé testovací procesy, typy testů apod. Ve třetí a čtvrté kapitole si stručně popíšeme technologie, se kterými se setkáme v průběhu práce. Zvláštní pozornost budeme věnovat webovým aplikacím. Vybraným nástrojům pro tvorbu automatických testů věnujeme 5. kapitolu, kde si nástroje představíme a popíšeme.

6. kapitolou už se dostáváme k praktickému řešení diplomové práce. Představíme zde testovanou aplikaci a také si definujeme oblasti, pro které budeme implementovat automatické testy. Následující kapitolou plynule přecházíme k samotné implementaci. V této kapitole také můžete nalézt ukázky zdrojových kódů. V předposlední kapitole se pokusíme porovnat použité nástroje pro tvorbu automatických testů. Budeme se snažit vycházet nejen ze zkušeností získaných v průběhu realizace práce, ale také se pokusíme zohlednit faktory či problémy, se kterými se softwarový inženýr denně setkává na reálných projektech.

Aktuální trendy testování webových aplikací směřují k testování nativních i hybridních aplikací pro mobilní zařízení. Toto téma je nad rámec diplomové práce, avšak některé z představených nástrojů podporují takovéto testování a bylo by velmi zajímavé navázat na výzkumy z této práce a rozšířit je o testování nativních a hybridních aplikací pro mobilní zařízení.

2 Testování

Pod pojmem testování si řada lidí představí pouze manuální provádění testů, ale ve skutečnosti se jedná o komplexní proces skládající se ze statických i dynamických aktivit, které se zabývají plánováním, přípravou a ohodnocením softwarového produktu. Tyto aktivity si blíže rozebereme v kapitole 2.2 - Testovací procesy.

Nyní si pojďme uvést hlavní cíle testování:

- Nalezení defektů v programu.
- Zjištění míry kvality produktu na základě vykonaných testů.
- Poskytnutí informací pro management o kvalitě produktu.

Základem testování je tedy nalézt vady, ale pouhým testováním nezvýšíme kvalitu produktu. Testování pouze identifikuje problémy, ale odstranění zůstává na vývojářích, a proto je velmi důležitá úzká spolupráce a dobrá komunikace mezi vývojáři a testery. Po odstranění defektu je potřeba vykonat opětovné testování a ověřit, zda chyba byla opravdu odstraněna a také, že se žádná nová neobjevila. Nyní již hovoříme o regresním a opakovaném testování. Blíže se tomuto tématu budeme věnovat v kapitole 2.4.4 - Testování změn.

2.1 Chyba, vada, selhání

Při testování se často setkáváme s pojmy jako chyba, vada či selhání. Pro jednoznačnost si definujme tyto pojmy podle ISTQB [1].

- Chyba je lidský omyl, který produkuje nesprávné výsledky.
- Vada nebo také defekt, je chybná část kódu či systému, která může způsobit selhání požadované funkcionality.
- Selhání je odchylka systému od očekávaného výsledku.

Jinými slovy, lidská chyba může vytvořit vadu a pokud narazíme na vadu může nastat selhání. Vidíme jasnou spojitost mezi jednotlivými pojmy.

Vada je velmi úzce spjata se specifikací produktu. Rob Patton definoval vadu následujícím způsobem [3]:

Definice 1 *O softwarovou vadu se jedná, je-li splněna jedna nebo více z následujících podmínek:*

1. *Software nedělá něco, co by podle specifikace dělat měl.*
2. *Software dělá něco, co by podle specifikace produktu dělat neměl.*
3. *Software dělá něco, o čem se specifikace nezmiňuje.*

4. *Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.*

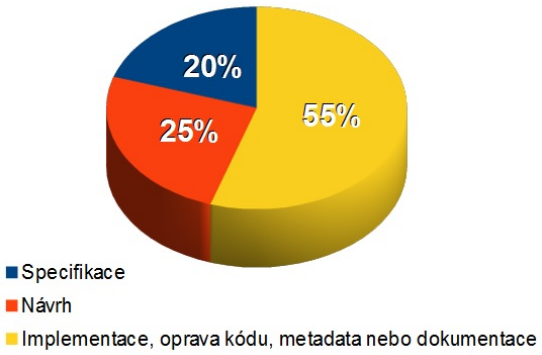
5. *Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý nebo - podle názoru testera softwaru - jej koncový uživatel nebude považovat za správný.*

Z definice je jasně patrný vztah mezi specifikací a vadou. Software testujeme vůči specifikaci a jakékoliv odlišnosti považujeme za vadu. Je důležité, aby veškerá funkcionalita popsaná ve specifikaci byla implementována a dokonce není přípustné aby software obsahoval funkcionalitu, která není popsána ve specifikaci.

Definice ovšem také zohledňuje uživatelské zkušenosti získané testerem. V tomto případě může software zcela odpovídat specifikaci, ale pokud tester vyhodnotí chování za nesprávné či jinak závadné, tak stále hovoříme o vadě. V tomto případě se vada vyskytla již ve specifikaci nebo návrhu aplikace.

Tato definice vychází z předpokladu detailního a podrobného zpracování specifikace požadavků, ale v praxi se často setkáváme pouze se základní specifikací. Zejména u agilních způsobů vývoje, jako je například metodika Scrum. Zde se klade důraz na jednoduchost s nejnutnější specifikací a funkčnost je prezentována formou dema na konci každé iterace. V tomto případě se zvyšuje přidaná hodnota testovacího týmu, který se stává nedílnou součástí specifikace požadavků. Testovací tým čelí nové výzvě v hledání vad a mnohdy je složité určit, zda se jedná o vadu či předpokládané chování. V tomto případě je nutná velmi úzká spolupráce testovacího týmu s týmem vývojářů, která je založena na neustálém vyjasňování jakýchkoliv nejasností.

Vady se mohou vyskytovat v průběhu celého životního cyklu produktu. Následující obrázek 1 znázorňuje procentuální rozložení vad napříč jednotlivými fázemi projektu. Můžeme vidět, že 20% vad se nalézá již ve specifikaci produktu, dalších 25% vad se vyskytuje v návrhu a zbylých 55% je obsaženo v implementační části, respektive v implementaci, opravách, metadatech či dokumentaci. Konkrétní data byla čerpána ze studie pana Jonese [4]:



Obrázek 1: Rozložení vad v jednotlivých fázích projektu

Pokud se zamyslíme nad závislostmi mezi vadami, uvidíme, že díky vadě ve specifikaci vzniká nesprávný návrh a tudíž nemůže být správný ani výsledný kód. Jedna taková vada může ovlivnit

i následující fáze projektu, a proto by bylo ideální nacházet vady ve stejné fázi, kde vznikají. Pochopitelně v reálném životě nejsme schopní zachytit všechny defekty vzniklé v aktuální fázi a objevení vady ve fázi pozdější se negativně projeví na úsilí a ceně potřebné k odstranění defektu. Poměr mezi cenou za odstranění defektu ve specifikační fázi a po vydání aplikace se značně liší v závislosti na složitosti produktu. Pro jednoduché aplikace můžeme hovořit o poměru 1:5, u komplexních aplikací se uvádí poměr dokonce až 1:100.

2.2 Testovací procesy

Jak už jsme si řekli na začátku této kapitoly, testování je komplexní proces. ISTQB definuje celkem 5 procesů, které na sebe navazují, v některých případech se mohou i překrývat. Podrobnější informace o testovacích procesech naleznete [1]

2.2.1 Plánování a řízení testů

Každé testování začíná plánováním, kde si musíme důkladně rozmyslet, co je naším cílem a jak toho chceme dosáhnout. Od toho se odvíjí zvolená strategie a definují se cíle. Výstupem plánovací fáze je test plán, který by měl podle IEEE 829 [11] obsahovat následující informace:

1. Identifikátor test plánu
2. Úvod
3. Testované objekty
4. Seznam funkcí, které budou testovány
5. Seznam funkcí, které nebudou testovány
6. Testovací přístupy
7. Kritéria úspěšného / neúspěšného dokončení testu
8. Kritéria pro přerušení a obnovení testu
9. Definování testovacích výstupů, dokumentů
10. Definování testovacích úkolů
11. Požadavky na testovací prostředí
12. Přiřazení zodpovědností
13. Požadavky na testovací tým a potřebná školení
14. Harmonogram

15. Identifikování rizik a mimořádných událostí

16. Schválení

S plánováním úzce souvisí také řízení. V průběhu procesu kontrolujeme, zda dosahujeme definovaných cílů v určených termínech a s danou kvalitou. V případě zjištění nedosažitelnosti cíle v termínu musíme provést přeplánování a tyto změny eskalovat výše, nejčastěji zákazníkovi a dalším zainteresovaným stranám.

2.2.2 Analýza a návrh testů

V plánovací fázi jsme si zvolili cíle a nyní v analýze a návrhu si popíšeme, jak daných cílů dosáhnout.

Základem pro analýzu a návrh je tzv. test báze, což jsou veškeré dokumenty, které definují požadavky na systém a jeho komponenty. Do test báze můžeme zařadit nejen specifikaci požadavků, ale také uživatelské příběhy či celou řadu diagramů, jako například diagram užití. Jak už jsme si řekli dříve, vada se může nacházet již v test bázi, a právě tyto vady by měly být nalezeny během analýzy.

V návrhu se snažíme specifikovat testovací případy. Testovacím případem rozumíme určení testované oblasti, popis testovacích kroků s požadovanými vstupy a očekávanými výstupy. Podle IEEE 829 [11] by měl testovací případ obsahovat následující informace:

1. Identifikátor testovacího případu

2. Předmět testování, popis

3. Specifikaci vstupů

4. Specifikaci výstupů

5. Informace o testovacím prostředí

6. Postup

7. Definování závislostí mezi jinými TCs

V tabulce 1 můžeme vidět konkrétní příklad testovacího případu pro přihlášení s neplatným heslem.

S takto podrobně rozepsanými testovacími případy se můžeme setkat ve velkých korporacích z bankovního či telekomunikačního sektoru, které většinou využívají vodopádového modelu pro vývoj systému. Z osobní zkušenosti s prací v agilním prostředí můžu říci, že se upouští od detailní a rozsáhlé dokumentace, jak je popsána normou IEEE 829 [11] z důvodu náročné údržby.

U menších společností s agilním přístupem se volí přístup efektivní a stručné dokumentace, což v praxi znamená, že během analýzy testovací tým specifikuje formou průzkumného testování

ID:	001
Název:	Přihlášení s neplatným heslem
Popis:	Pokus o přihlášení do emailovému klienta s neplatným heslem
Předpoklady (závislosti):	Uživatel se nachází na přihlašovací stránce s nevyplněným emailem a heslem
Vstup:	Platná a registrovaná emailová adresa s neplatným heslem
Postup:	1. Uživatel vloží emailovou adresu 2. Uživatel vloží heslo 3. Uživatel stiskne tlačítko přihlásit
Výstup:	Přístup je odepřen a varovná zpráva o neplatném heslu je zobrazena
Prostředí:	Otestovat na následujících webových prohlížečích: Chrome 57, Firefox 52, IE 11, Edge 38

Tabulka 1: Testovací případ - 001 - Přihlášení s neplatným heslem

jednoduché testovací případy bez specifikace jednotlivých kroků. Klade se důraz na rychlé automatizování, kde testovací skripty slouží jako specifikace a dokumentace testovacího případu. Z tohoto důvodu se klade důraz na dobrou čitelnost kódu, který musí splňovat definované konvence kódování.

Při vytváření testovacích případů by jsme měli také myslet na propojení testovacích případů s odpovídajícími dokumenty z test báze. Pomocí tohoto propojení jsme schopni určit pokrytí testovaných funkcí a také budeme mít ulehčenu údržbu testů v případě změn.

Posledním jmenovaným úkolem této fáze je příprava testovacího prostředí a identifikace potřebných dat pro testování. Také si zvolíme potřebné nástroje pro podporu testování.

2.2.3 Implementace a vykonávání testů

V implementační fázi dále pracujeme s testovacími případy. Ty se snažíme uspořádat do tzv. testovacích procedur, což můžeme chápat jako posloupnost po sobě jdoucích testovacích případů, které se mohou vzájemně ovlivňovat. Při vytváření procedur musíme brát v potaz, jakým způsobem budou testy vykonávány. Pokud hovoříme o manuální exekuci, volíme maximální možnou závislost mezi testovacími případy k dosažení maximální efektivnosti vykonávání testů. V případě automatického provádění testů je mnohem důležitějším kritériem stabilita testů a tudíž by měl být každý testovací případ nezávislý na ostatních.

Z testovacích procedur dále vytváříme testovací sady, kde přiřadíme jednotlivým testovacím případům priority a opět se snažíme zachovávat logickou posloupnost. Testovací sadu můžeme brát jako návod, jak má tester postupovat při test exekuci. V životním cyklu projektu může být více testovacích sad a řada z nich může být vykonávána opakovaně či pravidelně. V případě pravidelného provádění testovací sady, je vhodné zvážit nasazení automatizace. A právě v implementační fázi se zabýváme automatizací, jestli je součástí daného projektu. Blíže se této problematice budeme věnovat v kapitole 7 - Automatické testování.

Dále v této fázi dokončíme ostatní započaté aktivity z předchozí fáze, jako například vytvoření testovacích dat, ověření testovacího prostředí. Ideálně ještě před začátkem test exekuce.

Po dokončení všech aktivit z implementační fáze můžeme přistoupit k samotnému vykonávání testů. Testy provádíme podle připravených testovacích případů, respektive podle testovacích sad a můžeme je vykonávat automaticky nebo manuálně.

Průběh testování je potřeba zaevidovat a k tomu slouží testovací protokol. Každý protokol musí obsahovat co se testovalo a jakých se dosáhlo výsledků. Důležité jsou nejen výsledky testů, ale také popis testovacího prostředí a použité verze testovaného systému.

Podstatou provádění testů je porovnávání očekávaných a aktuálních výsledků a jakékoliv nesrovnalosti jsou označovány jako incident a musí být dále prozkoumány. Odchylka od očekávaného chování nemusí nutně znamenat vadu, ale mohlo dojít pouze k chybnému vykonání testu. Mohla být například použita chybná vstupní data a podobně. V případě zjištění vady, je potřeba ji zaevidovat. Podle IEEE 829 [11] by měl záznam o incidentu obsahovat následující:

1. Identifikátor incidentu
2. Souhrn
3. Popis incidentu (vstupní data, očekávané a aktuální výsledky, anomálie, datum a čas, kroky pro zopakování incidentu, prostředí, jméno testera)
4. Dopad

2.2.4 Vyhodnocení kritérií ukončení a reportů

V této fázi by jsme měli mít dokončené naplánované testování a mít vytvořené testovací protokoly. Tyto protokoly porovnáváme s výstupními kritérii, jenž jsme si definovali v plánovací

fázi. Také by jsme měli mít vyřešeny nebo akceptovány veškeré objevené incidenty. Následující aktivity záleží na typu projektu a dosažených výsledcích. Vždy ale musíme posoudit vzhledem k výstupním kritériím, jestli bylo testování dostatečné a zda bylo dosaženo požadované kvality.

Tyto poznatky je potřeba formálně zpracovat v dokumentu zvaném hodnocení kvality a prezentovat všem zúčastněným stranám.

2.2.5 Ukončení testovacích aktivit

V této fázi je projekt dokončen a vydán, ale tímto testování nekončí a následuje ještě jedna poslední fáze v níž provedeme mimo jiné tzv. retrospektivu.

Retrospektivu si můžeme představit, jako vyhodnocení odvedené práce a zamyšlení se, jak dané procesy vylepšit. Získané znalosti nám umožní lépe plánovat, navrhovat efektivnější postupy a celkově přispějí ke zdokonalení testovacích procesů.

Důležité je také si archivovat veškeré použité dokumenty, testovací data i testovací protokoly. Takové dokumenty hromadně označujeme jako testware.

Dále je velmi výhodné provést údržbu a optimalizaci testovacího prostředí, které budeme moci využit při dalším testování.

2.3 Jakostní charakteristiky

Už víme co znamená kvalita produktu, ale jak ji můžeme vyjádřit? Mezinárodní organizace ISO se touto problematikou zabývala v normě [ISO/IEC 9126-1:2001], kde definovala 6 jakostních charakteristik. Tato norma bylo roku 2011 nahrazena normou [ISO/IEC 25010:2011], která obsahovala už celkem 8 charakteristik. Kompletní seznam charakteristik i jejich pod charakteristik můžeme vidět v tabulce 2, která je převzata z normy [ISO/IEC 25010:2011] [10]

Funkční způsobilost	**Spolehlivost**
Funkční úplnost	Vyspělost
Funkční správnost	Dostupnost
Funkční vhodnost	Odolnost proti chybám
Efektivita provedení	Obnovitelnost
Časový průběh	**Bezpečnost**
Využívání zdrojů	Důvěrnost
Kapacita	Integrita
Kompatibilita	Nepopiratelnost
Koexistence	Odpovědnost
Interoperabilita	Pravost
Použitelnost	**Udržovatelnost**
Rozpoznatelnost	Modularita
Naučitelnost	Znovupoužitelnost
Provozuschopnost	Snadná analyzovatelnost
Ochrana proti uživatelským chybám	Modifikovatelnost
Přehledné uživatelské rozhraní	Testovatelnost
Přístupnost	

Tabulka 2: Model kvality produktu

2.4 Typy testů

2.4.1 Funkční testování

Jedná se o to nejzákladnější testování, kde zjišťujeme, jestli daný program či systém dělá to co je od něj požadováno. Jinými slovy zjišťujeme jestli vše funguje a je v souladu se specifikací.

Existují 2 možné přístupy. První na základě specifikace požadavků a druhý z pohledu byznys modelů. Oba přístupy můžeme zařadit mezi black-box techniky, kdy je potřeba znát pouze to, jak se má systém chovat. Není potřebná znalost vnitřních struktur, kódu apod.

2.4.2 Nefunkční testování - testování kvalitativních charakteristik

Přistoupíme-li k tomuto testování, předpokládá se, že systém je funkční a nyní se zkoumají kvalitativní charakteristiky. Jinými slovy, jak kvalitně, rychle, či spolehlivě jsou dané funkce

provedeny. Kompletní seznam jakostních charakteristik jsme si uvedli již dříve v kapitole 2.3 jakostní charakteristiky.

Mezi nefunkční testování můžeme například zařadit výkonnostní testování, zátěžové testování, stresové testování, testování spolehlivosti, testování použitelnosti či testování přenositelnosti.

Obecně řadíme nefunkční testování mezi black-box techniky.

2.4.3 Testování struktury/architektury

Strukturální testování můžeme zařadit mezi white-box techniky. Říkáme white-box, protože je nutná znalost vnitřních struktur či kódu. Typickým příkladem strukturálních testů je pokrytí kódu, kde zkoumáme, které části kódu byly spuštěny během vykonávání testů, a které nebyly. Výsledek je nejčastěji vyjádřen v procentech.

2.4.4 Testování změn

Změny v software mohou přicházet z mnoha důvodů. Ať už je změna z důvodu přidání nové funkcionality, upravení stávající a nebo z důvodu opravení defektu, vždy musíme ověřit, že změna byla úspěšná. Také musíme ověřit, že změnou nebyla zavedena nová vada do programu.

Opakované testování

K opakovanému testování dochází po opravě defektu. Při testování vycházíme ze záznamu o incidentu, kde najdeme vstupní data, informace o testovacím prostředí a přesné kroky, které vedly k selhání a předpokládáme, že nyní bude test úspěšný s novou verzí programu. Pokud ano, můžeme incident uzavřít jako vyřešený. Opakované testování není dostatečné, protože opravou defektu můžeme zanést či objevit nový defekt. K omezení těchto vedlejších efektů slouží regresní testování.

Regresní testování

Při regresním testování vykonáváme testy, které už byly dříve vykonány a ve většině případů byly úspěšné. Cílem tohoto testování je ověření, že díky implementovaným změnám nedošlo k poškození stávající funkcionality.

3 Webová aplikace, webová stránka

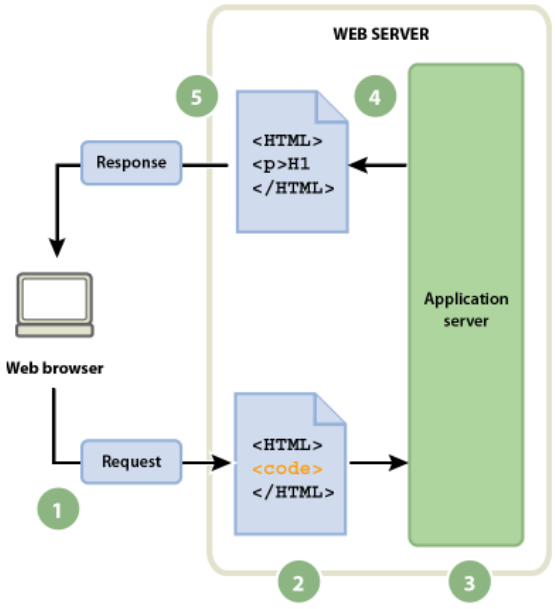
Jak už název napovídá, tématem této práce je testování webových aplikací. Oblast testování jsme si probrali v předchozí kapitole a nyní by jsme si mohli něco říci k webovým aplikacím či stránkám.

Mezi odbornou veřejností panuje vášnivá diskuze o rozdílech mezi webovou aplikací a webovou stránkou. Neexistuje jednoznačná definice a jednotlivé rozdíly mohou být chápány velmi subjektivně.

Webové stránky jsou převážně informativního charakteru, bez možné interakce uživatelů a vyznačují se převážně statickým obsahem. Typickým příkladem webové stránky může být například zpravodajský server cnn (http://cnn.com/). Určitě by někdo mohl namítat, že se zprávy mění každý den, a proto nejde o statický obsah. Ale z jiného úhlu pohledu se jedná jen o soubor statických HTML stránek.

Naopak webové aplikace umožňují interakci s uživatelem a ten dokáže znatelně ovlivnit zobrazovaný obsah, či vykonané funkce. Typickým příkladem webové aplikace může být emailový klient, který poskytuje uživateli možnost číst, psát emailové zprávy a mnoho dalšího.

Webová aplikace či stránka je postavena na modelu klient-server. Aplikace je spuštěna na webovém serveru a prostřednictvím internetu, nebo také intranetu je dostupná klientům. Klient si zobrazí webovou aplikaci pomocí webového prohlížeče, který dnes najdeme na každém počítači či mobilním zařízení.



Obrázek 2: Zpracování dynamické stránky

Oblast webových technologií prochází bouřlivým vývojem. Počátky spočívaly pouze v jednoduchých statických stránkách, nyní webové aplikace nabízejí dynamicky generovaný obsah s využitím databází a dokonce se přibližují svými funkcemi k aplikacím nainstalovaných přímo na počítači. Dnešní moderní aplikace jsou běžně optimalizovány pro všechny moderní webové prohlížeče, které mohou být spuštěny na počítači, mobilním telefonu či tabletu.

Na obrázku 2 můžeme vidět, jak jsou dynamické stránky zpracovány pomocí aplikačního serveru. Nejprve webový klient pošle požadavek na web server. Požadavek můžeme chápat jako zadání url. Webový server vyhledá požadovanou stránku a pokud stránka obsahuje kód pro tvorbu dynamických stránek, předá zpracování aplikačnímu serveru. Aplikační server dokončí stránku a vrátí ji zpět webovému serveru a ten ji pošle zpátky webovému prohlížeči.

Aplikační servery využívající javu označujeme jako J2EE a mezi nejznámější zástupce patří například Apache Tomcat či WebSphere. Dále také existují ASP či PHP aplikační servery. Také můžeme hovořit o dynamicky generované aplikaci, která nevyužívá aplikačního serveru, ale dynamický obsah je generován na straně klienta pomocí javascript. Obrázek 2 byl převzat z [12] a zde také najdete podrobnější informace.

Dynamický obsah ke stránkám je přidáván pomocí skriptovacích jazyků jako je PHP, ASP nebo javascript. Tyto jazyky můžeme dále rozdělit na jazyky se skriptováním na straně serveru, mezi ně patří například ASP a PHP. Naopak u javascript probíhá skriptování na straně klienta, více o tomto jazyku v kapitole 4.1. Hojně je také využíváno technologie AJAX, což umožňuje měnit obsah stránky, bez nutnosti kompletního načtení stránky.

Webové aplikace jsou velmi oblíbené z mnoha důvodů. Obrovskou výhodou je, že webové aplikace stavějí na standardních funkcích prohlížeče, a proto by aplikace měla fungovat správně na různých operačních systémech (Windows, OS X, Linux, Android, iOS) a různých webových prohlížečích(Firefox, Chrome, IE, Edge, Safari). Bohužel v praxi se setkávám s drobnými rozdíly mezi webovými prohlížeči, které přináší určité komplikace s nimiž se musí vývojáři vypořádat. Tyto rozdíly také způsobují problémy při tvorbě automatických testů, ale o této problematice více v kapitole 8. Další výhodou je jednoduchá správa aplikací. Aplikaci jednoduše aktualizujeme či upravíme na serveru a tyto změny se automaticky projeví u klientů.

Nevýhodou webových aplikací je závislost na stabilním internetovém připojení a možná bezpečnostní rizika z úniku dat uložených na serveru.

Mezi nejznámější a nejrozšířenější webové aplikace patří například facebook či google aplikace (emailový klient, překladač, mapy a další).

4 Použité technologie

V této kapitole uvedeme programovací jazyky a frameworky, které využijeme při implementaci automatických testů. Záměrem je pouze stručné představení použité technologie a ne detailní rozebrání daného jazyka nebo frameworku.

4.1 JavaScript

První verze javascript byla vydána roku 1995 a autorem je Brendan Eich. Jedná se o multiplatformní, objektově orientovaný skriptovací jazyk, který byl původně navržen pro jednoduchou interakci s webovou stránkou. Postupem času javascript mnohonásobně předčil původní záměr a v dnešní době je možné pomocí tohoto jazyku vytvářet komplexní webové aplikace a dokonce také desktop aplikace.

Existuje celá řada frameworků, které jsou postaveny na javascript. Z těch nejznámějších můžeme jmenovat Angular JS, React JS nebo také Ext JS, kterému se budeme dále věnovat.

4.2 Ext JS

Ext JS je javascript framework, který se zaměřuje na tvorbu HTML5 aplikací s vysokou datovou náročností a multiplatformní podporou počítačů a mobilních zařízení. Framework obsahuje celou řadu komponent pro usnadnění a zefektivnění tvorby webových stránek, respektive webových i mobilních aplikací.

Z těch nejzajímavějších bych jmenoval grid, kalendář, graf, d3 adaptér. Kompletní seznam komponent a také podrobnější informace naleznete v dokumentaci Ext JS [8]

4.3 Jasmine framework

Jasmin je BDD framework pro testování využívající javascript. Je to open source projekt a podrobné informace najdete zde [5].

Základem tohoto frameworku jsou funkce *describe* a *it*. *Describe* také označován jako testovací soubor a *it*, kterému odporně říkáme spec. Obě funkce mají 2 parametry, kde první je textový řetězec definující název a druhý parametr je funkce obsahující kód testovacího souboru nebo samotného testu. Každá spec obsahuje 1 či více assertions podle níž budou vyhodnoceny výsledky testů. Očekávání můžeme chápat jako porovnání očekávané hodnoty s aktuální.

Hlavní výhodou použití Jasmine frameworku je synchronní běh uvnitř Jasmine funkce a organizace testů do dílčích specs. Možnou alternativou Jasmine frameworku je framework Mocha, který disponuje velmi podobnou funkcionalitou. Jasmine je upřednostněna, protože Sencha Test využívá tohoto frameworku. Více informací o Mocha naleznete zde [6].

4.4 Node.js

Node.js je JavaScript runtime postavený na Chrome V8. Chrome V8 slouží jako interpret pro JavaScript a více informací můžeme nalézt zde [27].

Pomocí Node.js můžeme vytvářet vysoce škálovatelné aplikace využívající asynchronní I/O operace pro dosažení maximálního výkonu. Více informací o Node.js naleznete zde [26].

5 Nástroje pro tvorbu automatických testů

V této kapitole si nejprve popíšeme nástroje pro vývojáře, které ulehčují tvorbu a ladění automatických testů. Budeme pokračovat nástroji pro tvorbu automatických testů, které budou v práci použity a v závěru také porovnány.

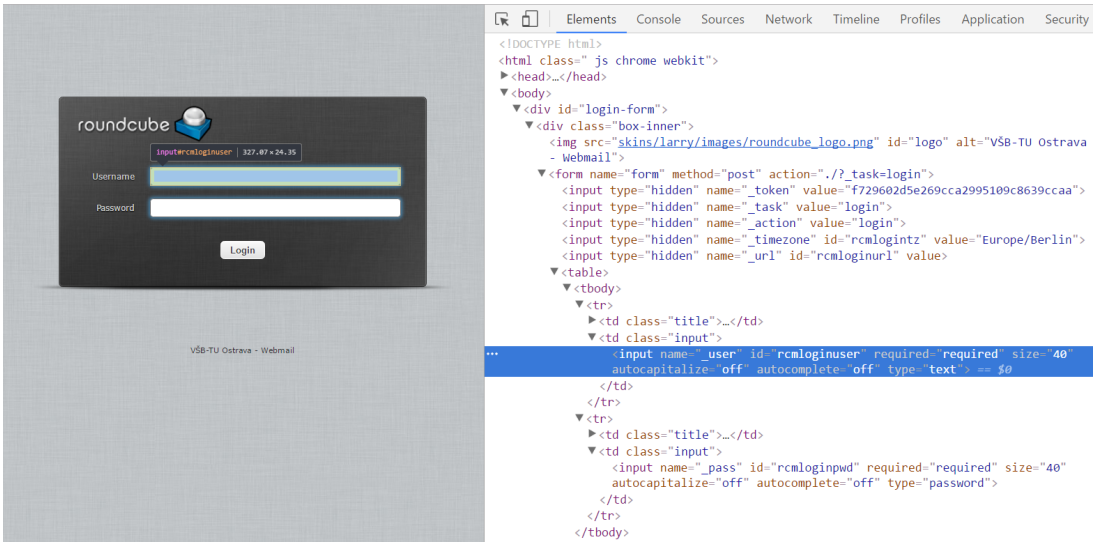
Speciální pozornost bude věnována Seleniu, které položilo základy automatického testování. Dodnes Selenium WebDriver patří k nejpopulárnějším testovacím nástrojům a byla nad ním vyvinuta celá řada knihoven, jako například WebDriverIO či Webdriver-sync.

Dále si uvedeme komerční nástroje, které jsem si vybral na základě uživatelských recenzí, žebříčků či diskusí. Zohlednil jsem také osobní preference a zkušenosti získané ze své několikaleté pracovní zkušenosti v oboru automatického testování. Více o vybraných nástrojích v následujících kapitolách.

5.1 Nástroje pro vývojáře

I když se nejedná o nástroj pro tvorbu automatických testů, považuji za velmi důležité jej zmínit, protože je to základní nástroj pro každého vývojáře či softwarového inženýra. Na obrázku 3 můžeme vidět HTML strukturu webové aplikace zobrazenou pomocí Chrome nástrojů pro vývojáře. Ale tento nástroj je poskytován i ostatními prohlížeči buď nativně nebo pomoci pluginů do prohlížeče. Na základě znalosti HTML struktury můžeme vytvářet selektory, pomocí kterých můžeme manipulovat s webovou stránkou.

Nástroje pro vývojáře také obsahují javascript konzoli, pomocí ní můžeme například ověřit správnost vytvořených selektorů. Také můžeme pomocí nástrojů pro vývojáře ladit javascript kód a mnohem více. Více informací ohledně Chrome nástrojů pro vývojáře naleznete zde [14]. Také přikládám možnou variantu pro webový prohlížeč Firefox [15].



Obrázek 3: Chrome - nástroje pro vývojáře

5.2 Selenium

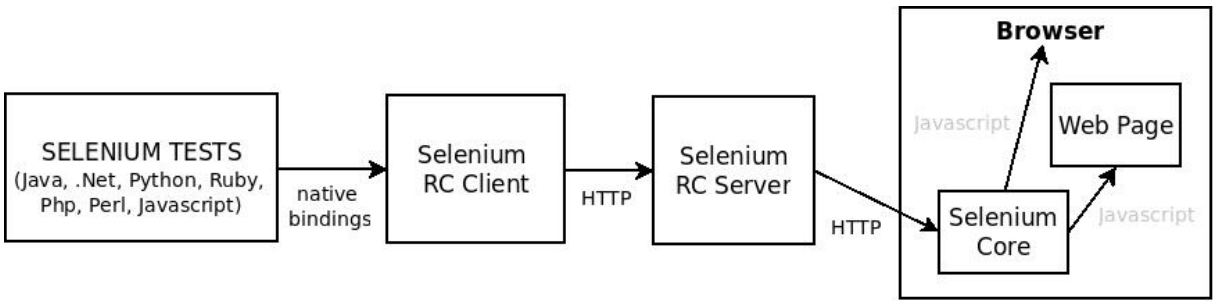
Základy Selenia byly vytvořeny v roce 2004 panem Jasonem Hugginsem, který vytvořil javasriptovou knihovnu umožňující automatické spouštění testů v různých prohlížečích a řízení interakce se stránkou. Postupem času se knihovna stala takzvaným Selenium Core, což je základem pro nástroje ze sady Selenia, konkrétně Selenium RC a Selenium IDE.

V roce 2006 se začal vyvíjet nástroj jménem Selenium WebDriver za účelem odstranění limitací z předcházejících nástrojů. Největším přínosem je nativní podpora pro interakce s prohlížečem a operačním systémem.

Nyní Selenium patří k jednomu z nejrozšířenějších open-source řešení pro automatické testování webových aplikací s velmi širokou komunitou a komerční podporou. Dokonce celá řada placených softwarů pro automatické testování využívá Selenia, zejména WebDriver technologie.

5.2.1 Selenium RC

Selenium Remote Control, také známý jako Selenium 1.0 je prvním významným projektem od Selenia. V dnešní době je pouze v údržbovém módě a je nahrazen Seleniem 2.0.



Obrázek 4: Architektura Selenia RC

Na obrázku 4 můžeme vidět architekturu Selenia RC. Skládá se ze 2 komponent. Selenium RC server se stará o interakci s prohlížečem a je také prostředníkem mezi Selenium příkazy a prohlížečem. Selenium RC klient je druhou komponentou, která poskytuje interface mezi programovacím jazykem a Selenium RC, což si můžeme představit jako Selenium příkazy. Můžeme si všimnout, že je podporována celá řada programovacích jazyků, jako například Python, Java a další.

Selenium RC používá javascript injection pro automatické testováni, které je společné pro všechny podporované prohlížeče. Z toho plyne omezená funkcionalita, například není možná podpora pop-up oken a stahování souborů.

5.2.2 Selenium WebDriver

Selenium WebDriver, také označován jako Selenium 2.0, je více objektově orientovanou a kompaktnější sadou API. Obrovským přínosem je nativní podpora nejrozšířenějších prohlížečů, což

umožňuje maximálně přiblížit automatické testování uživatelským interakcím, zvyšuje rychlost a stabilitu testování. Je zde odstraněna řada limitací předchozího Selenia 1.0, jako například stahování a nahrávání souborů, pop-up okna a dialogy. Na obrázku 5 můžete vidět architekturu Selenia WebDriver.

Obrázek 5: Architektura - Selenium WebDriver

I přes velký pokrok se WebDriver potýká s určitými nedostatky, které se snaží vyřešit knihovny rozšiřující WebDriver funkcionalitu. Pro porovnání jsem si vybral knihovny wd, selenium-webdriver a WebDriverIO. Nejlépe budou viditelné rozdíly na přiložených ukázkách zdrojového kódu. Všechny 3 přiložené ukázky vykonají totožné uživatelské akce, jedná se o přesměrování stránky pomocí odkazu a následné počkání na zobrazení daných elementů a vložení hodnoty do textového pole.

- **wd**

```
browser.init({browserName:'chrome'}, function() {
  browser.get("https://posta.vsb.cz/", function() {
    browser.elementByCss('a[href="https://posta.vsb.cz/roundcube"]',
        function(err, el) {
      browser.clickElement(el, function() {
        browser.waitForElementByCssSelector('input[name="_pass"]', 3000,
            function() {
          browser.elementByCss('input[name="_pass"]', function(err, el) {
            browser.type(el, 'dfhdh', function() {
              browser.quit();
            });
          });
        });
      });
    });
  });
});
```

Výpis 1: wd asynchronní - vzorový kód

– Obtížnější synchronizace asynchronních metod a z toho plynoucí rozsáhlejší a hůře čitelný zdrojový kód. Na ukázce 1 můžeme vidět synchronizaci pomocí tzv. callback. Callback je funkce, která je volána po dokončení operace a její užití můžete vidět na zmiňované ukázce.

- **Selenium-WebDriver**

```
driver.get('https://posta.vsb.cz/');
driver.findElement(By.linkText('Roundcube')).click();
driver.wait(until.elementLocated( By.id('rcmloginpwd') ), 3000);
driver.findElement(By.id('rcmloginpwd')).sendKeys('topSecret');
driver.quit();
```

Výpis 2: Selenium-WebDriver - vzorový kód

– Tato knihovna je rozšířena o synchronní API, čímž je dosaženo znatelného zjednodušení zdrojového kódu. jenž můžeme vidět na ukázce 2. Není potřeba využívat callback nebo promise pro synchronizaci. Promise, jak název napovídá tzv. příslib, vyjadřuje hodnotu, kterou ještě neznáme a obsahuje handler, který říká co se má dělat až bude známá hodnota příslibu.

- **WebDriverIO**

```
client
    .init()
    .url('https://posta.vsb.cz/')
    .click('a[href*="roundcube"]')
    .waitForVisible('#rcmloginpwd',5000)
    .setValue('#rcmloginpwd','topSecret');
    .end();
```

Výpis 3: WebDriverIO - vzorový kód

– Jedná se o kompletnější řešení se zabudovaným správcem testů. Správce testů umožňuje psát asynchronní příkazy, které budou automaticky vykonány synchronně.

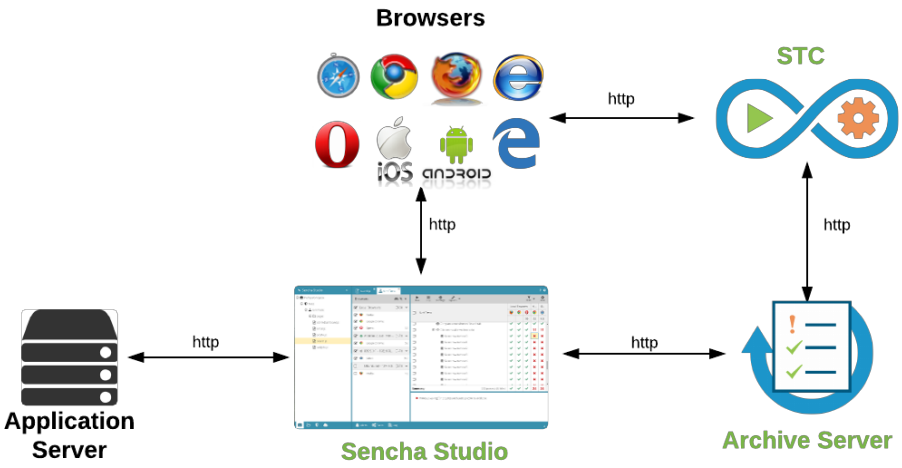
– Rozšíření a zjednodušení APIs. Jednoduchý příklad můžete vidět na ukázce 3

– Možnost rozšíření o užitečné externí funkce, například porovnávání obrázků.

Na základě srovnání výše můžeme vidět rozdíly mezi jednotlivými knihovnami. Knihovna Selenium-WebDriver i WebDriverIO řeší problémy asynchronního vykonávání programů a WebDriverIO navíc nabízí celou řadu nadstavbových funkcí, které jsou srovnatelné s komerčními nástroji. A právě z toho důvodu jsem si pro další porovnávání vybral knihovnu WebDdriverIO.

5.3 Sencha Test

Sencha Test je JavaScript testovací nástroj pro tvorbu unit a end-to-end testů s rozšířenou podporou pro ExtJS aplikace. Sencha Test, dále jen ST, je integrován s Jasmine test framework a z toho důvodu také testu jsou psány v JavaScript.



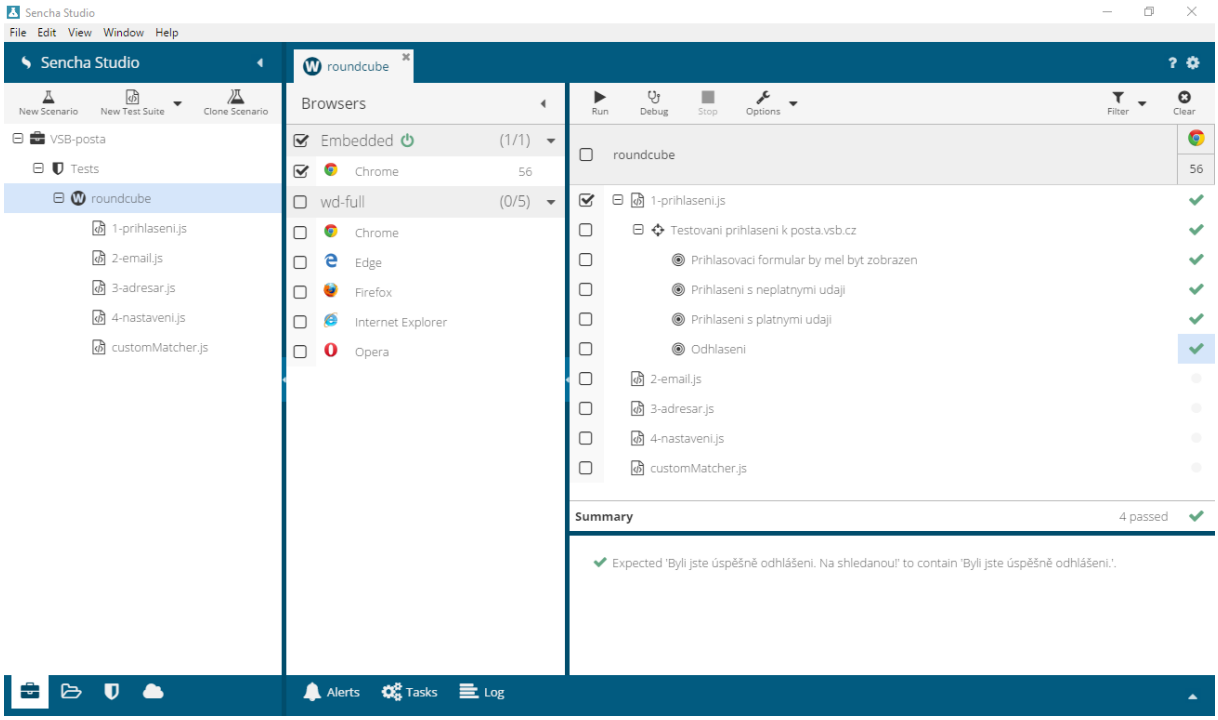
Obrázek 6: Architektura - Sencha Test

Jak můžeme vidět na obrázku 6, hlavní části ST jsou Sencha Studio a Sencha Test Command, dále jen STC. Taky si můžeme všimnout široké podpory webových prohlížečů jak na desktopových, tak mobilních platformách.

Sencha Studio je grafický uživatelský interface pro vytváření, spravování a spouštění testů. Také si můžeme ve studiu velmi přehledně zobrazit výsledky testů a dále s nimi pracovat. Sencha Studio je napsáno pomocí ExtJS frameworku a je spouštěno jako desktopová aplikace pomocí Electronu. Více o této technologii naleznete zde [16]

STC bylo implementováno k zajištění podpory automatického testování v Continuous Integration systémech, jako je například TeamCity nebo Jenkins. STC tedy umožňuje spustit testy z příkazové řádky na lokálním počítači nebo v CI prostředí. Dále je možno pomocí STC si vytvořit svůj vlastní archivační server, na který se budou posílat výsledky automaticky spuštěných testů. Podrobné informace o ST můžeme nalézt v dokumentaci [9]

Na obrázku 7 můžeme vidět Sencha Studio. Na levé straně se nachází stromová struktura testovacích souborů. Uprostřed obrazovky je seznam všech prohlížečů, které je možné použít k testování. Na pravé straně můžeme vidět stromovou strukturu testovacích případů s jejich tvrzeními (assertions). Součástí studia je také textový editor, který obsahuje užitečnou funkci našeptávání.



Obrázek 7: GUI - Sencha Test

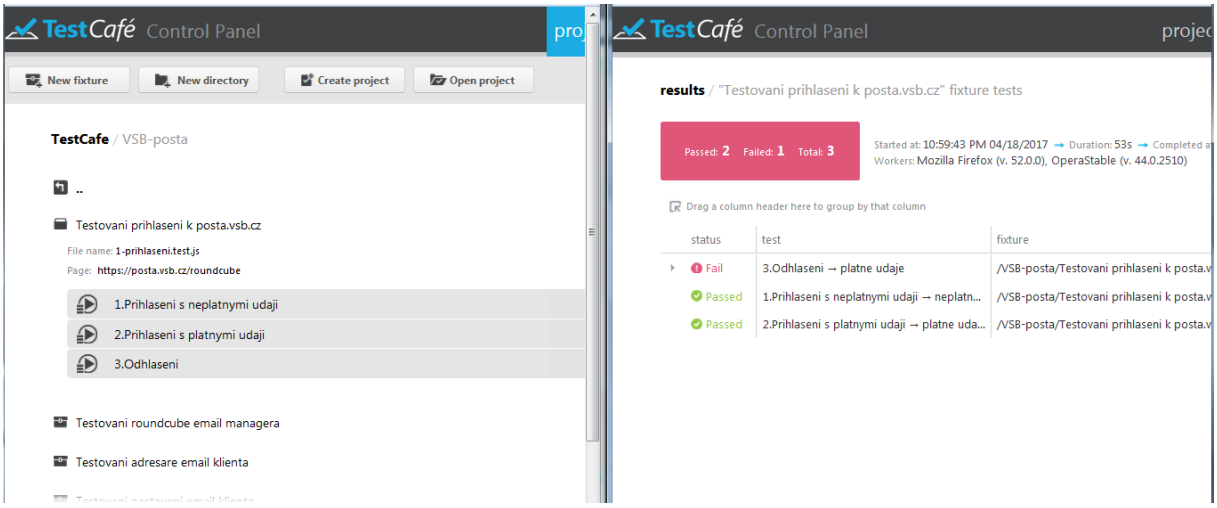
5.4 TestCafe

TestCafe plně využívá node.js technologie a je vyvinut za účelem end-to-end testování webových aplikací. TestCafe se stará o kompletní vykonávání automatických testů od otevření prohlížeče, spuštění testů, zobrazení výsledků a končí generováním reportů. Testy lze pustit na všech moderních prohlížečích bez složité konfigurace. Pro svou funkci TestCafe nevyužívá Selenium technologií, ale pouze node.js a z toho důvodu odpadá závislost na webdriver.

Automatické testy jsou psány v JavaScriptu a testy jsou v souboru organizovány do kategorií, které se nazývají fixture. Soubor může obsahovat více fixture a dále tyto kategorie obsahují již konkrétní testy, které jsou definovány klíčovým slovem test. TestCafe poskytuje bohatou sadu APIs a také vyvinulo celou řadu assertions, které usnadňují tvorbu automatických testů.

TestCafe řeší správu, spouštění testů, zobrazování výsledků pomocí vyvinuté webové aplikace zobrazené v libovolném prohlížeči. Po spuštění TestCafe je tato aplikace dostupná na nakonfigurovaném url. Na obrázku 8 můžeme vidět grafický interface TestCafe. Na levém okně prohlížeče je zobrazena projektová záložka, která obsahuje seznam vytvořených fixtures (například Testování přihlášeni k posta.vsb.cz) a seznam testů. Z této stránky je možné spouštět testy, nahrávat testy a také upravovat. Na pravém okně můžeme vidět výsledky vykonaných testů.

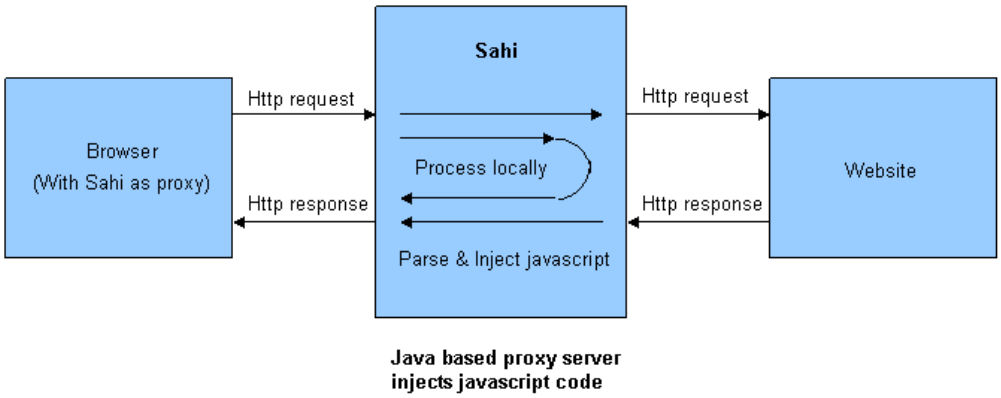
Detailní informace najde na stránkách TestCafe [17]



Obrázek 8: GUI - TestCafe

5.5 Sahi

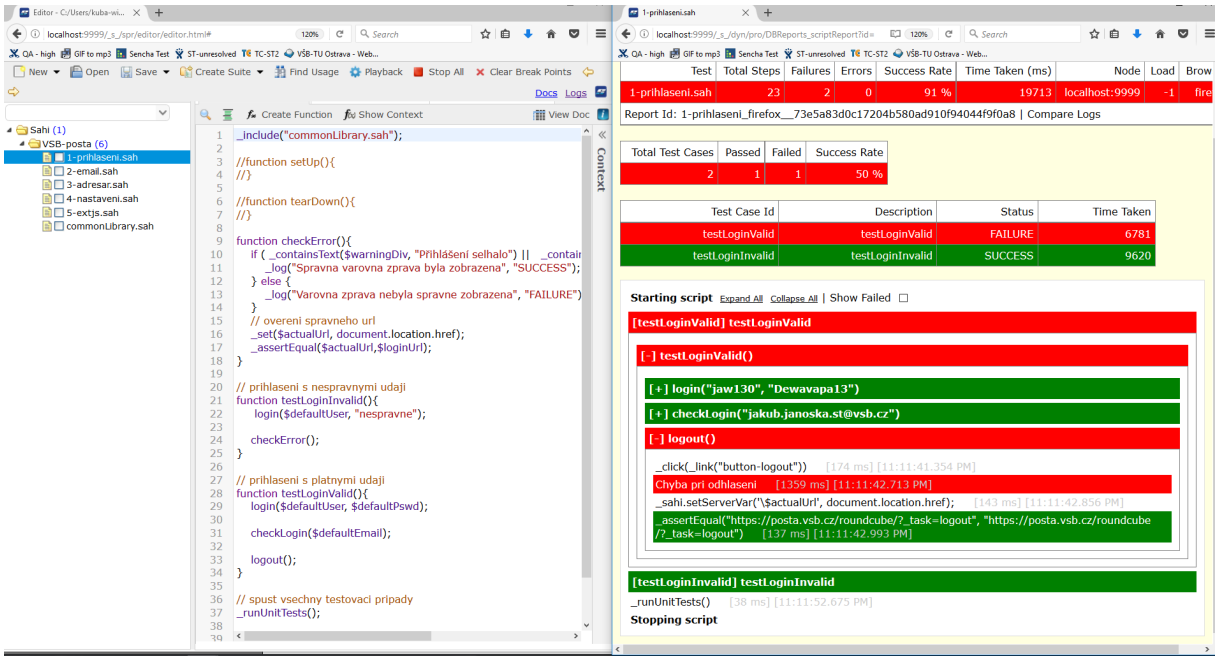
Dalším zajímavým nástrojem pro tvorbu automatických testů je Sahi. Jméno pochází z Hindštiny a znamená správný, způsobilý. Používá tzv. Sahi skriptovací jazyk, který má stejnou syntaxi jako JavaScript s jediným rozdílem, proměnné musí začínat s prefixem $. Sahi skripty jsou analyzovány a upravovány, aby automaticky byly schopny zvládat čekání na dané prvky či události, obnovit se po chybách, zaznamenávat výsledky. Modifikované skripty jsou plně validní s JavaScript a spouštěny pomocí Rhino engine.



Obrázek 9: Architektura - Sahi

Z architektury uvedené na obrázku 9 můžeme vidět, že Sahi je prostředníkem mezi webovým prohlížečem a testovanou webovou stránkou či aplikací. Důležité je upravit nastavení proxy webového prohlížeče k využívání Sahi proxy serveru. Toto nastavení je provedeno automaticky po spuštění Sahi.

Rhino engine je JavaScriptový interpret bežící uvnitř Sahi proxy. Pouze nutné příkazy jsou odesílány na prohlížeč, o zbytek se stará Rhino engine uvnitř Sahi proxy. Rhino je spuštěno na JVM a z toho pramení obrovská výhoda. Sahi skripty mohou přímo využívat také java kódu.



Obrázek 10: GUI - Sahi

Sahi řeší správu, vykonávání testů a zobrazení výsledků podobně jako předchozí popisovaný nástroj, TestCafe. Na obrázku 10 můžeme vidět grafický interface webové aplikace. Na levé straně můžeme vidět seznam testovacích souborů. Sahi využívá soubory s příponou .sah, ale umí také pracovat se soubory typu .js. Uprostřed se nachází textový editor a na pravém okně můžeme vidět výsledky testů. Konkrétně můžeme vidět report z testování přihlášení i s konkrétními assertion.

Detailní informace najde na stránkách Sahi [18]

6 Testované aplikace

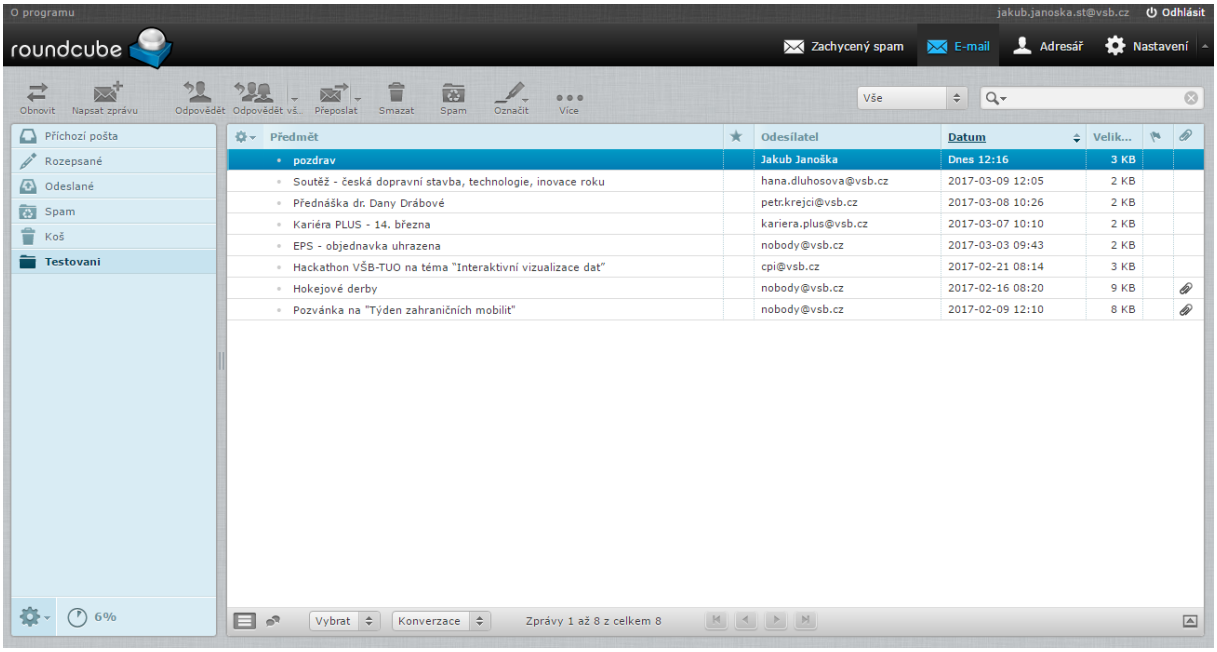
V této kapitole se budeme věnovat vybraným webovým aplikacím, na které dále budeme vytvářet automatické testy. Automatické testy budou implementovány ve všech 4 nástrojích pro tvorbu automatických testů popsaných v kapitole 5. Zásadním kritériem automatických testů je co nejbližší shoda s uživatelskými kroky a uživatelským použitím aplikace.

Pro svou práci jsem si vybral komplexní webovou aplikaci Roundcube, kde budou implementovány typické uživatelské postupy od přihlášení až po změnu v nastavení. Ext JS aplikaci se budeme věnovat pouze okrajově a budeme se na ní snažit vyzdvihnout přínosy zabudované podpory použitého frameworku.

Následující činnost odpovídá analýze a návrhu, jenž jsme si popsali v teoretické části práce v kapitole 2.2.2. Jak už jsme si řekli dříve, budeme se snažit porovnat vybrané nástroje. I v praxi se s takovouto činností můžeme setkat v případě výběru nového nástroje. V takovém případě se provádí tzv. proof-of-concept, kde podstatou je ověření, zda zadaný nástroj splňuje očekávání. A tímto bude také ovlivněna naše analýza a návrh. Podstatou není kompletní testování webových aplikací, ale základní cíl je identifikovat specifické oblasti, specifické funkce, které nám pomůžou rozhodnout, jaký nástroj je nejvhodnější. Bližší analýzy najdeme u konkrétních aplikací.

6.1 Roundcube - webový emailový klient

`https://posta.vsb.cz/roundcube/`



Obrázek 11: Roundcube - webový emailový klient

Roundcube je open source emailový klient. Využívá IMAP protokolu a je napsán v jazyce PHP. Více informací o tomto open source projektu naleztete zde [13].

Pro testování jsem si vybral tuto aplikaci, protože je spjata s VŠB a navíc se jedná o komplexní webovou aplikaci s přihlašovacím formulářem, správcem emailů a mnoho další funkcionality, kterou v této práci otestujeme. Na obrázku 11 můžeme vidět grafický interface emailového klienta.

V rámci analýzy jsme se seznámili s emailovým klientem Roundcube a provedli jsme tzv. průzkumné testování. Výsledkem průzkumného testování je identifikování konkrétních oblastí, na které se dále pokusíme vytvořit automatické testy. Nejdůležitější oblasti se týkají přihlášení, správy emailů, správy kontaktů a nastavení webového klienta.

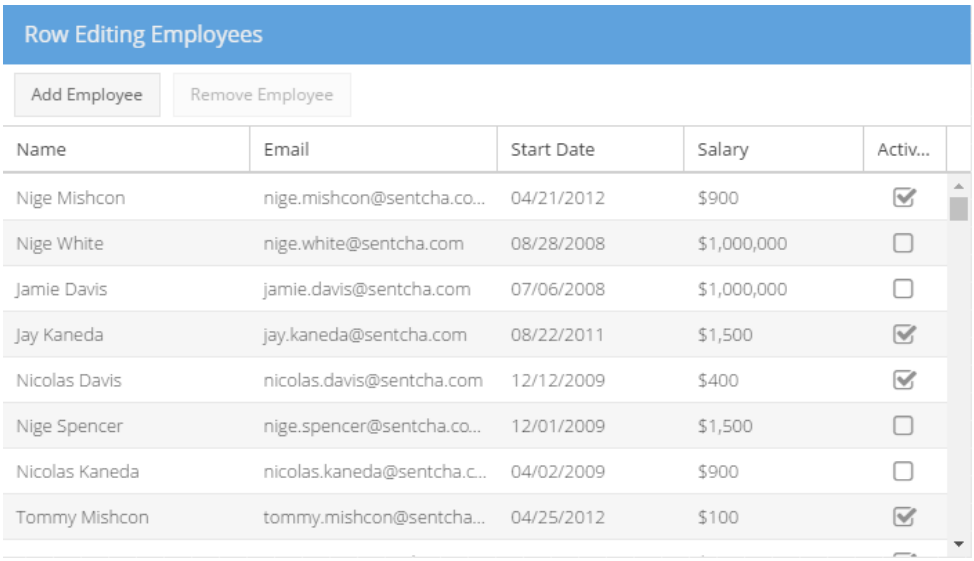
A nyní již přecházíme do fáze návrhu, kde si vytvoříme seznam konkrétních testovacích případů. Pro tento účel jsme vytvořili jednoduchou tabulku 3 se seznamem testovacích případů, které budeme implementovat.

ID	**Testovací případ**
Roundcube - Přihlášení	
001	Přihlášení s neplatnými údaji
002	Přihlášení s platnými údaji
003	Odhlášení
Roundcube - Email	
004	Odeslání emailu
005	Otevření emailu
006	Smazání emailu
Roundcube - Adresář	
007	Vytvoření kontaktu
008	Editace kontaktu
009	Odebrání kontaktu
Roundcube - Nastavení	
010	Změna jazyka
ExtJS - Row Editing	
011	Ověření správného zobrazení aplikace
012	Změna emailu v tabulce
013	Ověření změny emailu

Tabulka 3: Regresní testovací sada pro emailový klient Roundcube

6.2 Ext JS aplikace - Row Editing

`http://examples.sencha.com/extjs/6.2.0/examples/kitchensink/?modern_#row-editing`



Obrázek 12: Ext JS - Modern Kitchen Sink - Row Editing

Aplikace Row Editing je jedna z mnoha ukázkových příkladů od společnosti Sencha, která demonstruje funkce jednotlivých Ext JS komponent. V konkrétním příkladě je užita komponenta typu grid pro jednoduché zobrazení dat v tabulce. Celá aplikace je napsána v JavaScript frameworku Ext JS.

V rámci analýzy a návrhu jsme vytvořili další testovací případy pro vybrání konkrétního záznamu v tabulce a jeho úpravu. Konkrétní vytvořené testovací případy nalezneme také v tabulce 3.

Kompletní implementaci automatických testů naleznete v příloze. V kapitole 7.2 můžete nalézt ukázky zdrojových kódů.

7 Automatické testování

Touto kapitolou se dostáváme k samotné implementaci automatických testů. Ale ještě dříve než si ukážeme konkrétní ukázky zdrojových kódů, uveďme si nejčastější problémy, kterým čelí softwarový inženýr v průběhu implementace automatických testů.

Automatické testy se vytvářejí nad webovou aplikací. V průběhu životního cyklu každé aplikace dochází ke změnám z mnoha důvodů. A právě na toto by měl softwarový inženýr myslet a snažit se vytvářet testy robustní, snadno spravovatelné a velmi dobře čitelné. Robustností je myšlena hlavně správná volba selektorů a precizní synchronizace testů, které jsou odolné vůči menším změnám.

Nejčastější problémy se správnou synchronizací testů souvisí také s AJAX technologií či při přesměrování stránek. Z toho důvodu je nutná implementace vhodných čekání. Každý nástroj implementuje čekání jiným způsobem a také to bude předmětem porovnání jednotlivých nástrojů v kapitole 8 srovnání.

Snadná spravovatelnost testů je důležitá z toho důvodu, že se již při tvorbě testů musí počítat s budoucími změnami, které jistě přijdou. A proto je vhodné si vytvořit proměnné či funkce pro selektory, které budou dále používány v kódu. V případě změny, postačí upravit pouze proměnné nebo funkce a není potřeba procházet celý kód a upravovat každý lokátor zvlášť.

Také je výhodou si jednotlivé testovací kroky rozdělit do vhodně pojmenovaných funkcí. Tímto zlepšujeme nejen spravovatelnost testů, ale také jejich čitelnost. Jednoduchá čitelnost i pro netechnické osoby je velmi zásadním kritériem, protože automatické testy nemusí sloužit pouze pro výkon testování, ale mohou také nahrazovat detailní specifikaci testovacích případů.

Dalším problémem automatických testů je podpora různých platforem operačních systémů a různých prohlížečů. Jsou zde snahy sjednotit webové technologie, ale stále se naráží na specifické problémy mezi jednotlivými webovými prohlížeči a optimalizace testů na veškeré podporované prohlížeče můžeme být velmi problematická. V tomto případě velmi záleží na volbě nástroje pro tvorbu automatických testů a více si uvedeme v kapitole 8 srovnání.

7.1 Identifikace elementů

Základem pro vykonání jakékoliv uživatelské interakce je nalezení správného elementu a k tomu slouží selektory, kterým se budeme věnovat v této kapitole.

Jednotlivé nástroje používají různé strategie pro výběr daných elementů či komponent. My si uvedeme a podrobně rozebereme ty nejznámější a nejužívanější technologie, mezi nichž patří CSS, DOM či XPATH selektory.

Speciálním případem je JQuery technologie pro výběr prvků. V tomto případě se jedná o celý JavaScriptový framework, ale je také používána jedním porovnávaným nástrojem pro výběr elementů, a právě proto JQuery zařadíme do této kapitoly.

Pro maximální přehlednost použijeme konkrétní ukázku části HTML a dále na této struktuře uvedeme příklady selektorů vytvořených podle XPath, CSS a DOM. Vytvořené selektory slouží

pouze jako ukázka, některé konstrukce selektorů jsou zbytečně složité a slouží pro jednoduchou a srozumitelnou ukázku složitějších struktur. Avšak všechny selektory jsou funkční a ověřeny pomocí konzole z nástrojů pro vývojáře v Chrome. Použitá HTML ukázka 4 je použita z emailového klienta Roundcube, konkrétně se jedná o první řádek hlavičky obsahující jméno uživatele a tlačítko pro odhlášení.

```
<div id="topline">
   <div class="topleft">
      <a class="about-link" onclick="UI.show_about(this);return false" id="
          rcmbtn100" href="#">About</a>
   </div>
   <div class="topright">
      <span class="username">jakub.janoska.st@vsb.cz</span>
      <a class="button-logout" id="rcmbtn101" href="./?_task=logout" onclick="
          return rcmail.command('switch-task','logout',this,event)">Logout</a>
   </div>
</div>
```

Výpis 4: Ukázka HTML - část emailového klienta roundcube

7.1.1 XPath, CSS a DOM selektory

Element	XPATH
element typu ***span***	"//span"
element s ***id rcmbtn101***	"//a[@id='rcmbtn101']"
element ***span*** s atributem **class** a hodnotou atributu **username**	"//span[@class='username']"
element ***a*** s atributem ***onclick*** obsahující část hodnoty ***UI.show_about***	"//a[contains(@onclick,'UI.show_about')]"
element ***a*** specifikovaný pomocí 2 atributů	"//a[contains(@href,'logout')][@class='button-logout']"
element ***span*** obsahující část textu ***janoska***	"//span[contains(string(),'janoska')]"
víceúrovňový dotaz na ***div*** element s ***class topleft***, jehož rodič je ***div*** element s ***id topline***	"//div[@id='topline']/div[@class='topleft']"
2. ***div*** potomek ***div*** elementu s ***id topline***	"//div[@id='topline']/div[2]"
následující sourozenec elementu div s class topleft	"//div[@class='topleft']/following-sibling::*"
předcházející sourozenec elementu ***div*** s ***class topleft***	"//div[@class='topright']/preceding-sibling::*"

Tabulka 4: Ukázka XPath selektorů

Element	CSS
element typu ***span***	css="span"
element s ***id rcmbtn101***	css="#rcmbtn101"
element ***span*** s atributem **class** a hodnotou atributu **username**	css="span[class='username']"
element ***a*** s atributem ***onclick*** obsahující část hodnoty ***UI.show_about***	css="a[onclick*='UI.show_about']"
element ***a*** specifikovaný pomocí 2 atributů	css="a[href*='logout'][class='button-logout']"
element ***span*** obsahující část textu ***janoska***	-
víceúrovňový dotaz na ***div*** element s ***class topleft***, jehož rodič je ***div*** element s ***id topline***	css="#topline>.topleft"
2. ***div*** potomek ***div*** elementu s ***id topline***	css="#topline>div:nth-child(2)"
následující sourozenec elementu ***div*** s ***class topleft***	css=".topleft+*"
předcházející sourozenec elementu ***div*** s ***class topleft***	-

Tabulka 5: Ukázka CSS selektorů

Element	DOM
element typu ***span***	document.getElementsByTagName('span')
element s ***id rcmbtn101***	document.getElementById('rcmbtn101')
element ***span*** s atributem **class** a hodnotou atributu **username**	document.getElementsByClassName('username')[0]
element ***a*** s atributem ***onclick*** obsahující část hodnoty ***UI.show_about***	-
element ***a*** specifikovaný pomocí 2 atributů	-
element ***span*** obsahující část textu ***janoska***	-
víceúrovňový dotaz na ***div*** element s ***class topleft***, jehož rodič je ***div*** element s ***id topline***	document.getElementById('topline').getElementsByClassName('topleft')
2. ***div*** potomek ***div*** elementu s ***id topline***	document.getElementById('topline').getElementsByTagName('div')[1]
následující sourozenec elementu ***div*** s ***class topleft***	document.getElementsByClassName('topleft')[0].nextSibling
předcházející sourozenec elementu ***div*** s ***class topleft***	document.getElementsByClassName('topright')[0].previousSibling

Tabulka 6: Ukázka DOM selektorů

Z uvedených ukázek v tabulkách 4, 5 a 6 můžeme vidět, že XPath je nejuniverzálnější selektor, pomocí něhož umíme vyhledat i element, který obsahuje část textu a také můžeme procházet DOM stromovou strukturu směrem nahoru.

Naopak CSS selektory mají jednodušší zápis a některé jejich nedostatky mohou být řešeny pomocí JQuery.

Jak můžeme vidět DOM lokátory mají pouze omezenou funkcionalitu i jejich zápis je komplikovanější. Také bych chtěl upozornit na použití nextSibling a previousSibling, které je velmi nespolehlivé a často je namísto elementu vrácen pouze prázdný textový řetězec.

7.1.2 jQuery selektor

Základem jQuery selektoru je CSS selektor, který je obohacený o vlastní selektory a další funkce. Pomocí tohoto selektoru můžeme vyhledávat HTML elementy na základě jejich jména, id, třídy, typu, atributu, hodnoty a mnohem více.

jQuery selektor začíná znakem dolaru a selektor se nachází uvnitř kulatých závorek.

Vzhledem k tomu, že vychází z CSS selektorů, nebudeme uvádět znovu ukázky JQuery selektorů. Uvedl bych pouze jeden selektor, který nebyl možný pomocí CSS, ale JQuery jej podporuje. Jedná se o výběr elementu span obsahující část textu janoska.

Pomocí JQuery, by vypadal selektor takto: $("$span : contains('janoska')$"). Kompletní dokumentaci k oblasti jQuery selektorů naleznete zde [24].

7.2 Ukázka kódu

V této části si postupně ukážeme ukázky zdrojových kódů ve všech porovnávaných nástrojích. Pro ukázku jsem si zvolil testovací případ Přihlášení s platnými údaji. Kompletní seznam testovacích případů naleznete zde 3. Kompletní implementaci testovacích případů naleznete na přiloženém CD. Na přiloženém CD také najdete vytvořená dema, která ukazují běh automatických testů a demonstrují praktické využití porovnávaných nástrojů.

Zde v textu si představíme ukázku, již zmiňovaného testovacího případu, která obsahuje vytváření selektorů pro přihlašovací stránku k emailovému klientu. Dále si ukážeme implementaci funkce pro přihlášení a v této funkci uvidíte nejpoužívanější APIs, jako je například čekání, klik či vložení hodnoty.

7.2.1 WebDriverIO - Implementace testovacího případu přihlášení s platnými údaji

V ukázce 5 můžete vidět vytvoření funkcí pro selektory pomocí nástroje WebDriverIO. V tomto nástroji můžeme využití css a XPath selektorů. Takovéto vytvoření selektorů velmi usnadňuje jejich správu a také velmi ulehčuje implementaci kódu. Při vytváření konkrétních testů už se nezabýváme selektory a pouze voláme vytvořené funkce k identifikování potřebných elementů.

```
// selektory pro elementy na prihlasovacim formulari
 loginForm: function () {
```

```
    // css selektor pro prihlasovaci formular id=login-form
    return $("#login-form");
},
usernameField: function () {
    // XPath selektor pro textove pole 'Username'
    return $("//input[@name='_user']");
},
passwordField: function () {
    return $("//input[@name='_pass']");
},
submitButton: function() {
    return $("//input[@type='submit']");
},
warningMessage: function() {
    return $("//div[@class='warning']");
},
message: function() {
    return $("#message");
}
```

Výpis 5: Ukázka lokátorů implementovaných pomocí WebDriverIO

Implementaci přihlašovací funkce naleznete v ukázce 6. V této ukázce můžete vidět použití WebDriverIO API waitForExist(3000), které zajišťuje synchronizaci testů a čekání na přihlašovací formulář. Pomocí parametru definujeme maximální dobu čekání a v případě vypršení definované doby dojde k selhání testu. Dále si můžeme všimnout API setValue(name), které zajišťuje vložení hodnoty do pole. Pro klik slouží API s názvem click(). Můžeme vidět, že názvy APIs jsou velmi intuitivní a s použitím vhodného pojmenování funkcí a proměnných, také přidáním vhodných komentářů je kód velmi dobře čitelný a už na první pohled je zřejmé, co daná funkce vykonává. Pro kontrolování vykonaných akcí slouží assertion, v našem případě je použito expect(name).toBe(lUser.getValue). Tato assertion nám porovná jestli se shoduje očekávaná hodnota s aktuální. Assertion typu toBe() je velmi striktní a je potřeba naprosto totožných hodnot. V případě neshody dojde k selhání testu, s příslušnou chybovou hláškou, že se hodnoty neshodují. Daný testovací případ je ukončen s negativním výsledkem a pokračuje se dalším testem.

```
login: function (name, password) {
    this.loginForm().waitForExist(3000);
    // promenna s textovym polem pro zadani uzivatelskeho jmeno
    var lUser = this.usernameField();
    lUser.setValue(name);
```

```
        // assertion - byla vlozena spravna hodnota?
        expect(name).toBe(lUser.getValue());
        var lPswd = this.passwordField();
        // vlozeni hesla do prislusneho textoveho pole
        lPswd.setValue(password);
        expect(password).toBe(lPswd.getValue());
        // potvrzeni prihlaseni klikem na tlacitko
        this.submitButton()
            .click();
    }
```

Výpis 6: Ukázka login funkce pomocí WebDriverIO

Na ukázce 7 můžeme vidět implementaci testovacího případu pomocí Jasmine framework, který byl popsán v 4.3 a je možné jej využít ve WebDriverIO. Další možnou variantou by mohlo být využití Mocha framework. Více informací o Mocha framework naleztene zde [6]. Vidíme specs, tedy it(), který definuje testovací případ a uvnitř it(), je konkrétní implementace testu. Můžeme zde vidět volání funkce login(), kterou jsme si popsali výše. V této ukázce bych zvýraznil použití assertion expect(roundcube.username).toContain(roundcube.usernameValue().getText()). Tato assertion není tak striktní jako v předchozím případě a postačuje shoda části textového řetězce uvedeného v toContain().

Na přiložené ukázce vidíte jednoduchou kontrolu přihlášení na základě čekání na element obsahující list příchozích emailů a také si ověříme správné uživatelské jméno, které je zobrazeno po přihlášení do klienta. Také si ověříme pomocí url, že došlo ke správnému přesměrování stránky. K tomu slouží API getUrl().

```
it("Prihlaseni s platnymi udaji", function () {
   roundcube.login(roundcube.user, roundcube.pswd);
   // kontrola prihlaseni do roundcube
   // overeni zda je zobrazen seznam emailu
   roundcube.emailList().waitForVisible(5000);
   // overeni spravneho url (zda se stranka spravne presmerovala po prihlaseni)
   expect(browser.getUrl()).toContain("_task=mail");
   // overeni emailu prihlaseneho uzivatele
   expect(roundcube.username).toBe(roundcube.usernameValue().getText());
});
```

Výpis 7: Ukázka implementace TC 002 - Přihlášení s platnými údaji pomocí WebDriverIO

7.2.2 Sencha Test - Implementace testovacího případu přihlášení s platnými údaji

Nyní se přesouváme k ukázkám zdrojových kódů vytvořených nástrojem Sencha Test. Na ukázce 8, můžeme vidět definování funkcí pro selektory elementů nacházejících se na přihlašovacím formuláři. Kód je velmi podobný jako v případě WebDriverIO. Zde můžeme vidět užití selektorů typu At-path, XPath a css.

```
var RoundcubeLogin = {
        form: function() {
            // At-path lokator pro ID login-form
            return ST.element("@login-form");
        },
        usernameField: function() {
            // XPath lokator pro textove pole 'Username'
            return ST.element("//input[@name='_user']");
        },
        passwordField: function() {
            // css lokator pro textove pole 'Password'
            return ST.element(">>#rcmloginpwd");
        },
        submitButton: function() {
            return ST.element("//input[@type='submit']");
        },
        warningMessage: function() {
            return ST.element("//div[@class='warning']");
        }
    };
```

Výpis 8: Ukázka lokátorů implementovaných pomocí Sencha Test

Následuje ukázka 9 implementace přihlašovací funkce pomocí Sencha Test. Opět si můžeme všimnout, že funkce je velmi dobře čitelná a přehledná. Tomu pomáhá intuitivní pojmenování ST APIs. Můžeme si všimnout visible() API, které čeká na zobrazení elementu. Pro vkládání hodnot slouží API type(username). Další API sloužící pro vykonávání uživatelských akcí je API click(), které se stará o kliknutí na daný element. API textLike(username) slouží pro ověření, zda element obsahuje danou hodnotu. Pro identifikaci elementů využíváme funkcí vytvořených v předchozí ukázce.

```
function login (username, password) {
    // je prihlasovaci formular zobrazen?
    RoundcubeLogin.form()
     .visible();
```

```
    // vloz uzivatelske jmeno
    RoundcubeLogin.usernameField()
        .type(username)
        .textLike(username);
    // vloz heslo
    RoundcubeLogin.passwordField()
        .type(password)
        .textLike(password);
    // stiskni tlacitko Prihlasit
    RoundcubeLogin.submitButton()
        .click();
}
```

Výpis 9: Ukázka login funkce v ST

I Sencha Test využívá Jasmine framework pro vytváření testovacích případů. Konkrétní ukázku testovacího případu v Sencha Test můžete vidět v ukázce 10. Testovací případ v Jasmine terminologii specs je definována pomocí it(), kde textový řetězec určuje jméno testovacího případu a funkce uvnitř it() definuje chování testovacího případu. Běh testů uvnitř it() je automaticky synchronní. V případě Sencha Test není běh testu v it() ukončen po selhání assertion, ale dále se dokončí.

Také zde je velmi dobrá čitelnost kódu. Jako první voláme login() funkci, která zajišťuje přihlášení k emailovému klientu. Následuje ověření přihlášení pomocí visible() API nad elementem obsahující seznam příchozích emailů. Pomoci getUrl() získáváme hodnotu aktuálního url a ověřujeme pomocí assertion, jestli došlo ke správnému přesměrování stránky po přihlášení. Vidíme použití API execute(), které zajišťuje běh kódu přímo na testované stránce. A právě pomocí execute() API můžeme získat hodnoty zobrazené na testované stránce a například si ověřit správné uživatelské jméno po přihlášení.

```
it("Prihlaseni s platnymi udaji", function() {
    login (user, pswd);
    // kontrola prihlaseni do roundcube
    // overeni zda je zobrazen seznam emailu
    ST.element("@messagelistcontainer")
        .visible()
        .getUrl(function(url) {
          // overeni url
           expect(url).toContain("_task=mail");
        });
    // overeni emailu prihlaseneho uzivatele
    ST.element("//span[@class='username']")
```

```
            .visible()
            .execute(function(el) {
                return el.getText();
            })
            .and(function(future) {
                expect(future.data.executeResult).toBe(username);
            });
    });
```

Výpis 10: Ukázka implementace TC 002 - Přihlášení s platnými údaji pomocí ST

7.2.3 TestCafe - Implementace testovacího případu přihlášení s platnými údaji

Následující kódy ukazují řešení pomocí TestCafe. Ukázka 11 definuje funkce pro identifikování elementů na přihlašovací stránce k emailovému klientu. V případě TestCafe můžeme použít DOM, css a JQuery selektory, které jsou také všechny použity v přiložené ukázce.

```
// lokatory pro prihlasovaci formular
var RoundcubeLogin = {
        form: function() {
            // JQuery objekt identifikovany pomoci CSS selektoru
            // div s ID = login-form
            // prihlasovaci formular
            return $('div#login-form');
        },
        usernameField: function () {
            // JQuery objekt identifikovany pomoci CSS selektoru
            // input element s atributem name=_user
            // textove pole 'username'
            return $('input[name=_user]');
        },
        passwordField: function() {
            // DOM element s ID = rcmloginpwd
            // textove pole 'username'
            return document.getElementById('rcmloginpwd');
        },
        submitButton: function() {
            return $('input[type=submit]');
        },
        warningMessage: function() {
```

```
            return $('div.warning');
        }
    }
```

Výpis 11: Ukázka lokátorů implementovaných pomocí TestCafe

Ukázka 12 se už znatelně liší od implementací pomocí WebDriverIO a Sencha Test. Pomocí @mixin si můžeme vytvořit test, který je možné volat z jiných testů. V podstatě @mixin můžeme přirovnat k funkci. V našem případě si pomocí @mixin vytvoříme test pro přihlášení uživatele k emailovému klientu. Test se dále dělí na kroky. Například první krok 'Vloz uzivatelske jmeno' se stará o vložení hodnoty do určeného elementu. Element je určen pomocí funkce vytvořené v předchozí ukázce a vložení hodnoty je zajištěno pomocí API act.type(). Krok může obsahovat pouze jedno API nebo pouze jednu assertion. Vykonávání kroků je synchronní, navíc každý krok obsahuje textové pole s popisem kroku. Právě z tohoto důvodu jsou testy dobře čitelné a není potřebné přidávat další komentáře.

V ukázce můžeme dále vidět použití act.click() API a také assertion eq(), která porovnává, zda jsou oba objekty totožné.

```
'@mixin' ['login'] = {
    'Vloz uzivatelske jmeno': function () {
        act.type(RoundcubeLogin.usernameField(), this.user);
    },
    'Zkontroluj uzivatelske jmeno': function () {
        eq(RoundcubeLogin.usernameField().val(), this.user);
    },
    'Vloz heslo': function () {
        act.type(RoundcubeLogin.passwordField, this.pswd);
    },
    'Stiskni tlacitko Login': function () {
        act.click(RoundcubeLogin.submitButton());
    }
};
```

Výpis 12: Ukázka login funkce implementované pomocí TestCafe

Následuje ukázka 13 implementace testovacího případu přihlášení s platnými údaji. Každý testovací případ je v TestCafe definován pomocí @test a podobně jako @mixin se test skládá z kroků. Pomocí @testCases si definujeme proměnné, které budou použity v testu. V příkladu si můžeme všimnout API act.waitFor(element), které se stará o čekání na načtení listu emailu a tím je také zkontrolováno, že byl uživatel přihlášen do emailového klienta. Také porovnáváme pomocí assertion správnost uživatelského jména. V případě TestCafe jsem nenašel API pro získání aktuálního url pro ověření přesměrovaní pomocí url.

```
'@test'['2.Prihlaseni s platnymi udaji'] = {
    '@testCases': [
        {'@name': 'platne udaje', user: userG, pswd: pswdG}
    ],
    'Prihlaseni': '@mixin login',
    'Kontrola zobrazeni seznamu emailu': function () {
        act.waitFor('#messagelistcontainer');
    },
    'Overeni emailu prihlaseneho uzivatele': function () {
        eq($("span[class='username']").text(), usernameG);
    }
};
```

Výpis 13: Ukázka implementace TC 002 - Přihlášení s platnými údaji pomocí TestCafe

7.2.4 Sahi - Implementace testovacího případu přihlášení s platnými údaji

Dostáváme se k poslednímu porovnávanému nástroji a to je Sahi. V první ukázce 14 můžeme vidět, jak Sahi řeší identifikování elementů. Již pro identifikaci elementů používá vlastní API, jako například _div(selektor), _submit(selektor) či _textbox(selektor). Z uvedených příkladů vyplývá, že Sahi implementuje pro každý element své API a také selektory používají vlastní, jenž jsou vytvořeny nad DOM elementy. Sada API je velmi bohatá a nesetkal jsem se s chybějící API pro potřebný element.

```
// identifikator pro textbox se jmenem _user
var $usernameField = _textbox("_user"),
// JSON identifikator s atributem name a hodnotou _pass pro password type
$passwordField = _password({name:"_pass"}),
// ccs class identifikator pro tlacitko submit
$submitButton = _submit("button mainaction"),
// identifikace pomoci id
$messageDiv = _div("message"),
// identifikovani elementu pomoci class
$warningDiv = _div("warning");
```

Výpis 14: Ukázka lokátorů implementovaných pomocí Sahi

Ukázka 15 obsahuje vytvoření funkce pro přihlášení pomocí Sahi. V ukázce můžeme vidět celou řadu APIs, které se starají o uživatelské akce jako _setValue($passwordField,$tempPswd) a _click($submitButton). Dále je v ukázce _assertEqual($tempPswd, $passwordField.value) APIs, které slouží jako assertion pro porovnávání očekávaného a aktuálního výsledku.

```
    function login ($tempName, $tempPswd) {
      // vloz uzivatelske jmeno
      _setValue($usernameField, $tempName);
      _setValue($passwordField,$tempPswd);
      // kontrola vlozenych hodnot
      _assertEqual($tempName, $usernameField.value);
      _assertEqual($tempPswd, $passwordField.value);
      // stiskni tlacitko Prihlasit
      _click($submitButton);
   }
```

Výpis 15: Ukázka login funkce v Sahi

Ukázka 16 ukazuje vytvoření testovacího případu v případe Sahi. Testovací případ se odlišuje od ostatních funkcí klíčovým slovem test. Tedy pokud jsou spouštěny testy nad daným testovacím souborem, tak jsou vyhledány a spuštěny všechny soubory obsahující slovo test. Z tohoto příkladu bych zmínil API _containsText(_span("username"), $tempUser)), které slouží pro kontrolu uživatelského jména a vrací hodnotu true, když element obsahuje daný text. Toto API samo o sobě nedokáže vyhodnotit jestli test byl úspěšný, a proto jej používáme s podmínkou if() a pro vložení úspěšného nebo neúspěšného výsledku testu slouží API _log(msg, typLogu). Kde typem logu můžeme definovat úspěch či neúspěch testu.

```
function testLoginValid(){
   login($defaultUser, $defaultPswd);
   var $actualUrl;
   // ziskani aktualniho url a porovnani s ocekavanym url
   _set($actualUrl, document.location.href);
   _assertEqual($actualUrl,$logedUrl);
   // hledani pouze visible elementu v DOM
   _setStrictVisibilityCheck(true);
   // porovnani jestli odpovida aktualni email s ocekavanym
   if ( _containsText(_span("username"), $tempUser) ) {
      _log("Uzivatel " + $tempUser + " se uspesne prihlasil", "SUCCESS");
   } else {
      _log("Uzivatel " + $tempUser + " neni prihlasen", "FAILURE");
   }
   // hledani veskerych elementu v DOM (defaultni chovani)
   _setStrictVisibilityCheck(false);
}
```

Výpis 16: Ukázka implementace TC 002 - Přihlášení s platnými údaji pomocí Sahi

8 Srovnání

Dostáváme se k finální kapitole, kde se pokusíme porovnat použité nástroje. Výsledkem porovnání každé kategorie bude graf s vynesenou mírou spokojenosti. Míra spokojenosti je stupnice od 1 do 5, kde 5 je nejlepší.

Definovali jsme si 7 kategorií, jenž budou dále porovnány. Jelikož se kritéria pro výběr použitého nástroje pro tvorbu automatických testů velmi liší v závislosti na projektu, nelze tyto kritéria seřadit podle priority.

Protože jsou všechny nástroje aktivně podporovány a neustále se pracuje na jejich vylepšení a nových vydání, uvedeme si použité verze porovnávaných nástrojů:

- WebDriverIO: verze 4.6.2
- SenchaTest: verze 2.0.2
- TestCafe: verze 15.1.3
- Sahi: verze 6.3.2

Byly použity poslední vydané verze jednotlivých nástrojů. Poslední kontrola aktuálních verzí proběhla 15. dubna 2017.

8.1 Podporované webové prohlížeče a operační systémy

Podpora operačních systémů a webových prohlížečů jednoznačně ovlivňuje výběr vhodného nástroje pro tvorbu automatických testů a vždy je potřeba si udělat důkladný průzkum této oblasti. Pojďme se podívat na výsledky našeho pozorování.

✓ Prohlížeč je připraven bez dalších konfigurací.

⌛ Prohlížeč či cloud je použitelný s další konfigurací či obtížemi.

✗ Prohlížeč či cloud není podporován.

	Lokální prohlížeče						Cloud	
	Chrome	Firefox	IE	Edge	Safari	Opera	BrowserStack	SauceLabs
WebDriverIO	⌛	⌛	⌛	⌛	⌛	⌛	⌛	⌛
Sencha Test	✓	⌛	⌛	⌛	⌛	✗	⌛	⌛
TestCafe	✓	✓	✓	✓	✓	✓	⌛	⌛
Sahi	✓	✓	✓	⌛	✓	⌛	✗	✗

Tabulka 7: Podporované webové prohlížeče a cloud řešení

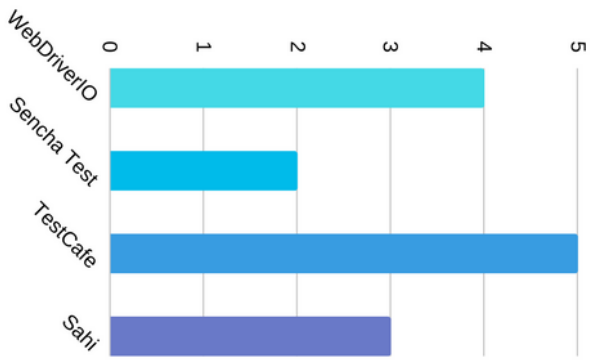
Pro větší přehlednost jsem použil tabulku 7 pro vyjádření podpory jednotlivých prohlížečů. První možnost, prohlížeč je připraven bez dalších konfigurací, je podmíněna dostupností prohlížeče na počítači u Sahi a TestCafe. Jinými slovy je potřeba mít nainstalovaný prohlížeč na svém počítači.

Sahi nedokázalo automaticky detekovat webový prohlížeč Opera a Edge a bylo potřeba upravit konfigurační soubor.

Pro WebDriverIO a Sencha Test jsem využil selenium standalone server a odpovídající webdriver. Potřebné soubory můžete nalézt zde [7]. Pozoroval jsem problémy na Internet Explorer webdriver, kde běh testů byl enormně pomalý. Dále Sencha Test i WebDriverIO má problémy s webdriver pro Firefox 48-52 a po spuštění prohlížeče API nefungují správně a také není prakticky možné běžet automatické testy na zmiňovaných verzích prohlížečů. Poslední funkční verze Firefox je tedy 47.

Pro využití cloud řešení webových prohlížečů jsou vždy nutné další konfigurace, které se liší na jednotlivých nástrojích. V případě Sahi není vůbec podporována tato funkce.

Všechny srovnávané nástroje shodně podporují OS X, Windows i Linux platformu a toto je také důvod, proč podporované platformy dále nerozebíráme.



Obrázek 13: Vyhodnocení podporovaných operačních systémů a prohlížečů

Jak je patrné z tabulky 7, nejlépe zpracovanou oblast podpory prohlížečů má TestCafe. TestCafe dokonce nabízí funkci připojení vzdálených počítačů pomocí tzv. workers. Díky této funkci můžeme využít prohlížeč nainstalovaný na vzdáleném počítači. Také podporuje cloud řešení a TestCafe jednoznačně vyčnívá mezi porovnávanými nástroji.

Další použité nástroje se potýkají s různými problémy, ale na základě pozorování hodnotím WebDriverIO jako druhý nejlépe vyhovující. Nevýhodou WebDriverIO je potřebná konfigurace a časté problémy s verzemi selenium server a webdriver. Ale i přes tyto problémy jsme schopni pokrýt všechny nejužívanější prohlížeče a také cloud řešení.

Sahi má kvalitně zpracované využití lokálních prohlížečů, ale bohužel není možné využít cloud.

Sencha Test je dodán s integrovaným Chrome prohlížečem a další prohlížeče je možné využít pomocí selenium technologií, až na prohlížeč Opera. Ten není možné využít ani pomocí cloud řešení. A právě z těchto důvodů Sencha Test hodnotíme jako nejméně vyhovující. U Sencha Test nástroje bych se rád zmínil o 2 možných typech scénářů testů, které využívají velmi odlišné přístupy k webových prohlížečům. Testování aplikace, která je na jedné stránce a nedochází k přesměrování má plnou podporu prohlížečů, jak zmiňuje část 5.3. Tento typ scénáře se nazývá inBrowser. V našem případě jsme porovnávali druhý typ scénáře zvaný webDriver, pro práci s komplexní aplikací využívající přesměrování a tento typ scénáře se bohužel potýká s celou řadou problémů, které jsou spojeny s již zmiňovanou technologií webdriver.

Obrázek 13 přehledně vyjadřuje výsledky porovnání této kategorie.

8.2 Sada APIs, selektory, práce s iframe a porovnávání snímků

V této kategorii si rozebereme asi nejpodstatnější část porovnávaných nástrojů a to je poskytovaná sada APIs a použité selektory. Sada APIs definuje, jak můžeme manipulovat s jednotlivými elementy a dokáže velmi urychlit a ulehčit tvorbu automatických testů. Jednotlivé rozdíly se pokusíme demonstrovat na konkrétních ukázkách zdrojových kódů.

- ***Selektory***

  Velmi podrobně jsme si popsali selektory v částí 7.1. Zde si uvedeme, jaké selektory využívají konkrétní nástroje, případně vysvětlíme méně obvyklé selektory. Selektory jsou zařazeny do této části, protože se jedná o nedílnou součást API a drtivá většina API využívá selektorů.

  - **WebDriverIO:** WebDriverIO podporuje css a XPath selektory a jejich využití naleznete v ukázce 5.
  - **Sencha Test:** V případě Sencha Testu můžeme použít at-path, XPath, css, component query a composite query pro identifikace elementů či komponent. Component query je speciální druh selektoru pro identifikování Ext JS komponent a praktickou ukázku tohoto selektoru můžete vidět v ukázce 18. Composite query je kombinací component query a css selektoru. Další ukázku užití různých druhů selektorů v Sencha Test můžeme nalézt v ukázce 8.
  - **TestCafe:** TestCafe využívá DOM, css a JQuery selektorů. Praktické užití naleznete v ukázce 11.
  - **Sahi:** Sahi používá pro identifikování elementů vlastní wrapper nad DOM elementy. Praktické ukázky jsou uvedeny na příkladu 14.

- ***Sada APIs***

  Při porovnávání sady APIs můžeme vycházet z kapitoly 7.2, kde jsou uvedeny ukázky zdrojových kódů i s komentáři. Zde si uvedeme další ukázky kódů, které implementují

automatické testy na Ext JS komponentu grid. Uvedené ukázky implementují testovací případ změny emailu v tabulce.

– **WebDriverIO:** WebDriverIO nepodporuje ExtJS komponenty, a proto je potřeba pracovat s grid komponentou jako s tabulkou. Na ukázce 17 si můžeme všimnout, že je potřeba použít složitějších selektorů. Také se nám zde objevuje nové API keys("Enter"), které umožňuje stisk speciálních kláves, jako je například enter, shift apod.

```
it("Zmena emailu v tabulce", function() {
    $("//table[@data-recordindex='4']/tbody/tr/td[2]/div").click();
    $("//table[@data-recordindex='4']/tbody/tr").keys("Enter");
    $("//input[@type='text']").waitForVisible(5000);
    $("//input[@name='email']").setValue("jakub@email.cz");
    expect('jakub@email.cz').toBe($("//input[@name='email']")
      .getValue());
    $("//span[contains(string(), 'Update')]").click();
});
```

Výpis 17: WebDriverIO - ukázka práce s grid komponentou

– **Sencha Test:** Sencha Test podporuje ExtJS komponenty a na ukázce 18 můžeme vidět celou řadu APIs vyvinutých pro grid komponentu. Například API rowAt(4) slouží k výběru řádky v komponentě grid. Můžeme si všimnout i component query selektoru, který je speciálně vyvinut pro práci s ExtJS komponentami. V ukázce si také můžeme všimnout využití type() API nejen pro vložení hodnoty, ale také pro stisk speciální klávesy, jako je enter.

```
    it("013 Zmena emailu v tabulce", function() {
      // component query selektor
      var locTable = "row-editing[title='Row Editing Employees']";
        ST.grid(locTable)
            .rowAt(4)
            .cellWith('dataIndex', 'email')
            .click()
            .type({key: "Enter"});
        ST.textField("textfield[name='email']")
            .setValue("").type("jakub@email.cz");
        ST.button("button[text='Update']")
            .click();
    });
```

Výpis 18: Sencha Test - ukázka práce s grid komponentou

– **TestCafe:** TestCafe nepodporuje ExtJS komponenty a je nutné používat složitější selektory pro identifikaci elementů. V ukázce 19 si můžeme všimnout nově použitého API pro uživatelské akce act.dblclick(selektor). Také pomocí act.type() API můžeme vkládat nejen text, ale i speciální klávesy jako enter.

```
var extGrid = {
        table: function() {
            return "div[id*='row-editing'][data-ref='bodyWrap']";
        },
        tableCell: function (row, column) {
            return ("table[data-recordindex='"+row+"']>tbody>tr>td:nth-
                child("+column+")>div");
        },
        emailField: function () {
            return "input[name='email']";
        }
    };
"@test"["2.Zmena emailu na 5.radku tabulky a jeho overeni"] = {
    "Kontrola zobrazeni komponenty grid": function () {
        act.waitFor( extGrid.table() );
    },
    "Vyber 5. radku tabulky pro editaci": function () {
        act.dblclick( $(extGrid.tableCell(4,2)) );
    },
    "Overeni zobrazeni textoveho pole pro editaci": function () {
        act.waitFor( extGrid.emailField() );
    },
    "Zmena emailu": function () {
        act.type(extGrid.emailField(), email, {
          replace: true
        });
    },
    "Ulozeni zmen": function () {
        act.press("enter");
    }
};
```

Výpis 19: TestCafe - ukázka práce s grid komponentou

– **Sahi:** Přestože Sahi nepodporuje ExtJS komponenty, jsme schopni napsat velmi efektivní kód pro grid komponentu pomocí Sahi APIs. Můžeme využít Sahi APIs pro práci s tabulkou, které bylo použito také v příkladě 20.

```
var $positionInTable = _cell(1, _in(_row(4)));
function testZmenaEmailu(){
   _doubleClick($positionInTable);
   _setValue(_textbox("email"), "jakub@email.cz");
   _click(_span("Update"));
}
```

Výpis 20: Sahi - ukázka práce s grid komponentou

- ***Práce s iframe***

  Ukázky kódů v této části jsou použity z testovacího případu změna jazyka, kompletní seznam testovacích případů nalezte 3.

  – **WebDriverIO:** Pro práci s iframe WebDriverIO poskytuje API *frame()* pro přepnutí do iframe. Pro návrat do hlavního obsahu stránky slouží API *frameParent()*. Také můžeme vidět využití API *selectByValue()*, které ulehčuje výběr hodnoty ze select listu. Popisované APIs můžete vidět na ukázce 21.

```
// selektor pro iframe
var myFrame = $("//iframe[@name='preferences-frame']");
// cekani nez je iframe nacten
myFrame.waitForVisible(3000);
// prepnuti do iframe
browser.frame(myFrame.value);
// vyber hodnoty ze select seznamu
$("#rcmfd_lang").selectByValue("en_US");
// potvrzeni vyberu klikem na tlacitko
$(".button.mainaction").click();
// prepnuti zpet z iframe
browser.frameParent();
```

Výpis 21: WebdriverIO - ukázka práce s iframe

  – **Sencha Test:** Sencha Test nepodporuje práci s iframe. Naštěstí existuje řešení, jak manipulovat s elementy uvnitř iframe. K tomu můžeme využít *execute()* API, které umožňuje vykonat kód přímo na testované stránce či aplikaci. Uvnitř *execute()* již nemůžeme využívat ST API a je potřeba využít čistý JavaScript. Na ukázce 22 můžeme

vidět manipulaci s elementy v iframe pomocí Sencha Test. Sencha Test neposkytuje žádné APIs pro usnadnění práce se select seznamem.

```
// vyber elementu nad iframe
ST.element("//div[@class='iframebox']")
   // cekani na zobrazeni daneho elementu
   .visible()
   // moznost vykonani kodu primo na strance
   .execute(function(){
      // vytvoreni promenne odkazujici na content iframe
      var iframe = document.getElementById("preferences-frame").
          contentWindow;
      // vyber hodnoty ze select seznamu
      iframe.document.getElementById("rcmfd_lang").value = 'en_US';
      // potvrzeni vyberu klikem na tlacitko
      iframe.document.querySelector('.button.mainaction').click();
   });
```

Výpis 22: Sencha Test - ukázka práce s iframe

– **TestCafe:** Na ukázce 23 můžete vidět, jak je řešená manipulace s elementy za iframe pomocí TestCafe. Je potřeba použít inIFrame funkci, kde si definujeme selektor pro iframe. Uvnitř funkce jsme už přepnutí do iframe a můžeme pracovat s elementy běžným způsobem. Dále na této ukázce můžeme vidět, že TestCafe neposkytuje API pro práci se select seznamem a výběr hodnoty musí být realizován pomocí jednotlivých kliknutí. Výhodou takovéhoto řešení je přiblížení se ke skutečnému uživatelskému nastavení hodnoty.

```
"Otevreni select seznamu": inIFrame($("#preferences-frame"), function() {
   act.click("select#rcmfd_lang");
}),
"Vybrani jazyku": inIFrame($("#preferences-frame"), function() {
   act.click("option[value='en_US']");
}),
"Ulozeni zmen": inIFrame($("#preferences-frame"), function() {
   act.click("input[class='button mainaction']");
})
```

Výpis 23: TestCafe - ukázka práce s iframe

– **Sahi:** Na ukázce 24 můžeme vidět sílu Sahi nástroje, který zvládá automatické přepínaní mezi iframe, frame a zpátky do hlavního obsahu stránky. Je potřeba uvést pouze selektory elementů a o zbytek se postará Sahi za nás. Také můžeme vidět použití Sahi API *_setSelected()*, které velmi ulehčuje vybrání hodnoty ze select listu.

```
// nastaveni hodnoty v select elementu, ktere je v iframe
_setSelected(_select("_language"), "en_US");
// klik na tlacitko v iframe
_click(_button("button mainaction"));
```

Výpis 24: Sahi - ukázka práce s iframe

- ***Porovnávaní očekávaného a aktuálního snímku obrazovky***

  Následující ukázky jsou použity z testovacího případu ověření správného zobrazení aplikace.

  – **WebDriverIO:** Pro porovnávání aktuálního a očekávaného snímku jsem použil wdio-visual-regression-service, což je node module. Více informací i postup instalace naleznete zde [25]. Konfigurace je složitější, ale dobře popsána. Výhodou je automatické vytvoření očekávaného snímku při prvním vykonání testu. Při dalším spuštění testu se tento očekávaný snímek porovná s aktuálním. Samozřejmě je zde možnost očekávaný snímek kdykoliv změnit a každý snímek je specifický pro daný prohlížeč. Jak můžeme vidět na ukázce 25, samotné porovnání snímku se provede pomocí API checkViewport(). Použitý parametr misMatchTolerance definuje toleranci odlišnosti obrázku. Zde je použitá nulová tolerance, aby byl při jakékoliv odlišnosti vytvořen i rozdílový snímek, ale v žádném případě to neznamená, že budou končit testy chybou. Běh testů pokračuje i při rozdílnosti snímků a výsledek testu se vyhodnocuje až ve funkci screenComparing(). Tato funkce získává pro nás zajímavou hodnotu z objektu, který je vrácen pomocí checkViewport() API. Pro nás zajímavá hodnota misMatchPercentage říká o kolik procent se liší snímek a podle této hodnoty je určeno jestli test dopadne úspěšně či s chybou. K tomu je použita assertion expect().toBeLessThan().

```
function screenComparing(objects, toleration) {
    objects.forEach( function(object) {
      expect(object.misMatchPercentage).toBeLessThan(toleration);
    });
};
// cekani na zobrazeni elementu
$(rowTableSelector).waitForVisible(10000);
// porovnani s toleranci 0% => pri rozdilnosti vytvoren rozdilovy
    obrazek, ale testy pokracuji
```

```
var comparing = browser.checkViewport({misMatchTolerance: 0});
// jestli je rozdil vetsi nez 5% => negativni vysledek testovaciho
    pripadu
screenComparing(comparing, 0.05);
```

Výpis 25: WebDriverIO - porovnávání snímků

– **Sencha Test:** Už na první pohled z přiložené ukázky 26 můžeme vidět, jak je jednoduché porovnání snímků pomocí Sencha Testu. Je potřeba použít pouze API visible(), které čeká na zobrazení dané komponenty a poté screenshot() API. Použité parametry definují název vytvořeného snímků a toleranci. Nevýhodou je nutnost zadávat toleranci rozdílu snímku pomocí pixelů. V případě screenshot() API, je nutné spustit testy pomocí STC. Výhodou je, že není potřebná žádná další konfigurace. Očekávané obrázky jsou opět vytvořeny při prvním vykonání screenshot(). Pro každý prohlížeč je vytvořený specifický snímek.

```
ST.grid(locTable).rowAt(4).visible();
ST.screenshot('extjs-grid-img', 50);
```

Výpis 26: Sencha Test - porovnávání snímků

– **TestCafe:** Porovnání snímků není momentálně podporováno, zde můžete nalézt požadavek na přidání této funkcionality [23]. Existuje pouze API act.screenshot(), ale i zde najdeme další omezení. Linux platforma act.screenshot() API nepodporuje. Ukázka 27 obsahuje act.waitFor() API, které čeká na načtení porovnávaného elementu. Po načtení elementu je vytvořen snímek a ten je přidán do test reportu, který musí být manuálně porovnán. Tedy v tomto případě jsme schopni vytvořit pouze poloautomatický test, kde porovnávat očekávaný a aktuální snímek musí člověk. Právě z toho důvodu je použita assertion ok(), která vždy končí chybou s vysvětlující zprávou. Tímto způsobem zajistíme selhání v test reportu a zvýrazníme, že snímek nebyl porovnán automaticky.

```
"Kontrola zobrazeni komponenty grid": function () {
    act.waitFor( extGrid.table() );
},
"Vytvoreni snimku aktualniho zobrazeni aplikace": function () {
    act.screenshot();
},
"Porovnani snimku - potreba overit snimky manualne": function () {
    ok(false, "Snimek pridan do test reportu, nutna dalsi kontrola");
}
```

Výpis 27: TestCafe - porovnávání snímků

– **Sahi:** Pro porovnání snímků můžeme použít API _assertSnapShot(). Pro plnou funkčnost porovnávání je potřeba nakonfigurovat GraphicsMagick. Popis konfigurace najdete u popisu API _assertSnapShot() na stránkách Sahi [18]. Nevýhodou je nutnost si vytvořit manuálně snímek pro každý použitý prohlížeč. Na ukázce 28 můžete vidět, že například pomocí _isIE() API zjišťujeme použitý prohlížeč a podle prohlížeče vybíráme snímek pro porovnávání. Navíc je nutné mít okno prohlížeče aktivní a nesmí jej nic překrývat pro vytvoření správného aktuálního snímku. Z toho důvodu je nemožný paralelní běh testů. API _focusWindow() zajišťuje fokus na okno prohlížeče. Dále pro spolehlivé výsledky porovnávání je potřeba si okno prohlížeče maximalizovat, toto jsme zajistili API _windowAction() s parametrem maximize.

```
var $path_screen = "D:\\sahi\\screenshots\\extjs-grid";
if (_isIE()) $path_screen = $path_screen + "-ie.png";
else if (_isChrome()) $path_screen = $path_screen + "-chrome.png";
else if (_isFF()) $path_screen = $path_screen + "-ff.png";
else if (_isOpera()) $path_screen = $path_screen + "-opera.png";
else $path_screen = $path_screen + "-edge.png";
_focusWindow();
_windowAction("maximize");
_assertSnapShot($path_screen, _div("content-panel"), 5, true);
```

Výpis 28: Sahi - porovnávání snímků

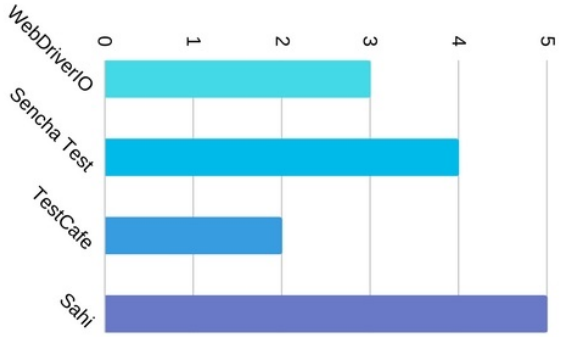
Tuto rozsáhlou oblast nejlépe zpracovává podle mého názoru nástroj Sahi, který disponuje opravdu širokou nabídkou APIs, které automaticky čekají na načtení stránky či AJAX události. Dokonce i detekce iframe je automatická a uživatel neřeší přepínání kontextu stránky. Právě díky zmiňovaným důvodům je psaní testů pomocí Sahi velmi efektivní. Drobnou nevýhodou vidím ve vlastním řešení selektorů, které jsou lehce omezující. Podle mého názoru je při tvorbě selektorů upřednostněna efektivita a jednoduchost před stabilitou. Implementace porovnávání snímků v případě Sahi zaostává za řešením pomocí Sencha Test a WebDriverIO, ale ostatní výhody stále převyšují.

Sencha Test jsem vyhodnotil jako druhý nejvíce vyhovující nástroj z důvodu nejlepšího zpracování porovnávání snímků, kde není potřeba doinstalovávat žádné přídavné moduly ani není potřebná žádná další konfigurace či jiná příprava. Stačí pouze spustit testy pomocí STC. Sencha Test také nabízí široký výběr selektorů, dokonce nabízí selektory speciálně navržené pro ExtJS komponenty a i sada API je rozšířena o celou řadu APIs manipulující s ExtJS komponentami. Největší slabinou Sencha Testu v této oblasti je zpracování iframe problematiky, kde je k tomu potřeba využít čistého javasriptu.

Třetím nejlépe vyhovujícím nástrojem je WebDriverIO. Tento nástroj dostatečně zpracovává jednotlivé oblasti, ale v žádné nevyčnívá. Disponuje širokou nabídkou APIs i selektorů. Také

problematika iframe je zde řešena pomocí speciálních APIs. Konfigurace porovnávání snímků je komplikovanější a i pro získávání výsledků je potřeba dalších úprav.

TestCafe hodnotím jako nejslabší v této oblasti hlavně kvůli chybějícího řešení pro porovnávání snímků a omezené sady API. Výběr selektorů i řešení iframe problematiky je dostatečné. Hodnocení této části můžete nalézt na obrázku 14.



Obrázek 14: Vyhodnocení APIs

8.3 Reporty

Velmi podstatnou částí automatických testů je prezentace dosažených výsledků, a právě proto jsem se rozhodl porovnat i tyto možnosti jednotlivých nástrojů a výsledky můžete vidět na obrázku 15.

- **WebDriverIO:** Pro vytváření reportů je potřeba nainstalovat potřebné nodejs moduly. Postupy jsou velmi dobře popsány na stránkách WebDriverIO [19] v sekci Developer Guide. Po nainstalování potřebných modulů je možné vytvářet reporty ve formátech xml a json. Pro zobrazení reportů je možné využít open-source nástroj Allure, který je možné nainstalovat opět jako node modul. Více o Allure naleznete zde [20].

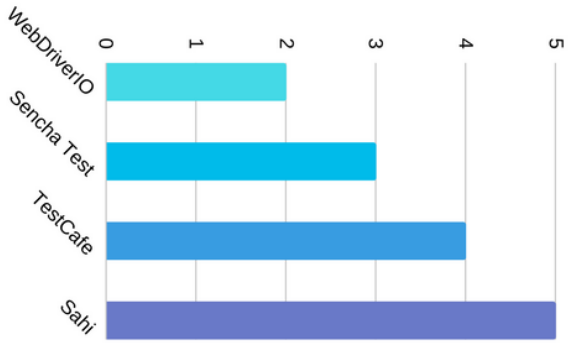
- **Sencha Test:** Pro vytváření reportů je potřeba nakonfigurovat a spustit archiv server pomocí STC a také je potřeba spouštět testy pomocí STC. V případě spuštění testů pomocí grafického prostředí, uživatel uvidí výsledky testů bezprostředně po spuštění testů, ale po spuštění nových testů či zavření okna jsou výsledky ztraceny. Pomocí STC je možno vytvářet reporty v junit, json a xml.

- **TestCafe:** Reporty jsou automaticky ukládány na disku ve formátu json. Pomocí grafického prostředí si je můžeme zobrazit a exportovat do junit, nunit a json.

- **Sahi:** Reporty jsou ukládány do databáze a jejich zobrazení je možné pomocí grafického prostředí. Pomocí grafického prostředí je také možné reporty konvertovat a stáhnout jako html, xml, junit a excel.

Podle mého názoru Sahi nejlépe zpracovává oblast reportů. Reporty jsou automaticky ukládány do databáze a dále lze s nimi velmi dobře pracovat. Jako jediný nástroj umožňuje i filtrování a řazení reportů.

TestCafe automaticky ukládá výsledky na disk a jsou řazeny podle data vytvoření. Sencha Test také řadí reporty podle data vytvoření, ale je nutné použít STC k vytvoření reportu. V případě WebDriverIO je nutné vždy nainstalovat potřebný node module. Allure nástroj sice poskytuje velmi komplexní práci s reporty, ale jedná se o externí nástroj, a proto není zohledněn v porovnání.



Obrázek 15: Vyhodnocení reportů

8.4 Nadstandardní funkce, rozšíření

Pro porovnání jsem vybral 3 nadstandardní funkce či rozšíření, které jsou uvedeny v tabulce 8. První rozšíření je podpora integrace CI systémů a můžeme vidět, že je podporována všemi nástroji. Běh komplexních sad automatických testů může trvat i několik hodin, a právě proto je logický krok přenést automatizaci na CI systém, který se postará o automatické a pravidelné testování.

	WebDriverIO	Sencha Test	TestCafe	Sahi
Průběžná integrace - Continuous integration	✓	✓	✓	✓
Rekordér uživatelských kroků	✗	✓	✓	✓
Podpora ExtJS frameworku	✗	✓	✗	✗

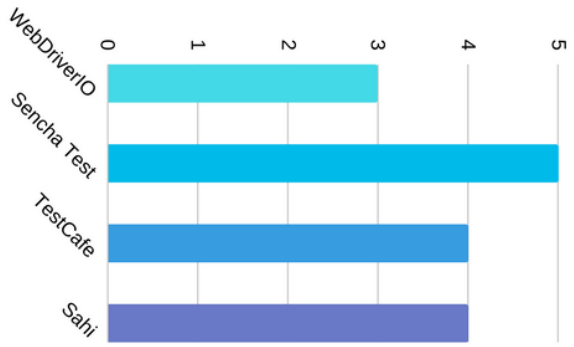
Tabulka 8: Nadstandardní funkce, rozšíření

Rekordér uživatelských kroků je další užitečnou nadstandardní funkcí, ale není možné jej využít k vytváření kvalitních testů. Může pomoci s lokalizací elementů, nebo může nalézt uplatnění pro tvorbu jednoduchých automatických testů netechnickými osobami.

Jako třetí rozšíření jsem si vybral podporu Ext JS frameworku. Jedná se o velmi specifickou oblast, ale chtěl bych na tomto příkladu ukázat, jak moc může ulehčit a urychlit automatizaci nativní podpora daného frameworku, kterou poskytuje Sencha Test. V části 8.2 naleznete ukázky využití APIs podporující Ext JS komponenty, konkrétně ukázka 18 pracuje s Ext JS grid komponentou.

Samozřejmě existuje celá řada jiných i používanějších JavaScript frameworků, ale já si vybral ExtJS z důvodu osobních preferencí a oblíbenosti. Z těch nejrozšířenějších frameworků bych jmenoval AngularJS a testovací nástroj Protractor, který se specializuje na Angular a AngularJS aplikace. Více informací můžete nalézt [21]. Také nesmíme zapomenout na JavaScript framework ReactJS, pomocí něhož je například vytvořen facebook, jedna z nejpoužívanějších webových aplikací. Pro ReactJS je vyvinut testovací nástroj Jest, více informací naleznete zde [22].

V tomto případě míra spokojenosti se přímo odvíjí od tabulky 8 a graf s vynesenou mírou spokojenosti můžete vidět na obrázku 16.

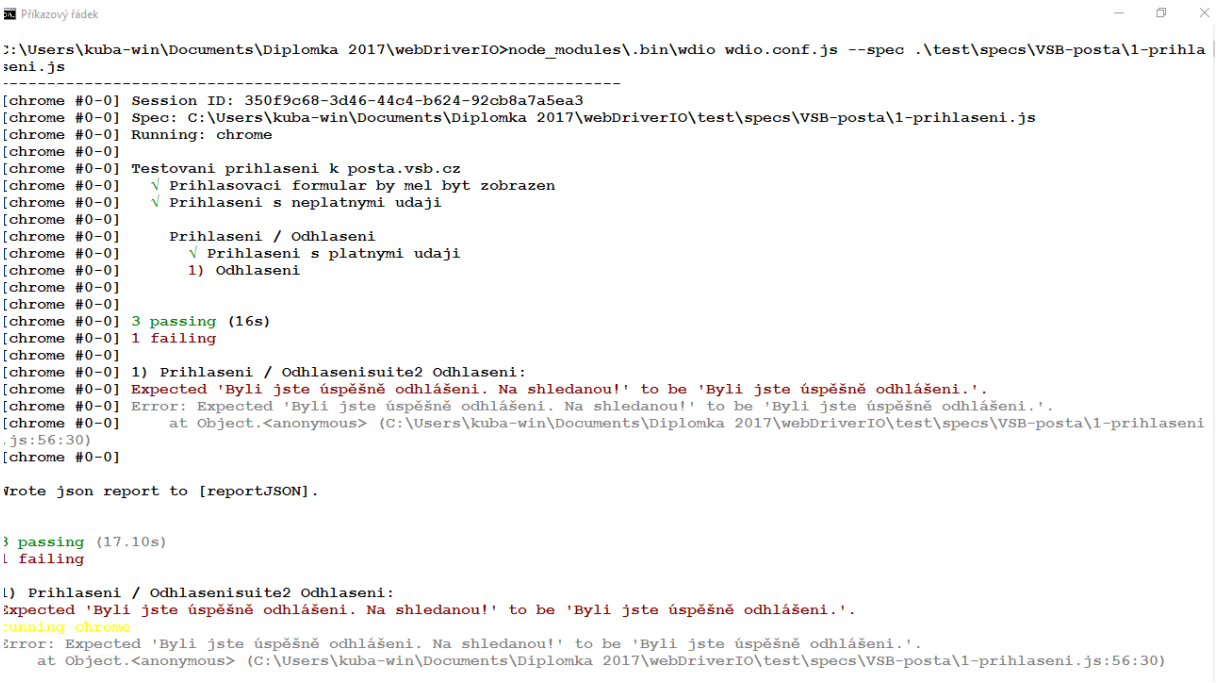


Obrázek 16: Vyhodnocení nadstandardních funkcí a rozšíření

8.5 Uživatelské prostředí, uživatelská přívětivost

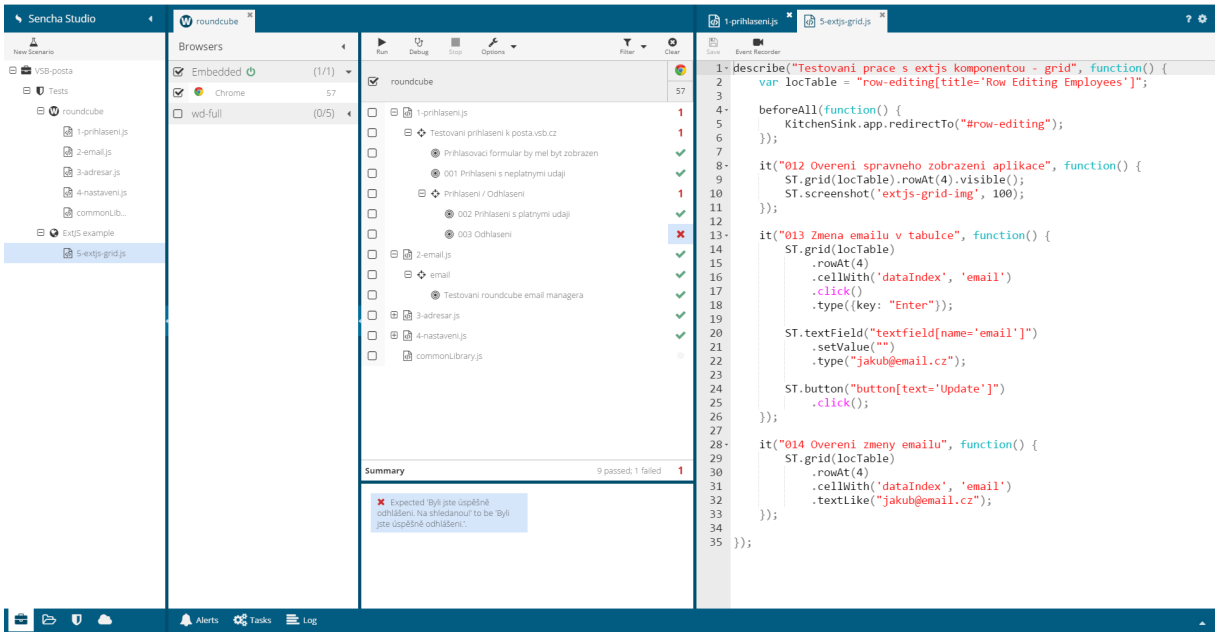
V této kategorii se zaměříme na uživatelskou přívětivost použitých nástrojů. Kvalitní grafické zpracování sice neovlivní funkcionalitu automatických testů, ale může velmi urychlit, usnadnit a také zpříjemnit práci.

- **WebDriverIO:** WebDriverIO neposkytuje grafické prostředí. Pro vývoj testů je potřeba využít IDE třetí strany. Pro konfigurace a spouštění testů je využito příkazové řádky. Výsledky jsou také zobrazeny v příkazové řádce a je možné vybírat z pěti možných druhů výpisů, které jsou následující: dot, spec, teamcity, a concise. Na obrázku 17 můžeme vidět ukázku výpisu typu spec v příkazové řádce.



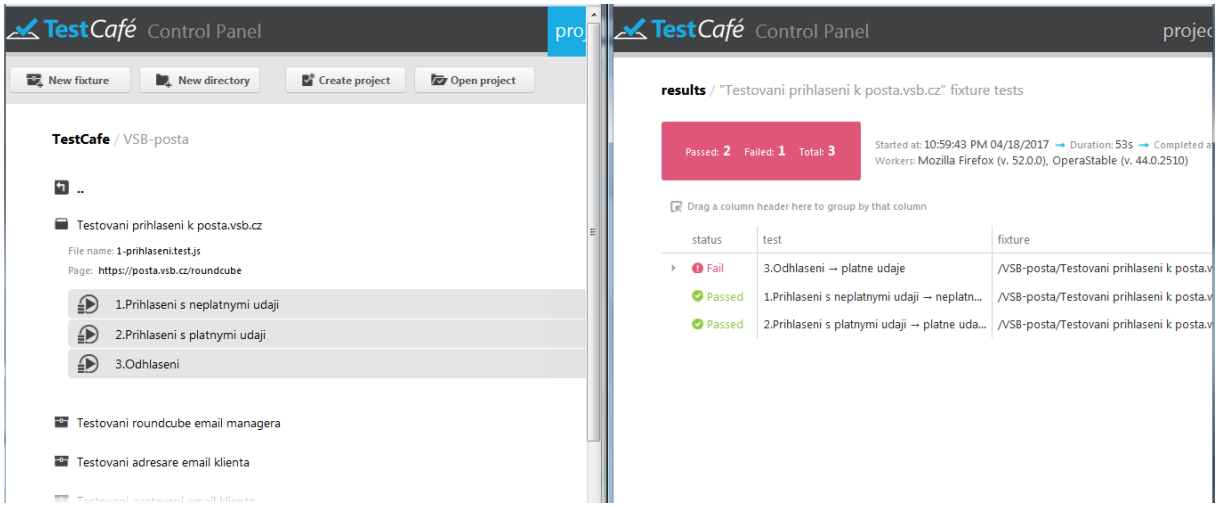
Obrázek 17: WebDriverIO - spec výpis v příkazové řádce

- **Sencha Test:** Sencha Test využívá velmi kvalitně zpracovaného grafického prostředí, které je spuštěno jako desktop aplikace a je nazývaná Sencha Studio. Sencha Studio umožňuje vytvoření projektu, jeho správu, spouštění testů a zobrazování výsledků. Také obsahuje textový editor s našeptávačem a základní kontrolou syntaxe. Na obrázku 18 můžete vidět zleva zobrazenou stromovou strukturu testovacích souborů, dále seznam dostupných webových prohlížečů, stromovou strukturu výsledků i se zobrazenými assertion a nakonec textový editor.



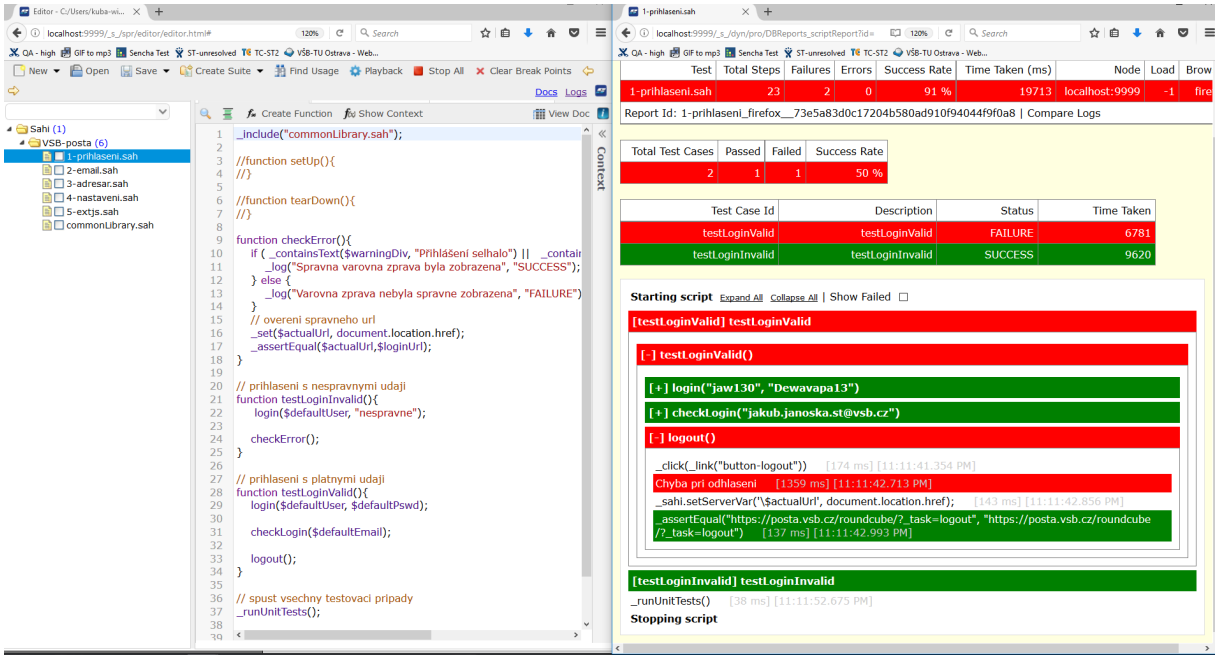
Obrázek 18: Sencha Test - Sencha Studio

- **TestCafe:** TestCafe poskytuje grafické prostředí prostřednictvím vytvořené webové aplikace, která je dostupná z libovolného webového prohlížeče. Pomocí této aplikace můžeme vytvořit projekt, spravovat, spouštět testy a zobrazovat výsledky. Také zde je možnost využít textového editoru. Na obrázku 19 můžete vidět otevřená 2 okna webového prohlížeče. Na levém okně jsou zobrazeny testovací případy, tedy soubory s vytvořenými testy a na pravé straně můžete vidět výsledky automatických testů.



Obrázek 19: TestCafe - grafické prostředí

- **Sahi:** Sahi využívá také webové aplikace pro vytvoření, správu, spouštění testů a zobrazení výsledků. I zde je možné si otevřít testy pomocí textového editoru. Na obrázku 20 můžete vidět 2 otevřená okna, kde na levém okně se nachází seznam testovacích souborů i s otevřeným textovým editorem. Na pravé straně jsou uvedeny výsledky automatických testů.



Obrázek 20: Sahi - grafické prostředí

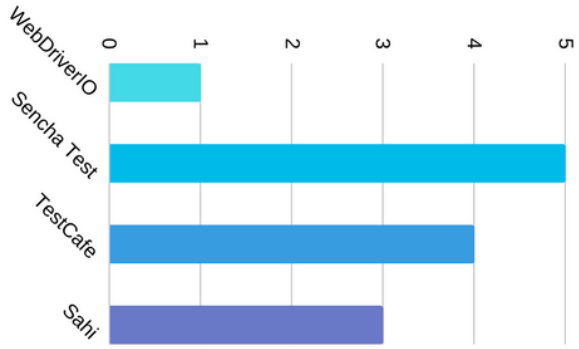
Tato kategorie je velmi subjektivní a každému může vyhovovat jiné grafické zpracování. Na obrázku 21 můžete vidět vyhodnocení podle mého názoru a nyní se pokusím vysvětlit své hodnocení.

Nejlepší grafické zpracování má z mého pohledu Sencha Test. Jako jediný nástroj je Sencha Studio spuštěno jako samostatná aplikace s možností libovolného upravení zobrazených oken. Můžeme zde vidět na jednom okně testované soubory, zdrojové kódy, výsledky i assertions, což velmi urychluje vývoj automatických testů a není nutné se složitě proklikávat k požadovaným výsledkům. Také Sencha Test nejlépe zpracovává textový editor, který podporuje našeptávání a kontrolu syntaxe.

TestCafe využívá webového prohlížeče s moderním designem. Nevýhodou je složitější navigace mezi výsledky a samotnými testy. Textový editor obsahuje jednoduchou kontrolu syntaxe, ale tento editor je spíše vhodný k drobným změnám.

V případě Sencha Testu i TestCafe bylo možné provádět změny v konfiguraci projektu pomocí grafického prostředí. V případě Sahi je nutné dělat tyto změnu přímo v konfiguračním souboru. Grafické prostředí Sahi je zpracované pomocí webové aplikace a podobně jako v případě TestCafe je nutná složitější navigace mezi výsledky a testy.

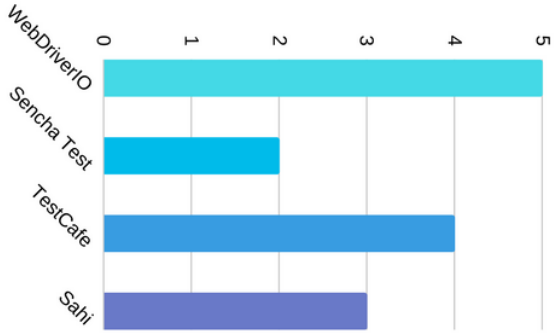
WebDriverIO neposkytuje žádné grafické prostředí a veškerá interakce probíhá prostřednictvím příkazové řádky.



Obrázek 21: Vyhodnocení grafického prostředí

8.6 Cena

- **WebDriverIO:** zdarma
- **Sencha Test:** 495$ za uživatele na 1 rok. Navíc se licence prodávají jako balíček s minimálně 5 licencemi.
- **TestCafe:** 499$ za uživatele na 1 rok. Je také nabízena verze pod MIT licencí, která je zdarma. Tato verze ovšem nenabízí plnou funkcionalitu. Chybí grafický interface, rozdílné APIs a syntaxe a také chybí rekordér uživatelských akcí a řada dalších funkcí.
- **Sahi:** 695$ za uživatele na 1 rok. Také je nabízena verze Sahi OS s omezenou funkcionalitou.



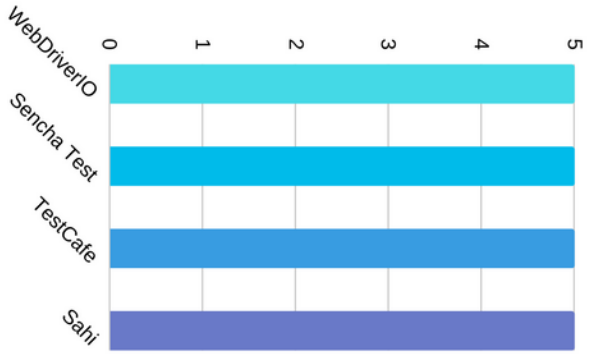
Obrázek 22: Vyhodnocení - Ceny porovnávaných nástrojů

Zde je míra spokojenosti zcela jasná a odvíjí se od ceny licence za uživatele na jeden rok. Navíc TestCafe i Sahi nabízí bezplatnou verzi nástroje, avšak s velmi omezenou funkcionalitou. Obrázek 22 jasně vyjadřuje výsledky v této porovnávané kategorii.

8.7 Dokumentace, technická podpora

Každý z nástrojů nabízí velmi přehledné webové stránky, kde najdeme veškeré potřebné informace. Obsahují například návod pro začátečníky, podrobně zdokumentovaná APIs a podrobné popisy složitějších konfigurací. Podle poskytovaných informací jsem byl schopen velmi rychle nainstalovat daný nástroj a začít s implementací automatických testů.

Technická podpora je také na velmi dobré úrovni. Při položení dotazu na technickou podporu jsem vždy dostal odpověď v rámci několika dní.



Obrázek 23: Vyhodnocení dokumentace, technické podpory a komunity

Z již zmiňovaných důvodů všechny testované nástroje maximálně uspokojily mé potřeby a nevidím zde žádné znatelné rozdíly. Grafické vyjádření míry spokojenosti můžete vidět na obrázku 23.

9 Závěr

Cílem práce bylo porovnat vybrané nástroje pro tvorbu automatických testů. K tomuto jsme navrhli testovací sadu, nad kterou jsme implementovali automatické testy. V průběhu práce se objevuje celá řada ukázek implementací navržených testovacích případů či jejich částí a kompletní implementaci testovací sady naleznete na přiloženém CD/DVD. Součástí přílohy jsou také vytvořená videa, na kterých můžete vidět běh automatických testů v jednotlivých nástrojích.

Nedílnou částí této práce je teoretické zpracování oblasti testování, kde jsem se pokusil také přinést vlastní zkušenost získanou během práce softwarového inženýra.

Srovnání se věnuje celá kapitola 8 a můžeme vidět, že se nejedná o triviální záležitost a prakticky není možné určit jednoznačně nejlepší nástroj pro tvorbu automatických testů. Vždy musíme zohlednit typ projektu a podle aktuálních potřeb si zvolit hlavní kritéria pro výběr vhodného nástroje. Já se pokusil uvést ty nejdůležitější, které ovlivňují výběr vhodného nástroje.

Výsledek našeho porovnání je následující. Sahi nejlépe zpracovává oblast APIs a reportů. Sencha Test může oslnit zpracováním uživatelského prostředí a také poskytuje nejvíce nadstandardních funkcí. V případě nadstandardních funkcí je nutné zmínit, že se zde zohlednila podpora Ext JS framework, což je velmi specifická oblast. Silnou stránkou TestCafe je velmi dobré pokrytí webových prohlížečů a právě v této kategorii TestCafe vyhovuje nejvíce. Ovšem všechny zmiňované nástroje jsou zpoplatněny až na WebDriverIO, který je zdarma.

V dnešní době existují stovky testovacích nástrojů a přínosem této práce může být také navržení způsobu, jak postupovat při výběru vhodného nástroje. Osobní přínos práce vidím v prohloubení znalostí v oblasti testovacích nástrojů, kterou mohu dále uplatnit ve svém zaměstnání.

Vzhledem k tomu, že oblast automatického testování webových aplikací je velmi rozsáhlé téma, nabízí se celá řada možných rozšíření této diplomové práce. My se zde věnovali testování UI pro desktop zařízení a nabízí se rozšíření pro mobilní zařízení. Také vhodným rozšířením by mohlo být automatické testování webových stránek napsaných pomocí JavaScript frameworku, kterých je celá řada, jako například Angular JS, React JS a další.

Literatura

[1] Rex Black, Erik van Veenendaal and Dorothy Graham, *Foundations of Software Testing: ISTQB Certification, 3rd Edition*, Cengage Learning EMEA, 2012. ISBN 978-1-4080-4405-6.

[2] Elfriede Dustin, *Effective Software Testing, 2nd printing*, Addison-Wesley, March 2005. ISBN 0-201-79429-2.

[3] Ron Patton, *Testování softwaru*, Praha: Computer Press, 2002. ISBN 80-7226-636-5

[4] Capers Jones, *Estimating Software Costs*, New York: The McGraw-Hill Companies, 2007. ISBN 9780071483001

[5] *Jasmin Introduction* [online], c2016, [cit. 2017-03-01].
Dostupné z: <https://jasmine.github.io/2.5/introduction.html>.

[6] *Mocha - the fun, simple, flexible JavaScript test framework* [online], c2017, [cit. 2017-03-01].
Dostupné z: <https://mochajs.org/>.

[7] *SeleniumHQ Downloads* [online], c2017, [cit. 2017-04-15].
Dostupné z: <http://www.seleniumhq.org/download/>.

[8] *Ext JS 6.2.1 - Modern Toolkit* [online], c2017, [cit. 2017-03-15].
Dostupné z: <http://docs.sencha.com/extjs/6.2.1/modern/Ext.html>.

[9] *Sencha Test 2.0.0* [online], c2017, [cit. 2017-03-20].
Dostupné z: <http://docs.sencha.com/sencha_test/2.0.0/index.html>.

[10] *ISO/IEC 25010:2011(en)* [online], c2011, [cit. 2017-02-01].
Dostupné z: <https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en>.

[11] *IEEE SA - 829-2008 - IEEE Standard for Software and System Test Documentation* [online], c2008, [cit. 2017-02-01].
Dostupné z: <https://standards.ieee.org/findstds/standard/829-2008.html>.

[12] *Co jsou to webové aplikace a dynamické webové stránky?* [online], c2017, [cit. 2017-02-02].
Dostupné z: <https://helpx.adobe.com/cz/dreamweaver/using/web-applications.html>.

[13] *About Roundcube Webmail* [online], c2017, [cit. 2017-02-20].
Dostupné z: <https://roundcube.net/about/>.

[14] *Chrome DevTools Overview - Google Chrome* [online], c2017, [cit. 2017-03-31].
Dostupné z: <https://developer.chrome.com/devtools/>.

[15] *Firebug* [online], c2017, [cit. 2017-03-31].Dostupné z: <http://getfirebug.com//>.

[16] *Electron · GitHub* [online], c2017, [cit. 2017-02-10].
Dostupné z: <https://github.com/electron>.

[17] *Web App Testing Features: TestCafe | DevExpress* [online], c2017, [cit. 2017-02-16].
Dostupné z: <https://testcafe.devexpress.com/Details/>.

[18] *Automation Testing Tool For Web Applications | Free - Sahi* [online], c2017,
[cit. 2017-02-19]. Dostupné z: <http://sahipro.com>.

[19] *WebDriverIO - Selenium 2.0 javascript bindings for nodejs* [online], c2017, [cit. 2017-04-01].
Dostupné z: <http://webdriver.io>.

[20] *Allure | Test report and framework for writing self-documented tests* [online], c2017, [cit. 2017-04-09]. Dostupné z: <http://allure.qatools.ru>.

[21] *Protractor - end-to-end testing for AngularJS* [online], c2017, [cit. 2017-04-13]. Dostupné z: <http://www.protractortest.org>.

[22] *Jest · Painless JavaScript Testing* [online], c2017, [cit. 2017-04-13]. Dostupné z: <https://facebook.github.io/jest/>.

[23] *Layout testing · Issue #1207 · DevExpress/testcafe · GitHub* [online], c2017, [cit. 2017-04-22]. Dostupné z: <https://github.com/DevExpress/testcafe/issues/1207>.

[24] *Selectors | jQuery API Documentation* [online], c2017, [cit. 2017-02-22].
Dostupné z: <http://api.jquery.com/category/selectors/>.

[25] *GitHub - zinserjan/wdio-visual-regression-service: Visual regression service for WebdriverIO.* [online], c2017, [cit. 2017-03-25].
Dostupné z: <https://github.com/zinserjan/wdio-visual-regression-service>.

[26] *Node.js* [online], c2017, [cit. 2017-04-20]. Dostupné z: <https://nodejs.org/en/>.

[27] *Chrome V8 | Google Developers* [online], c2017, [cit. 2017-02-02].
Dostupné z: <https://developers.google.com/v8/>.

A Seznam příloh

Příloha na CD/DVD.

Adresářová struktura přiloženého CD/DVD:

- Dema
- Sahi
  - ExtJS-grid
  - VSB-posta
- SenchaTest
  - test
    - ExtJS-grid
    - VSB-posta
- TestCafe
  - ExtJS-grid
  - VSB-posta
- WebDriverIO
  - test
    - specs
      - ExtJS-grid
      - VSB-posta