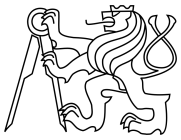
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Automatizace testování výkonnosti prostředí JavaScript
Student:	Bc. Petr Kubín
Vedoucí:	Ing. Jaroslav Kuchař, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Existuje mnoho operačních systémů, prohlížečů a prostředí umožňující běh programů v jazyce JavaScript. Cílem práce je navrhnout webovou aplikaci umožňující zvolit prostředí (např. operační systém nebo webový prohlížeč) a testovanou úlohu (např. webovou stránku či konkrétní JS kód). Aplikace otestuje celkovou výkonnost daného prostředí pro danou úlohu (zahrnující časovou náročnost).

- Seznamte se s aktuálním stavem a proveďte rešerši stávajících řešení.
- Zvolte vhodnou technologii umožňující automatické a opakované sestavení testovaného prostředí v podobě virtuálního stroje.
- Navrhněte a implementujte webovou aplikaci, která bude umožňovat:
  - konfiguraci a sestavení testovaných prostředí,
  - konfiguraci testovacích úloh,
  - spouštění a vyhodnocení testů pro dané prostředí.
- Vytvořte základní sadu testů, která prověří limity zvoleného prostředí.
- Proveďte experimentální testování s různou kombinací prohlížečů a OS.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 20. ledna 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Automatizace testování výkonnosti prostředí JavaScript

Bc. Petr Kubín

Vedoucí práce: Ing. Jaroslav Kuchař, Ph.D.

28. června 2017

Poděkování

Chtěl bych poděkovat Ing. Jaroslavu Kuchařovi, Ph.D. za vedení mé diplomové práce a cenné rady. Děkuji také své rodině za podporu, kterou mi poskytli při zpracování této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 28. června 2017 .....

Odkaz na tuto práci

Kubín, Petr. *Automatizace testování výkonnosti prostředí JavaScript.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Existuje mnoho operačních systémů, prohlížečů a prostředí umožňující běh programů v jazyce JavaScript. Cílem práce je navrhnout webovou aplikaci umožňující zvolit prostředí (např. operační systém a webový prohlížeč) a testovanou úlohu (např. konkrétní JS kód). Aplikace otestuje celkovou výkonnost daného prostředí pro danou úlohu (zahrnující počet operací proveditelných za určitý časový úsek).

Klíčová slova testování JavaScriptu, výkonnost JavaScriptu v prohlížečích, Benchmark testování, automatizace testování

Abstract

There are many operating systems and browsers, enabling environment for running a program in JavaScript. The aim is to design a Web application, allowing you to select the environment (eg. operating system and Web browser) and test task (eg. specific JS code). Application will test the overall performance of the environment for the job (including operation count in specified amount of time).

Keywords JavaScript testing, JavaScript performance in browsers, Benchmark testing, testing automatization

Obsah

Úvod **1**
Motivace 1
Struktura práce 1

1 Rešerše a analýza **3**
1.1 Nastínění problému 3
1.2 Webový prohlížeč 3
1.3 Vylepšení kompilátoru 13
1.4 Různé způsoby zpracování informací 15
1.5 Měření a prezentování výkonnosti 18
1.6 Nástroje pro testování JavaScriptu 20
1.7 Nástroje pro testování komplexního výkonu prohlížeče 23
1.8 Zajištění prostředí pro testování na více OS 26

2 Návrh a realizace **31**
2.1 Analýza a návrh řešení 31
2.2 Hlavní webová aplikace 38

3 Experimenty **49**
3.1 Srovnání výsledků v rámci jednoho stroje 49
3.2 Srovnání výsledků v rámci více strojů 54
3.3 Srovnání prohlížečů v rámci jednoho stroje 55

4 Testování **57**
4.1 Heuristická analýza 57

Závěr **61**

Literatura **63**

A Seznam použitých zkratek **67**

B Obsah přiloženého CD **69**

Seznam obrázků

1.1 Celosvětová statistika prohlížečů od 2009 do 2017[1] 9
1.2 Evropská statistika prohlížečů od 2009 do 2017[1] 10

2.1 Měření přistoupení na stránku google.com a vyhledání výrazu . . . 33
2.2 Test rozdílu času při zatížení výpočetního stroje pro Firefox 33
2.3 Hlavní diagram aplikace 34
2.4 Sekvenční diagram komunikace mezi hlavní aplikací a API 36
2.5 Návrh uživatelského rozhraní pro tvorbu prostředí 40
2.6 Návrh uživatelského rozhraní pro tvorbu testů 42
2.7 Návrh uživatelského rozhraní pro tvorbu testů 46
2.8 Výsledný vzhled výsledku testů 47
2.9 Hlavní strana výsledné aplikace 48

Seznam tabulek

1.1 Tabulka benchmarkovacích testů 25

2.1 Komponenty serveru 35

3.1 Tabulka výsledků pro Windows 10 a Firefox 50
3.2 Tabulka výsledků pro Windows 10 a Google Chrome 51
3.3 Tabulka výsledků pro Windows 10 a Operu 52
3.4 Srovnání prohlížečů v rámci jednoho stroje první test 55
3.5 Srovnání prohlížečů v rámci jednoho stroje druhý test 55
3.6 Srovnání prohlížečů v rámci jednoho stroje třetí test 56
3.7 Srovnání prohlížečů v rámci jednoho stroje čtvrtý test 56

Úvod

Motivace

S rostoucí oblibou jednostránkových moderních aplikací psaných v jazyce JavaScript je potřeba umět zjistit jejich rychlost v různých prohlížečích. Dále by bylo dobré změřit, zda nějak závisí daný algoritmus na operačním systému a jeho konfiguraci. Webový vývojáři by mohli poté testovat své řešení například ve virtuálním stroji přes více prohlížečů.

Jako webového inženýra mě zaujala možnost vytvořit aplikaci, která bude umět testovat JavaScriptové úlohy přes několik operačních systémů a prohlížečů zároveň. Díky nutnosti práce na více OS spolu se všemi hlavními prohlížeči, kterými jsou Mozilla Firefox, Google Chrome, Opera a pro Windowsovské distribuce Microsoft Edge, se jedná o zajímavou práci.

V práci se zmíním o správné konfiguraci virtuálních strojů a konfigurování testovacích úloh. Upozorním čtenáře na určitá specifika a nutnosti, které musí pro správné fungování aplikace dodržet.

Struktura práce

Práce je rozdělena do následujících kapitol:

- Kapitola 1 „**Rešerše a analýza**“, kde uvedu celou problematiku, představím specifika webových prohlížečů, zpracování informací, optimalizaci kompilátorů, měření výkonnosti a možnosti univerzálního přístupu pro testování na více OS a prohlížečích zároveň.

- Kapitola 2 „**Návrh a realizace**“, kde popíši technologie, algoritmy a metody použité pro výslednou aplikaci. Dále zde popíši práci s aplikací a na co si dát pozor při sestavování virtuálních strojů.

- Kapitola 3 „**Experimenty**“, kde vyzkouším mezní nastavení mého řešení na více virtuálních strojích. Provedu experimentování s pamětí, procesory a prohlížeči.
- Kapitola 4 „**Testování**“, které se skládá z Heuristické analýzy a oprav chyb v návrhu.

Kapitola 1

Rešerše a analýza

1.1 Nastínění problému

V dnešní době existuje velké množství webových prohlížečů. Jejich hlavními úkoly je vyhledávání, zobrazování a postupné vytváření zdrojů na světové internetové síti (WWW). Pro správné fungování potřebují znát adresu zdroje. Zdroj je reprezentován pomocí URI/URL, které mohou být webová stránka, obrázek, video, metadata a jiné. Webový prohlížeč ovšem nemusí sloužit pouze pro přístup ke vzdáleným souborům na webu, ale je také schopný fungovat jako lokální prohlížeč v privátní síti, či systému souborů.

Webový prohlížeč neslouží pouze pro přístup na statické stránky a jejich zobrazování uživateli. Musí zpracovávat další informace jako `JavaScript` a styly. Protože ne každý prohlížeč je postavený na stejném základu je možné, že bude vykazovat lepší či horší vlastnosti při zpracování JS kódu a nejeden webový vývojář by si přál vědět rozdíly v rychlosti.

1.2 Webový prohlížeč

1.2.1 Historie webové sítě

Historie internetu[2] má překvapivý začátek v Sovětském svazu. V roce 1957 došlo k vypuštění družice Sputnik I na oběžnou dráhu, což vedlo k masivnímu náskoku ve vědě před USA, přestože Spojené státy americké měli svou družici téměř dokončenou. Tato událost přímo vedla k vytvoření amerického ministerstva obrany (ARPA), které sloužilo pro pokročilý výzkum moderních technologií a jejich realizaci. Jedním z těchto projektů bylo vytvoření celosvětové sítě, která bude umět propojit počítače (internet).

V roce 1960 počítačový vědec Joseph Licklider publikoval článek Man-Computer Symbiosis, který obsahoval základní myšlenku internetové sítě umožňující ukládání a vyhledávání dat. V roce 1962 vytvořil výzkumný tým, který měl tuto myšlenku realizovat. Sám ovšem tento tým opustil ještě před do-

končením projektu. Plán této počítačové sítě (pojmenované ARPANET) byl představen v říjnu 1967 a v prosinci 1969 byly úspěšně propojeny první 4 počítače. Základním problémem této sítě bylo propojení separátních fyzických sítí bez přímého přístupu ke koncovým strojům. Vědci vyřešili tento problém pomocí přepínání paketů, které zahrnuje rozdělení celého balíku dat do menších celků (paketů) a jejich následné posílání kanálem (sítí). Takto rozdělené pakety je možné rychleji zpracovat a v případě chyby nahradit ztracený paket, místo celého datového bloku. Tento princip přetrval dodnes.

Internet ovšem nevyvíjeli pouze v USA, ale také ve zbytku světa. V Anglii používali internet pro rozesílání zpráv a souborů v rámci interní univerzitní sítě. Neboť vývoj neprobíhal centrálně, docházelo k odlišnostem v implementaci. S řešením přišel Robert Kahn, který pracoval v projektu ARPA, kdy stanovil několik základních pravidel pro otevřenější architekturu, která měla nahradit stávající systém ARPANET. Později se připojil Viton Cerf ze Standfordské univerzity a vytvořili nový protokol, který maskuje rozdíly konkrétního síťového protokolu. Celé řešení vyšlo v publikaci v roce 1974 pojmenované `Internet Transmisison Control Program`. Výsledkem bylo zjednodušené propojování počítačů a přesunutí odpovědnosti na hostovský počítač. V roce 1977 byla provedená úspěšná demonstrace propojení tří různých sítí. Do roku 1981 byla odladěná specifikace a o rok později ARPANET předělal své vnější sítě na nový TCP/IP protokol. Tím se zrodil internet tak, jak ho dnes známe.

1.2.2 Vytvoření celosvětové sítě

První systém pro sdílení informací[3] vytvořila univerzita v Minnesotě. Gopher začal fungovat v roce 1990 a umožňoval sdílení odkazů na další soubory, což bylo pro akademickou půdu velmi výhodné pro vyhledávání a získávání informací. Pomohl vysokým školám k centralizaci ukládání a správě dokumentů. V únoru 1993 oznámili Američané, že budou vybírat licenční poplatky za využívání tohoto systému. To přimělo organizace, využívající tento systém, k práci na vlastním řešení.

Evropská rada pro jaderný výzkum (CERN) ve Švýcarsku takovou alternativu nabídla. `Tim Berners-Lee` pracoval na systému pro správu informací, který byl schopný pracovat s hypertextovými odkazy, které umožňují čtenáři přeskočit na jiný dokument. Vytvořil server pro publikování a čtení těchto dokumentů spolu s nástrojem pro čtení, který nazval celosvětovou internetovou sítí (WWW). Tento software poprvé spatřil světlo světa v roce 1991, ale masivního rozšíření se dočkal až déle a nakonec nahradil původní Gopher.

Třináctého dubna 1993 uvolnil CERN zdrojové kódy celosvětové sítě internetu pro volné použití a vývoj bez poplatků. Později v témže roce NCSA vypustila webový prohlížeč pojmenovaný Mosaic, který kombinoval původní webový prohlížeč a Gopher klienta. Z počátku podporoval Mosaic pouze Unixovou platformu a až koncem roku 1993 začal fungovat na počítačích Apple

Macintosh a na operačním systému Microsoft Windows, což mělo za následek rapidní nárůst popularity internetu a webového prohlížeče. Po Mosaicu začal velmi dramaticky růst počet prohlížečů.

1.2.2.1 Války prohlížečů

V roce 1994 vznikl Netscape, který měl dle mého názoru uživatelsky přívětivé rozhraní nabízející nováčkům ve světě internetu určitou nápovědu, co lze a nelze dělat. Jako další prohlížeč se na scéně objevila Opera, která si stále drží své místo i v dnešní době. První vydání bylo v dubnu 1995 a poskytovala podobné rozhraní jako její dva konkurenti. O čtyři měsíce později byl světu představen Internet Explorer, který se stal nedílnou součástí platformy Microsoft Windows. Co je ovšem zajímavější, tak Internet Explorer vznikl odkoupením licence technologie v Mozaicu.

Všechny prohlížeče začali soupeřit přidáváním nových vychytávek a vylepšení, aby na sebe upoutali pozornost vývojářů. Především Internet Explorer a Netscape dávali přednost přidávání nových vylepšení, místo opravování stávající funkčnosti. Dále přidávali své proprietární doplňky, které nebyli kompatibilní s jinými prohlížeči. Softwarový vývojáři tuto situaci řešili vytvářením několika kopií stránek, které byly vždy navržené přesně pro daný prohlížeč, případně se prostě rozhodli, že budou podporovat pouze jeden. Tím vznikla potřeba pro sjednocení struktury a základního chování prohlížečů. Tyto nové regulace jsou zavedeny jako webové standardy.

1.2.3 Webové standardy

1.2.3.1 Založení W3C

V roce 1994 Tim Berners-Lee založil konsorcium (W3C) v Massachusettském technologickém institutu spolu s podporou CERNu, ARPY a Evropské komise. Vize W3C byla standardizace protokolu, technologií a obsahu, který by měl být přístupný co největší množině populace. V průběhu několika dalších let publikovalo W3C specifikace (takzvané doporučení) obsahující HTML 4.01, format PNG obrázků a CSS styly verze 1 a 2.

W3C tyto specifikace nevynucuje, neboť to jsou pouze doporučení. Softwarový vývojáři by měli pouze kontrolovat, zda je produkt přesně podle specifikace, pokud chtějí mít označení W3C kompatibilní. Díky nedirektivnímu doporučení pokračovaly války prohlížečů v devadesátých letech v téměř nezměněné podobě.

1.2.3.2 Nastolení standardu

V roce 1998 byly na špice Internet Explorer 4 a Netscape Navigator 4. Zanedlouho došlo k vypuštění beta verze Internet Exploreru 5, který podporoval proprietární HTML, což znamenalo pro vývojáře webových stránek znát pět

způsobů zápisu JavaScriptového kódu. Proto vývojáři založili skupinu Web Standart Project (WaSP), která měla za cíl změnit doporučení W3C za standard a donutit Microsoft a Netscape podporovat tato doporučení.

Práci na standardech prováděli dobrovolníci ze společností Microsoft, Opera, Mozilla, Apple, Google, IBM, Adobe spolu s hrstkou zaměstnanců W3C. Sedm zaměstnanců vytvořilo skupinu, která identifikovala největší problémy v podpoře CSS stylů. Tyto chyby opravila pouze Opera. Ostatní prohlížeče nereagovali.

1.2.4 Nový směr webových standardů

Po roce 2003 se začalo znovu pracovat na vylepšování webových standardů. Na internetu již nebyli pouze statické stránky, ale spíše aplikace, které jsou v mnohém podobné těm desktopovým. Tyto dynamické aplikace měli problém s přístupností, použitelností a sémantikou.

Tato problematika vedla k zavedení XHTML 1.0, které mělo striktnější pravidla než dosavadní standart HTML 4.01. Později následovalo XHTML 2.0, do kterého bylo zavedeno velké množství vylepšení. Veškeré nové stavění tohoto standardu narazilo na nedostatek ve zpětné kompatibilitě a díky majoritnímu zastoupení Internet Exploreru, který nepodporoval XHTML, došlo k zamítnutí tohoto standardu.

V roce 2004 vývojáři z Mozilly, Opery a později Applu vytvořili specifikaci webových aplikací, která umožňovala následný vývoj bez stálých problémů se zpětnou kompatibilitou. Obsahovala zavedení pravidel pro chování prohlížečů, pravidla pro analýzu DOM a práci s API. Po mnoha diskuzích v březnu 2007 se obnovili práce na aktualizování verze HTML, kde z této specifikace vznikl HTML5.

Toto rozhodnutí představovalo několik výhod. Vývojáři se nemuseli učit nový jazyk, neboť HTML již znají. Došlo k usnadnění práce s videem, JavaScriptem a DOM. HTML5 má také přesně definované parsovací algoritmy, které umožňují prohlížečům vytvořit stejný DOM bez ohledu na validitu.

1.2.5 A co vzhled

Evoluce webových prohlížečů sebou přinesla změny i v kaskádových stylech (CSS). Nejedná se ovšem o tak bouřlivý vývoj jako v případě HTML. Druhá verze CSS byla dokončena kolem roku 2000. V této době zde bylo několik nedokonalostí, které museli vývojáři obcházet pomocí JavaScriptu, například animace, dynamické rozložení prvků a potřebu individuálního písma.

S těmito problémy začalo bojovat CSS3. Došlo ke zvýšení modularity celé struktury kódu, který byl rozdělen do menších, snáze udržitelných kusů. Tento krok usnadnil webovým vývojářům, prohlížečům a návrhářům celkový vývoj a práci s CSS.

1.2.6 Verze prohlížečů

Mezi nejznámější webové prohlížeče patří Internet Explorer (Microsoft Edge), Firefox, Chrome, Safari a Opera. Každý prohlížeč má své verze, kdy s každou další verzí vývojáři slibují lepší kompatibilitu, rychlost, stabilitu a další spoustu věcí, aby byl právě ten jejich prohlížeč nejlepší možný. Internet Explorer a Opera byly jedni z prvních prohlížečů, které spatřili světlo světa. Oba prohlížeče vznikly v roce 1995 a vydrželi do současnosti. Internet Explorer se změněným názvem. Pro více informací o přesném uvedení prohlížeče na trh, jeho vzhledu a webových technologiích doporučuji stránky `http://www.evolutionoftheweb.com/`[3].

1.2.6.1 Opera

Opera[4] byla jeden z prvních prohlížečů. Vznikla již v roce 1995 a udržela se na scéně dodnes. Mezi hlavní výhody tohoto prohlížeče patří vysoká dostupnost na platformách Windows, macOS i Linux. Dále je dostupná i pro mobilní zařízení.

Obsahuje také vestavěné prohlížení záložek, doplňky a správce stahování. Zajímavá je rychlá volba, která umožňuje uživateli přidat neomezený počet stránek ve formě náhledů po otevření nové záložky. Podobnou funkčnost nabízí i Google Chrome, který má 8 těchto náhledů. Toto vylepšení zrychluje práci a navigaci přes vybrané webové stránky.

Prohlížeč Opera byl schopný fungovat i na Linuxových distribucích v 64 bitové verzi, ovšem tuto podporu byla nucena společnost Opera[5] ukončit. Zbyla tak 64 bitová varianta pro Windows a 32 bitová ve starší verzi pro Linux.

Dále obsahuje automatické blokování vyskakovacích oken a filtry na reklamu, díky čemuž má rychle načítání stránek a pro uživatele příjemnější nerušené prostředí. Přes všechna vylepšení disponuje Opera pouze 3.68 % celkového trhu prohlížečů.

1.2.6.2 Internet Explorer

Prohlížeč Internet Explorer vznikl zakoupením licence Mosaicu a následným vlastním vývojem. Neboť celý vývoj zařizuje firma Microsoft, stal se IE součástí Microsoft Windows jako základní prohlížeč, který je k dispozici každému uživateli, který platformu MS Windows používá. První verze byla k dispozici v roce 1995 stejně jako Opera.

Za dobu své existence se tento prohlížeč potýkal s nespočtem problémů, ať se jednalo o kompatibilitu, bezpečnost či dodržování nových standardů W3C. Zde byl značně pomalejší přístup než v případě Opery a v roce 2015 byl tento prohlížeč předělám na Microsoft Edge[6] a v původní verzi zanikl.

V roce 2010 byl prohlížeč Internet Explorer využit při operaci Aurora. Jednalo se o IE6, IE7 a IE8 na nejčastějších tehdejších platformách. Několik

evropských vlád doporučilo přechod na alternativní prohlížeče. Ve verzích 6 až 11 byla možné vzdálené spouštění kódu a uživatelům bylo doporučeno hlídat verze antivirových programů spolu s aktualizacemi Windows. K vyřešení tohoto bezpečnostního problému došlo až v roce 2014, kdy byl Internet Explorer zatlačen do pozadí jinými prohlížeči.

V roce 2009 profitoval IE téměř 60 % celkového zastoupení prohlížečů. O tři roky později byl jeho podíl téměř stejně velký jako Google Chromu (29 %). Ani opravení bezpečnostních nedostatků v roce 2014 nezastavilo prudký pád zastoupení a aktuálně se jeho procento na trhu pohybuje okolo 4.18 %.

1.2.6.3 Mozilla Firefox

Projekt Mozilla[7] vznikl v roce 1998 spolu se zveřejněním zdrojových kódů prohlížeče Netscape. Celý projekt byl koncipován k sjednocení vývojářů po celém internetu a ponechání volné ruce v inovacích a funkčnostech. Vytvořením otevřené komunity vývojářů se projekt Mozilla stal větším než jakákoli společnost. Členové komunity se zapojily různými způsoby, ale jednalo se často o rozličné vize v moderním pojetí prohlížeče.

Po několika letech vývoje byla v roce 2002 představena Mozilla 1.0. Obsahovala mnoho vylepšení prohlížeče, emailového klienta, ale lidé stále používali Internet Explorer. O rok později vznikla nezávislá nezisková organizace starající se o správu a vývoj. Emailový klient byl pojmenován Thunderbird a Firefox jako internetový prohlížeč.

Firefox[8] 1.0 byl spuštěn v roce 2004 a jednalo se o absolutní senzaci. Během prvního roku zaznamenal více než sto milionů stažení. Další verze rychle doplňovali mezery ve funkčnosti a v roce 2013 přišli s platformou pro chytré telefony a dodnes má otevřené zdrojové kódy. Každý vývojář, který chce pomoci k vylepšení této platformy má možnost tento software vylepšit.

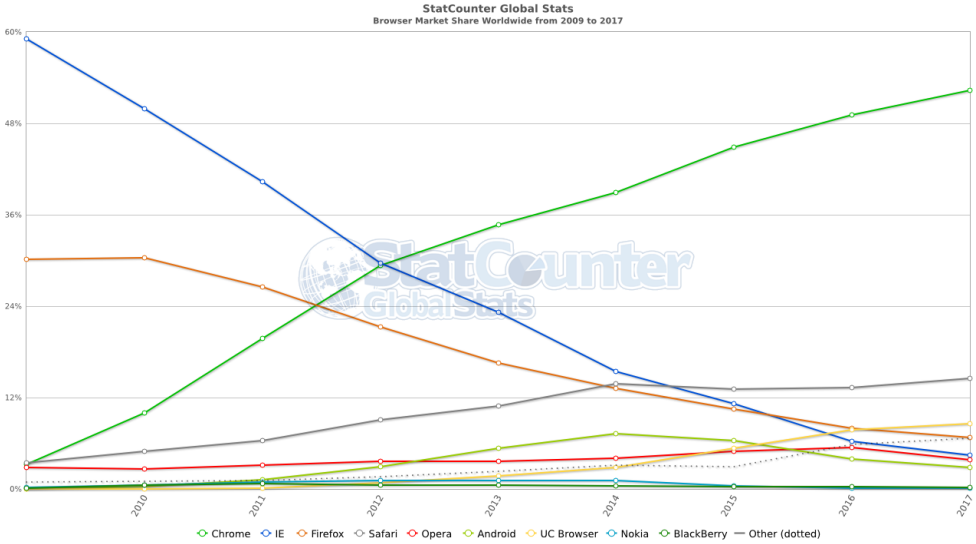
V roce 2009 patřilo Firefoxu 30 % trhu. Od roku 2010 jeho oblíbenost klesá díky prudkému přechodu na Google Chrome. Koncem roku 2011 se propadl popularitou pod Google Chrome. V červnu 2014 se propadl pod Safari. Takto skončil na 6.67 %.

1.2.6.4 Safari

V roce 2003 se na scéně objevil další prohlížeč, který fungoval na platformě Mac OS. Je založený na renderovacím enginu WebKit. Safari verze 1 bylo zavedeno do Mac OS X stejně jako Internet Explorer do Microsoft Windows. Internet Explorer zůstal jako alternativní prohlížeč.

Od Safari 2 již nebyl Internet Explorer součástí Mac OS X. V roce 2007 začalo Safari fungovat i jako prohlížeč pro chytré telefony po představení prvního iPhone.

Dále zkusil Apple zaútočit na podíl Internet Exploreru u Windows, když v témže roce představil verzi Safari 3 s podporou pro Windows XP a Windows



Obrázek 1.1: Celosvětová statistika prohlížečů od 2009 do 2017[1]

Vista. Při přestavení vsadil Apple na rychlejší prvotní načtení stránky, kde Safari předčilo Internet Explorer i Mozillu Firefox.

Jak se později ukázalo rychlost ovšem není všechno, neboť Safari pod Windows obsahovalo nemálo chyb. Přes možnost vzdáleného spuštění kódu až po nepodporování fontů jako například Tahoma a Trebuchet MS. Nakonec Apple upustil od myšlenky ovládnutí prohlížeče u MS Windows a zůstal u své desktopové a mobilní platformy.

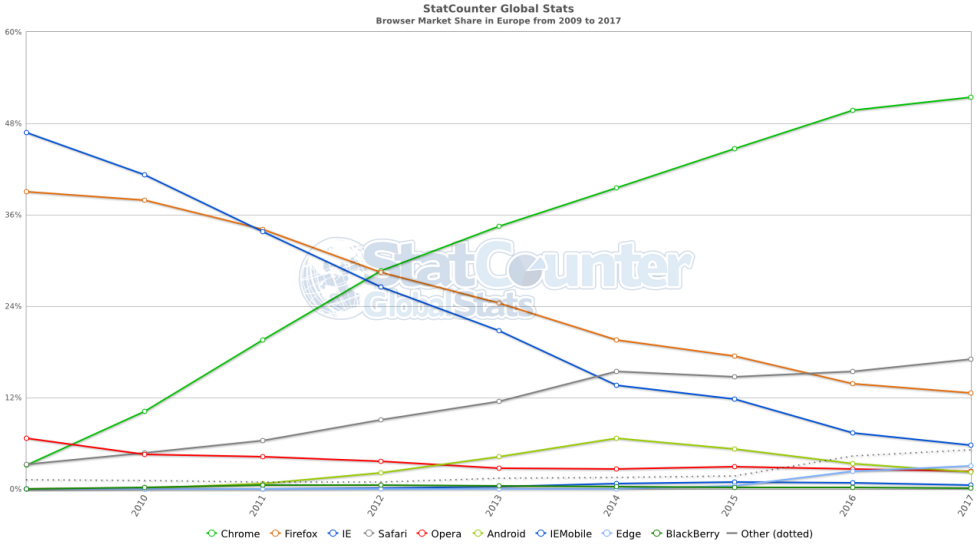
V roce 2009 patřilo Safari pouze 3.4 % trhu, která do roku 2014 vzrostla na 13.77 %. Od roku 2015 se jedná o druhý nejpoužívanější prohlížeč hned po Google Chrome. V první čtvrtině roku 2017 připadá na Safari 14.63 % z celkového zastoupení prohlížečů.

1.2.6.5 Chrome

V roce 2008 vstoupil na trh další prohlížeč od společnosti Google. První verze prohlížeče Google Chrome[9] měli podporu na platformě Microsoft Windows a dále se povedlo vývojářům rozšířit použití na Linux, Mac OS i Android. Protože Google začínal s vývojem velmi pozdě, byla jeho budoucnost nejasná. Google Chrome je založený na opensource projektu Chromium.

Jedna z velkých výhod prohlížeče Google Chrome je bezesporu vysoká bezpečnost spolu s vývojem koncipovaným pro aktuálně používané frameworky a webové technologie. Poslední krok k lepší bezpečnosti bylo odstranění podpory Flash Playeru od verze 53. Pro novou záložku používá chrom sandboxy[1].

[1]Bezpečnostní mechanismus, který slouží k oddělování procesů.



Obrázek 1.2: Evropská statistika prohlížečů od 2009 do 2017[1]

Takto vytvořené záložky sice spořádají více paměti než u konkurence, ale je u nich zaručena vyšší bezpečnost, neboť spolu nesdílí operační paměť ani uživatelské soubory. Dále používá systém blacklistu pro phishingové[2] a malware[3] útoky o které není na internetu nouze. Jedná se o automatické blokování vyskakovacích oken, blokování reklam a další bezpečnostní prvky.

Google Chrome je současně nejvšestrannějším a nejméně platformě závislým prohlížečem s podporovanými systémy:

- Microsoft Windows 7 a novější
- OS X 10.9 a novější
- 64-bit verze Ubuntu, Debian, openSUSE, Fedora a ArchLinux
- Android 4.1 a novější
- iOS 9 a novější

Již v prvním roce svého vývoje oslovil 1 % uživatelů. V polovině roku 2012 přesáhl svou oblíbeností Firefox, ze kterého si vypůjčil svůj základ ve svých začátcích a o půl roku později předstihl také Internet Explorer. Tím se dostal před všechny konkurenční prohlížeče a drží se s velkým náskokem na první příčce v oblíbenosti mezi uživateli. V průběhu posledních dvou měsíců se povedlo prohlížeči Google Chrome získat další 2 % a současně mu patří 54.14

[2]Útok spočívající v odcizení citlivých informací jako přihlašovací údaje, záznamy o kreditních kartách, číslech účtu a podobných

[3]Zastřešující pojem pro počítačové viry, červy, trojské koně a další škodlivý software

% v celkovém zastoupení přes všechny platformy. V případě desktopových platforem se jeho podíl zvýší až na 63.4 %.

1.2.7 Základní stavební kameny

Po krátké rekapitulaci historie a vývoje prohlížečů, které dnes můžeme potkat u klientů při vývoji webových aplikací, je důležité nahlédnout pod uživatelské rozhraní a zjistit základ, na kterém je daný prohlížeč postaven. Pro zobrazování a vykreslování komponent používá prohlížeč vykreslovací nástroj. Ten umí uspořádat dynamické a statické prvky, zobrazit obrázky, audio, video, XML soubory a pracovat s přídavným softwarem. Hlavní prohlížeče, které jsem zmínil výše, používají WebKit, Gecko a Blink.

1.2.7.1 WebKit

Webkit[10] je nástroj pro vykreslování webových stránek v prohlížeči. Je používám v prohlížeči Safari od společnosti Apple. Další uplatnění našel u společnosti Google, která si vzala jádro s otevřeným kódem a svůj projekt pojmenovala Blink.

Komponenty

Webové jádro Webové jádro obsahuje knihovnu pro rozložení, vykreslování a DOM[4] a SVG[5]. Jedná se o kompletní zdrojový kód vyvinutý společností Apple a KDE pod GNU-LGPL[6] licencí. Jádro je napsané v jazyce `C++` a JavaScriptové jádro skriptovacím nástrojem. Aktuální verze již obsahuje multiplatformní knihovny a rozšíření.

WebKit splňuje Acid2 i Acid3 specifikaci bez jakýchkoli nedostatků v oblastech vykreslování, časování či plynulosti.

JavaScriptové jádro JavaScriptové jádro je framework, který poskytuje JavaScriptový nástroj pro WebKit implementace. Vzniklo odvozením z KDE JavaScriptového enginu a PCRE knihovny pro regulární výrazy. Od prvního naklonování prošel JavaScript značnou modernizací pro zlepšení výkonu a podpory různých vylepšení.

V roce 2008 změnili vývojáři kompilaci JavaScriptu přímo do strojového kódu, čímž eliminovali potřebu pro bytecode interpreter.

Dalším vylepšením byla optimalizace JIT[7] compileru, který generuje optimalizovaný strojový kód. O vylepšeních kompilátoru se věnuji více v kapitole Vylepšení kompilátoru.

[4]Document object model
[5]Scalable Vector Graphics
[6]Licence svobodného softwaru, s nutností uvést zdroj na vnořený software
[7]Just in time

1.2.7.2 Gecko

Gecko[11] je webový engine používaný v mnoha projektech společnosti Mozilla, například emailovém klientu Thunderbird, prohlížeči Firefox a dalších open source projektech. Je navržený pro podporu webových standardů, které jsem zmiňoval v kapitole o prohlížečích.

Jako základ pro stavbu tohoto enginu posloužil jazyk C++ a JavaScript. Od roku 2016 je vyvíjen v jazyce Rust[8], což umožnilo funkčnost pro Android, Linux, Windows i macOS.

1.2.7.3 Blink

Blink[12] vznikl naklonováním WebKitu a vývojem jako součást projektu Chromium. O vývoj se zasloužil Google, Opera, Adobe Intel a Samsung. Je používán v prohlížečích Chrome a Opera od verze 15. Největší odlišnosti jsou v implementaci sandboxu a více-procesorového modelu.

Zde jsem ukázal nejčastěji používané enginy pro rozložení webových stránek. Dále je potřeba se zaměřit na enginy, které se starají explicitně o JavaScript a snaží se vylepšit jeho rychlost zpracování.

1.2.8 JavaScriptové enginy v prohlížečích

1.2.8.1 Chrome V8

Chrome V8[13] je JavaScriptový engine vyvinutý jako součást projektu Chromium pro prohlížeč Google Chrome. Dále je používán v mnoha dalších projektech jako MongoDB[9] Node.js, kde běží na straně serveru.

V8 před spuštěním kompiluje javascript přímo do nativního strojového kódu místo překladu do bytecodu či kompilace celého programu. Zkompilovaný kód je poté optimalizován v několika cyklech. O vylepšeních kompilátoru se dočtete více v následující kapitole Vylepšení kompilátoru.

1.2.8.2 SpiderMonkey

Dalším JavaScriptovým enginem je SpiderMonkey[14], který vznikl již v roce 1995 ve společnosti Netscape. Aktuální využití našel v prohlížeči Firefox či GNOME 3. SpiderMonkey obsahuje dvě části:

- TraceMonkey
- JägerMonkey

[8] Vylepšený jazyk C++, který se sám stará o paměť.
[9] Multiplatformní NoSQL databáze využívající JSON formát.

TraceMonkey je první JIT[10] kompilátor napsaný pro jazyk JavaScript. Objevil se v prohlížeči Firefox verze 3.5, kdy pomohl k rychlejšímu zpracování JS kódu. JägerMonkey je kompilátor který pracuje na úrovni bytecodu a optimalizuje v něm jednotlivé operace. Tímto bylo znovu dosaženo značného zrychlení.

1.2.8.3 Charka

Ze strany společnosti Microsoft se můžeme setkat s JavaScriptovým enginem používaným v Internet Exploreru. Tento engine vytvořili v roce 2008 a obsahuje všechny charakteristiky jako předchozí JavaScriptové enginy. Na oficiálním blogu společnosti Microsoft jsou dokonce ukázky benchmarků oproti ostatním prohlížečům a Microsoft Edge spolu s enginem Charka exceluje.

Uvidíme jak se osvědčí v praxi při testování na reálných datech s jednotným systémem.

1.3 Vylepšení kompilátoru

V této kapitole použiji anglické názvy, neboť jsou důležité pro případné dohledání či použití a jejich překlad není všude jednotný. Všechny anglické pojmy podrobně vysvětlím, co přesně znamenají a jak dané vylepšení funguje.

1.3.1 Constant Propagation

Propagace konstanty spočívá v optimalizaci přiřazování konstantní hodnoty do proměnné. V grafu reprezentujícím proměnné a operace s nimi se nám propaguje konstanta a nahradí použití proměnné. Příklad kódu by vypadal takto.

```
/* Simple example of constant propagation */
x = 10;
y = 5 - x / 2;
return y * ( 5 * x + 2 );
```

Po prvním kroku propagace konstant bude kód vypadat takto.

```
/* Code after first optimalization cycle */
x = 10;
y = 5 - 10 / 2;
return y * ( 5 * 10 + 2 );
```

Dále už může kompilátor rovnou vypočítat hodnotu proměnné `y` a tím pádem znát návratovou hodnotu.

[10] Just In Time

```
/* Final result */
x = 10;
y = 0;
return 0;
```

Z původně pracně napsaného kusu kódu nám zbyla pouze návratová hodnota rovna konstantě. Zkušený programátor si tohoto nedostatku všimne a ušetří kompilátoru práci, kdy může celou funkci smazat.

1.3.2 Constant folding

V tomto případě se jedná o proces rozpoznávání a vyhodnocování konstantních výrazů v čase kompilace (sestavení) místo v čase běhu. Podmínkami pro konstantní výrazy jsou například literál typu integer, ale může se jednat i o proměnné, jejichž hodnota je známá v čase kompilace. Jako příklad výrazu s proměnnými může posloužit tento kód.

```
x = 250 * 46 / 42;
```

V takovémto případě kompilátor nebude generovat operaci násobení a dělení, ale vypočítá rovnou výslednou hodnotu, kterou přiřadí do proměnné x. Dále toto urychlení funguje například pro násobení nulou či zřetězení řetězců.

1.3.3 Loop invariant

Pro správné pochopení tohoto pojmu nejdříve vysvětlím, co znamená invariant. Invariant je libovolná množina obsahující všechny dosažitelné stavy[15]. Invariant cyklu poté můžeme ověřit následujícím způsobem.

```
for i = 0 to n
  @@β
```

Proměnná @@β je zde právě zmiňovaný invariant cyklu, u kterého ověřujeme pravdivostní hodnotu. Pokud se při provádění cyklu nic nemění a ověřovací asserce nám dávají stále stejný výsledek, našli jsme invariant cyklu. Tento kus kódu můžeme přesunou mimo cyklus, což nám ušetří jeho stálé provádění, které nemá žádný přínos či vliv na náš program.

Obdobně se dají ověřovat i smyčky while. Příkladem takového ověření by byl následující kód.

```
while P do
  @@β
```

Invariant smyčky nám zajistí přesun zbytečného kódu z vnitřní části cyklu do vnější. Tím nám program poběží rychleji neboť daný kód vykonáme pouze jednou a ne v počtu kroků smyčky.

1.3.4 Dead code elimination

Dead code elimination je způsob práce kompilátoru, který odstraňuje mrtvý kód, jež nemá žádný vliv na výsledek. V případě lidské alternativy jste se mohli setkat s pojmem „Programování mazáním“. Odstranění zbytečného kódu má hned několik pozitivních vlastností.

- Zmenší se celková velikost programu, což umožní rychlejší přenos a načítání.
- Umožní programu vyhnout se provádění irelevantních operací, které pouze zvyšují čas výpočtu.
- Zpřehlední výsledný kód, který neobsahuje slepé uličky při procházení.

Na příkladu si ukážeme různé způsoby mrtvého kódu, které označím komentářem.

```
int foo() { // dead code function
      x = 1;
      y = 2;
}

int main() {
  int a = 2, b = 1;
  int c = a + b;
  return c;
  /* Two lines of dead code under this comment */
  c += 2 * a + b
  return c;
}
```

Hned na začátku můžeme vidět mrtvou funkci, kterou nikde nevoláme. Dále jsme v hlavní funkci `main` zapomněli `return` a vytvořili jsme tak další 2 řádky mrtvého kódu. Jelikož se jedná o velmi jednoduchý program, který toho moc neumí, bude výsledný kód po všech optimalizacích, které jsem zde vysvětlil, vypadat takto.

```
int main() {
  return 3;
}
```

1.4 Různé způsoby zpracování informací

Počítačové technologie za posledních několik let prodělali enormní vzestup. Většina strojů v síti má tím pádem dostatečný výkon pro výpočty a zpracovávání kódu. Server proto nemusí vykonávat veškeré složité výpočty, které

zvládne již každý stolní počítač, aniž by uživatel poznal konzumaci výpočetních prostředků.

1.4.1 Zpracování na serveru

Potřeby serverů velmi vzrostly právě díky vzniku celosvětové internetové sítě. Bylo potřeba mít jeden centrální sklad dat, na který se budou moci připojovat klienti. Klient takto dostane možnost mít přístup ke stejným datům přes více zařízení.

Pro takto vzniklý server je velmi důležitá dostupnost, neboli jak často bude docházet k výpadku serveru. Dostupnost je udávána v procentech a lidé si jí často pletou se spolehlivostí, která je označována MTBF (Mean Time Between Failures). Toto číslo určuje střední dobu mezi poruchami a slouží právě k ohodnocení spolehlivosti serveru. Dostupnost je ovšem ovlivněna i výpadky kvůli technickým kontrolám, zálohování, výměně disku v diskovém poli a podobných událostech. Spolehlivost čistě určuje, za jakou dobu můžeme očekávat selhání výrobku.

Dostupnost je u serveru právě ta nejpodstatnější vlastnost. U ideálního serveru je potřeba dostupnost 24 hodin denně. Při nedávném výpadku serverů na přelomu roku 2016 a 2017 společnosti Google v Evropě, který trval více než 3 hodiny, zaznamenala spousta společností finanční ztráty, neboť používají jejich API, vyhledávání, mapové podklady či statistiky pro zlepšení své pozice na trhu.

Další velmi podstatnou metrikou pro server je škálovatelnost. Nikdo nezačíná s vybavením za desítky miliard, aby 4 roky nemusel nic měnit. Obvyklejší scénář je start s menším serverem a v případě, že server přestane stíhat, je potřeba rozšíti jeho možnosti vertikálním či horizontálním škálováním.

Vertikální škálování je vylepšování konkrétního serveru, například přidáním pamětí RAM, vylepšením procesoru, výměnou disků. Horizontální škálování je přidání dalšího stroje a propojení s původním, aby si případně mohli mezi sebou sdílet data.

Posledním z hlavních kritérií webového serveru je efektivní propustnost. Jedná se o metriku, která určuje celkovou výkonnost daného serveru. Například se může jednat o počet proveditelných operací za vteřinu. Tato metrika se počítá za určitou dobu při plném zatížení serveru pro co nejpřesnější estimaci výsledku.

Po správném navržení serveru nastává otázka, jak se chovat k datům. Podle platné legislativy musíme zabezpečit citlivé údaje jako jsou například jména, hesla, rodná čísla a další. Na serveru k tomu používáme šifrování, které se bohužel časem může stát prolomitelným díky růstu výpočetního výkonu. Dále nemusí být zaměstnanci či správci daného serveru řádně proškoleni a mohou nechtěně prozradit nějaké cenné informace.

Klient tato data musí přijímat a odesílat na server. Útočník může buď zachytit komunikaci, prolomit zabezpečení na serveru, nebo se zmocnit klien-

tova počítače. Problém se zmocněním klientova počítače je v rámci architektury webových služeb neřešitelná záležitost a tak se zaměříme na ty dvě zbylé, neboli jak zamezit toku citlivých dat na server a zpátky a jak vylepšit jejich uložení.

1.4.2 Zpracování dat u klienta v prohlížeči

Pojem počítačový klient je používán pro hardware nebo software využívající prostředky sdílené serverem. Komunikace mezi klientem a serverem závisí na správném posílání požadavků a odpovědí od každé ze stran. Například webový prohlížeč je klient, který se připojuje na server a jako odpověď na požadavek obdrží nejčastěji HTML stránku, JavaScriptový kód, kaskádové styly a případně další data ve formátu XML, JSON a jiných. Client musí tuto odpověď od serveru interpretovat, aby byl výsledek uživatelsky přívětivý.

Clienty řadíme do třech kategorií podle objemu zpracovávaných dat a závislosti na serveru. První kategorie je takzvaný tlustý klient, který zpracovává velké množství dat a nemusí záviset na serveru. Jako příklad tlustého klienta může sloužit klasický počítač, na kterém vyvíjíme software případně modelujeme v Blenderu či CADu a výsledek pouze nahrajeme na server, abychom docílili sdílení s ostatními. Tím veškerá logika a výpočty zůstávají u klienta a server není nijak výrazně zatěžován.

Na druhou stranu existuje takzvaně tenký klient, který je přesným opakem tlustého. Jeho cílem je pouze prezentování dat, které zpracuje a odešle server. Obvykle je tenký klient reprezentován softwarem, který je připravený pracovat v klient/server architektuře. Dále může být tenký klient počítač bez pevného disku, který slouží jako konzole pro správu serveru. Tento typ klienta můžeme najít v nemocnicích, školách, aerolinkách či v telefonních ústřednách.

Poslední kategorií je hybridní klient, který je tvořen sloučením tlustého a tenkého klienta. Spoléhá na úložiště na serveru, ale je schopný vlastního datového zpracování.

Zpracování dat u klienta sebou nese několik výhod. Citlivá data neodesíláme na server, ale stačí odeslat pouze klíč, který v případě bezpečnostního incidentu přegenerujeme. V případě přihlášení lze poslat pouze úspěšné či neúspěšné příhlášení podle kontroly nějakého součtu na klientském počítači.

Na druhou stranu jsou kladeny větší nároky na výpočetní výkon klientů, který je spojen s předpokládaným minimálním výkonem či nainstalovaným softwarem. V případě, že klient nesplní tyto požadavky stává se pro něho služba nedostupná. Při dnešních cenách a výkonech běžných sestav se tento problém téměř nevyskytuje a dále existují statistiky od prodejců, které umožňují lépe odhadnout průměrný hardware a software.

1.5 Měření a prezentování výkonnosti

Když říkáme, že jeden počítač je rychlejší než druhý, co tím myslíme[16]? Běžný uživatel může říci, že je jeden počítač rychlejší než druhý, pokud program doběhne v kratším čase. V případě serveru se můžeme zaměřit na počet operací za určitou jednotku času. Vývojáři se snaží zajistit snižování `času odezvy` což zlepší `čas výpočtu`, neboli času od začátku výpočtu až po jeho dokončení. V případě serveru se dá zvyšovat například `propustnost` pomocí horizontálního či vertikálního škálování. Propustnost určuje míru současně proveditelných operací, nebo maximální objem dat posílaných komunikačním kanálem.

V případě porovnání dvou výpočetních uzlů, kdy říkáme, že je uzel „X rychlejší než uzel Y“, znamená `čas odezvy` uzlu X je menší než čas odezvy uzlu Y. Dále je dobré vědět, jak moc rychlejší či pomalejší daný uzel je. Pro tento výpočet si zavedeme konstantu `n`. Ta nám bude vyjadřovat tvrzení, „X je n krát rychlejší než Y“.

$$\frac{\texttt{čas výpočtu}_Y}{\texttt{čas výpočtu}_X} = \texttt{n}$$

Z tohoto vzorečku můžeme snadno odvodit požadovaný vztah pro výkon, neboť $\texttt{čas výpočtu} = \frac{1}{\texttt{výkon}}$. Celkové odvození pak vypadá takto:

$$\texttt{n} = \frac{\texttt{čas výpočtu}_Y}{\texttt{čas výpočtu}_X} = \frac{\frac{1}{\texttt{výkon}_Y}}{\frac{1}{\texttt{výkon}_X}} = \frac{\texttt{výkon}_X}{\texttt{výkon}_Y}$$

Fráze „`propustnost X je 1.3 krát větší než Y`“ znamená, že X je schopno dokončit 1.3 krát operací více než Y. Vidíme tak nepřímou závislost výkonu na čase výpočtu. Čím nižší čas výpočtu, tím vyšší výkonnost. Ne vždy ale můžeme použít k měření výkonnosti čas. Na zatíženém stroji běží algoritmus pomaleji než na stroji, který má dostatečné množství výpočetních prostředků. Je proto dobré provádět testování jak na zatíženém, tak nezatíženém stroji, což nám dá lepší představu o celkové výkonnosti algoritmu.

1.5.0.1 Měření ceny jedné operace

Pro změření ceny jedné operace je potřeba zjistit, co chceme měřit. Chceme měřit, za jak dlouho jsme schopni provést určitý cyklus, součet či jinou operaci, která je součástí našeho algoritmu. Pro základní testování potřebujeme zjistit čas před začátkem výpočtu a poté čas po dokončení výpočtu. Pseudokód v jazyce JavaScript by vypadal přibližně takto:

```
function benchmark() {
  var startTime = Date.now();
  /* Algorithm goes here */
  return (Date.now - startTime);
}
```

V kódu výše máme jeden zásadní problém. JavaScriptová knihovna, ze které bereme aktuální čas, pracuje s přesností na milisekundy. Pro velmi rychlé operace jako je sčítání, kdy pouze v registru sečteme dvě čísla, nám tato přesnost nebude stačit a nedocílíme statisticky dostačujících výsledků. Kód proto budeme muset poněkud upravit.

```
function benchmark() {
  var startTime = Date.now();
  for ( var i = 0; i < n; ++i ) {
    /* Algorithm goes here */
  }
  return (Date.now() - startTime) / n;
}
```

Po upravení kódu do této podoby jsme docílili několikanásobného opakování měřené operace, což nám prodloužilo celkový čas potřebný pro výpočet. Parametr n nám určuje počet opakování ve for cyklu. Poté získáme čas C potřebný pro jednu operaci $C = \frac{\texttt{naměřený čas} * N}{N}$.

1.5.0.2 Měření času potřebného k výpočtu

V tomto režimu by ovšem pracoval pouze naivní počítač. Po provedení určité operace se její zpracovávání a výsledek uloží do cache a v případě opakování dojde k jejímu načtení z cache místo nového výpočtu. Toto zmenší měřený čas. Pokud bychom si chtěli přesněji vyjádřit tento komplexnější model, musíme vzít v potaz časy jednotlivých operací. R je konstanta pro čtení, L je konstanta pro lokální operaci, W je konstanta pro zápis, $\langle inside \rangle^{n-1}$ znamená n-1 opakování vnitřního výpočtu inside.

```
function benchmark() {
  var startTime = Date.now(); // R + W
  for ( var i = 0; i < n; ++i ) {
    inside
  }
  return (Date.now()-startTime)/n; // R+L+L+W
}
```

$$R + L + inside + \langle inside \rangle^{n-1} + R + 2 * L$$

V tomto případě je potřeba rozepsat vnitřek proměnné `inside`, která nám značí vnitřní operaci, u níž provádíme testování výkonu. Při prvním provedení je čas potřebný pro zpracování roven času reálného výpočtu. Při dalším opakování je zmenšený o rychlost cache.

$$R + W + inside + \langle \frac{inside}{cache\_acceleration} \rangle^{n-1} + R + 2 * L + W$$

Pokud si celý vztah rozepíšeme, dostaneme konstantu, která obsahuje načtení počátečního a výpočet konečného času. Dále vypadá výpočet $C_0 + \sum_{i=1}^{n-1} C_i^{cache}$, kde index `cache` značí zrychlený výpočet. Po zjištění příslušných hardwarových specifik lze výpočet rozvést do finálního výsledku pro konkrétní stroj.

1.6 Nástroje pro testování JavaScriptu

1.6.1 Selenium

Selenium je framework pro testování webových aplikací, který poskytuje možnost vytvoření testu pomocí nahrání scénáře. Díky tomu je možné vynechat učení skriptovacího Selenium IDE. Dále poskytuje možnost pustit tyto testy na většině moderních webových prohlížečů spolu s operačními systémy Windows, Linux či OS X.

Tento způsob testování přímo kopíruje průchod uživatele skrz webovou stránku, nebo testuje určitý scénář. Nejedná se tedy o přesné testování výkonnosti pouze JavaScriptu, ale celkové souhry vykreslovacího enginu, prací s DOM modelem, HTML, CSS i JavaScriptem.

1.6.1.1 Selenium IDE

Selenium IDE je kompletní prostředí pro tvorbu testů. Je implementováno jako doplněk pro Firefox a umožňuje nahrávání, editaci, a krokování testů.

1.6.1.2 Selenium WebDriver

Selenium WebDriver je schopný přijímat požadavky skrz API a dále je přeposlat na prohlížeč. Toto přeposílání se provádí přes prohlížečově závislý ovladač. Firefox má již Selenium WebDriver integrovaný, ovšem pro Chrome je potřeba doinstalovat ChromeDriver.

V první verzi byl potřeba pro běh Selenia server, přes který se posílaly příslušné požadavky. Nyní už je možné pustit Selenium přímo v prohlížeči. Díky tomu se provádí mnohem menší počet požadavků na API a spolu s obohaceným rozhraním se jedná o slušný nástroj pro komplexní testování webových stránek.

1.6.1.3 Selenium Grid

Posledním důvodem, proč jsem zkoumal zrovna Selenium, je možnost pouštět testy ve vzdáleném prohlížeči. To umožňuje rozprostřít zátěž na jednotlivé stroje. Tak máme šanci docílit přesnějších výsledků, které budeme moci statisticky zpracovat v případě shodných konfigurací strojů a zátěží pro dané testování. Dále je možné testování provést na různých operačních systémech i prohlížečích.

Pro komplexnější testování jsem zvolil Capybaru, která podporuje Selenium a umožňuje psát velmi rychle testy naprosto stejným stylem jako RSpec testy pro jazyk Ruby.

1.6.2 Capybara

Capybara[17] je software, který slouží k automatizaci testování, které simuluje různé scénáře a průchody webovou stránkou či aplikací. Je součástí testovacího frameworku Cucumber, který je psaný v jazyce Ruby.

Hlavní charakteristikou Capybary, kterou využívají webový vývojáři, je modularita, snadné psaní testů a snadná údržba. Tyto výhody jsou především potřeba v agilních projektech, kde je velmi vysoká potřeba vše testovat. V případě testování špatnými prostředky se s rostoucím projektem dramaticky zhoršuje i možnost testování, neboť se stává nepřehlednou.

Capybara[18] obsahuje Selenium-webdriver, který umožňuje propojení testování obou frameworků. Takto je možné testovat JavaScript, přistupovat k HTTP zdrojům mimo aplikaci a případně použít prohlížeč v headless[11] režimu.

1.6.2.1 Složení Capybary

Capybara je knihovna/gem, který funguje nad webovým ovladačem. Poskytuje uživatelsky přívětivý DSL[12] a spuštění akce pomocí kliknutí na tlačítko, link a podobně. Po načtení stránky Capybara lokalizuje element v DOM modelu a provede s ním příslušnou akci.

Několik webových ovladačů

rack::test První ovladač slouží jako základní. Je o poznání rychlejší než následující, ale má několik nevýhod. Především neumí přistupovat ke zdrojům HTTP mimo aktuální aplikaci, ve které probíhá testování a dále nepodporuje JavaScript.

selenium-webdriver O tomto ovladači jsem psal již v předchozí kapitole u Selenia. Umí opravit oba dva nedostatky předchozího ovladače, ale pro testování pouze JavaScriptu se nehodí, neboť je zaměřený na uživatelské scénáře.

capybara-webkit Pro pravé testování pouze JavaScriptu je zde poslední ovladač. Používá `QtWebKit` a je o poznání rychlejší než selenium. Při testování také nenačítá celý prohlížeč. Bohužel jde použít pouze v headless[13] režimu.

[11] Režim bez grafického rozhraní pro testování či běh na serveru.
[12] Domain Specific Language
[13] Režim prohlížeče bez uživatelského rozhraní.

Použití v tomto režimu simuluje nepřesně prohlížeč, který má před sebou reálný uživatel a dále s sebou nese další režii v podobě ovladačů, které se musí do prohlížeče instalovat.

1.6.3 JSPerf

Cílem JSPerfu je poskytnout snadný způsob, jak vytvářet a sdílet testové kusy JavaScriptového kódu. Porovnává výkon různých kusů kódu pomoci knihovny BenchmarkJS. Testy může uživatel vytvořit sám, nebo může již vytvořený test vyhledat. Nejčastější použití je pro zjištění, který kód vykoná danou práci rychleji než druhý.

Zde je potřeba se zastavit ohledně optimalizací prováděných prohlížečem a jejich následků. V případě, že napíšeme benchmark[19], který bude mít řádově tisíckrát rychlejší zpracování je možné, že kompilátor provedl optimalizaci kódu a my měříme něco úplně jiného. O možných optimalizacích jsem psal v jedné z předchozích kapitol.

V současné podobě poskytuje JSPerf[20] možnost náhledu výsledků testů v různých verzích prohlížeče pomocí grafu a četnosti spuštění. Daný výsledek ale nic nevypovídá o hardwaru, na kterém tyto testy běžely a nevíme, zda jsou tyto hodnoty v grafu porovnatelné. Dále nepodporuje jakoukoli automatizaci testování přes různé operační systémy a tím pádem mi posloužil pouze jako dobrá inspirace a nikoli předpřipravený projekt.

1.6.4 BenchmarkJS

Jak jsem již zmínil v předchozí kapitole, na knihovně BenchmarkJS[21] stojí nejznámější testovací nástroj BenchmarkJS. Jedná se o velmi chytře napsanou knihovnu, která používá časovače s vysokým rozlišením a veškeré výsledky zpracovává způsobem, který umožňuje jejich další statistické zpracování.

BenchmarkJS[22] obsahuje velmi srozumitelnou dokumentaci, která plně pokrývá jeho funkčnosti. Následující ukázka kódu je kompletní JavaScriptový kód potřebný pro otestování tří testovacích scénářů. Na začátku dojde k inicializaci Benchmarku. Dále se přidají scénáře, v průběhu listenery vypíší průběh testů a na konci dojde k vyhlášení vítěze. Poslední volba je pro asynchronní spuštění testovacích scénářů, které umožní průběžný výpis do konzole, prohlížeče či HTTP požadavkem na libovolnou URL.

```
var suite = new Benchmark.Suite;

// add tests
suite.add('RegExp#test', function() {
  /o/.test('Hello World!');
})
.add('String#indexOf', function() {
  'Hello World!'.indexOf('o') > -1;
```

```
})
.add('String#match', function() {
  !!'Hello World!'.match(/o/);
})
// add listeners
.on('cycle', function(event) {
  console.log(String(event.target));
})
.on('complete', function() {
  console.log('Fastest is ' +
    + this.filter('fastest').map('name'));
})
// run async
.run({ 'async': true });
```

1.7 Nástroje pro testování komplexního výkonu prohlížeče

Pro společnosti, které se zabývají vývojem prohlížečů, je důležité umět ukázat, jak moc dokázali prohlížeč v jejich nejnovější verzi zrychlit oproti konkurenci. Na jednotlivých ukázkách kódu z praxe se toto těžko provádí a proto existují určité nástroje[23], které otestují například rychlost šifrování, vykreslování elementů, propustnost a další metriky, na které si vývojáři potrpí. Mezi nejznámější patří JetStream, Octane a méně známý SunSpider.

1.7.0.1 SunSpider

SunSpider je JavaScriptový benchmark, který testuje pouze JavaScriptové jádro a nikoli práci s DOM modelem či API prohlížeče. Je navržený k porovnávání různých verzí stejného prohlížeče, případně konkurenčních prohlížečů mezi sebou. Testování probíhá přes různé oblasti, které ovlivňují výkon. Od matematických operací přes smyčky až po práci s řetězci.

Dále tento benchmarkovací nástroj nabízí odstínění šumu měření. To znamená zanedbání určitých výsledků a určení pravděpodobnosti a míry chyby, kterou daná metrika vyprodukovala. V případě detailnějšího prozkoumání práce SunSpideru dochází k několikanásobné práci s cache místo reálnějšího scénáře, který uvažuje jeden výpočet, poté anulování paměti a dále znovu stejný výpočet. Při načítání z cache dochází k masivnímu zrychlení a daná metrika se stává téměř bezcennou.

Dalším negativem je repetice přes každý test pouze pětkrát. Pokud naměříme pětkrát určité hodnoty a pro lepší statistický výsledek zanedbáme nejlepší a nejhorší měření, zbudou nám pouze tři hodnoty. Pokud uděláme následně

průměr, tak se nemusí přiblížit správnému rozdělení a špatně tak odhadneme celkový test. Jako poslední nedostatek je konec s udržováním tohoto nástroje.

1.7.0.2 JetStream

JetStream[24] kombinuje řadu referenčních kritérií JavaScriptu, které pokrývají celou řadu pokročilých úloh a programovacích technik. Výsledné skóre vyváží pomocí generického průměru. Před začátkem testu změří zřetelné pracovní zatížení. Díky použití velké škály úloh je odstíněno použití speciálních optimalizačních technik pro ovlivnění výsledku testu.

Samotné testování kontroluje tyto metriky.

- Latence[14] - jak rychle je schopná aplikace nastartovat, dostat se do plného výkonu a ustálit se bez výpadků
- Throughput[15] - měří trvalý špičkový výkon bez doby načtení a záchvěvů
- Geometric Mean[16] - slouží jako celkový výsledek testů

Některá měřítka prokazují kompromisy, kdy v případě přílišné optimalizace konkrétního benchmarkovacího kritéria dojde ke zpomalení v jiné části. JetStream obsahuje benchamrky ze SunSpider a Octane 2. Tato měřítka zahrnují jak klasická měření výkonnosti - jako například strojní a jazykově nezávislá referenční metoda Richards[17], tak používá příklady i z reálného světa - projekt od Mozilly pro renderování PDF (pdfjs), JQuery a Closuje JavaScript knihovny a další.

Celkové testování probíhá v několika cyklech, které se provádí navíc opakovaně. To zaručí dobré statistické výsledky pro následné zpracování, neboť se omezí vliv zatížení stroje na výkon prohlížeče.

1.7.0.3 Octane 2.0

Octane 2.0 se skládá ze 17 testů, což je o čtyři více než jeho předchůdce. Každý test je volený pro pokrytí většiny případů použití, které se vyskytují na reálném webu. S tímto požadavkem byly přidány právě čtyři nové testy do druhé verze benchmarkovacího nástroje.

Protože se velké množství testů shoduje ve všech třech nástrojích, uvedu zde několik příkladů. Ve výčtu uvedu hlavní a vedlejší záměr benchmarku.

Pro detailnější popis a zdrojový kód se podívejte na stránky od Octane případně JetStreamu.

[14] Zpoždění

[15] Propustnost

[16] Geometrický průměr

[17] Strojově a jazykově nezávislý benchmark. Více na `http://www.cl.cam.ac.uk/~mr10/Bench.html`

Tabulka 1.1: Jednotlivé benchmarkovací testy

Název testu	Primární zaměření	Sekundární zaměření
Richards	vlastnosti zátěž/ukládání volání funkce a metody	optimalizace kódu, eliminace redundantního kódu
Deltablue	polymorfismus	OO programování
Regexp	regulární výrazy	-
Crypto	bitové operace	-
Splay	tvorba a destrukce objektů	-
pdf.js	operace s poli	matematické a bitové operace
Mandreel	emulace	-
Code loading	parsování a kompilace JS	-
Box2DWeb	práce s desetinnými čísly	-
zlib	kompilace a spuštění kódu	vlastnosti obsahující double přístup k vlastnostem
Typescript	spuštění komplexních aplikací	-
base64	testování manipulace s řetězci	-
code-first-load	prevence cache prohlížeče	-
crypto-aes	testování práce s celými čísly	-
date-format-tofte	práce s řetězci, používání knihovních funkcí JS	-
gcc-loops	vylepšení GCC a LLVM	-
hash-map	využívání jazykových specifik a objektových konstrukcí	-
quicksort.c	quicksort benchmark	-

JavaScriptové benchmarky se stále vyvíjejí. Je zde tendence dosáhnout co nejlepšího výsledku, který povede ke zvýšené popularitě v případě dostatečné propagace. To vede vývojáře k tendenci spustit benchmark a vylepšovat skóre u jednotlivých položek, které ovšem ne vždy vypovídají o reálném výkonu webového prohlížeče.

Tato neustálá vylepšování dávají benchmarkům konečnou životnost, protože implementace virtuálních strojů začnou nadměrně optimalizovat konkrétní benchmarkovací scénáře a benchmark přestane ukazovat reálnou výkonnost. SunSpider byl první benchmark, který částečně zamezil těmto optimalizacím. Po nějaké době přestaly tyto omezení fungovat a SunSpider zanikl.

V první verzi byl schopný i Octane zmírnit určité dopady optimalizací na jeho mikrobenchmarky. Přes všechny optimalizace vedoucí pouze pro zvýšení špičkového výkonu v testování, byly také provedeny kvalitní úpravy, které dostali JavaScript na úroveň C++ či Javy. Dále došlo k vylepšení garbage

collectoru, které umožnilo prohlížečům vyhnutí neočekávaným pauzám a zásekům.

Při detailním rozboru práce microbenchmarků se ukázalo, že dané úlohy nezaměstnávají V8 analyzátor ani zásobník způsobem jako reálné načítání stránek Wikipedia, Facebook a jiných. Navíc styl Octane jazyka JavaScript neodpovídá idiomům[18] a vzorům, které se používají v moderních knihovnách a frameworcích.

Veškeré optimalizace tedy ve výsledku nezlepšily výkon webových aplikací v reálném světě, ba naopak. V Octanu byla například chyba ve standardu Box2DWeb, kde byl použit operátor < a >=. Zde dosáhl Octanový výkon více než 15% zlepšení. Octane dokonce penalizoval určité techniky, které měli za následek skutečné zrychlení webových aplikací. Mezi tyto techniky patří například lazy parsing[19].

Tyto automatizované benchmarkovací nástroje slouží tedy pouze k pobídkám vývojářů VM, kteří nadměrně optimalizují úzké případy užití a nedostatečně JavaScript v reálných webových aplikacích. Octane umožnil JavaScriptu dosáhnout značného zisku ve výpočetně náročných situacích. Pro mou práci jsou proto tyto benchmarkovací techniky nevhodné, neboť mým cílem je určit rychlost reálných kusů kódu či webové aplikace.

1.8 Zajištění prostředí pro testování na více OS

Pro zajištění optimálního nerušeného prostředí je potřeba omezit veškerou činnost a aplikace na pozadí, které by zpomalovaly příslušný počítač. Pro tuto funkcionalitu jsem se rozhodl využít virtuálního prostředí ve formě stroje, který bude možné konfigurovat, vytvořit, zrušit či případně upravit podle aktuálních požadavků na testování.

Pro virtuální stroje a prostředí pro testování existují dvě hlavní varianty, které zde prozkoumám. První variantou je nástroj, který vytváří kompletní přenosné prostředí pomocí virtuálních strojů - Vagrant. Druhou variantou je Docker, který pracuje s kontainery, jež izolují běh aplikací. Dále z předchozího výzkumu potřebuji toto prostředí umět nakonfigurovat pro běh více prohlížečů naráz. Jedná se o prohlížeče Opera, Firefox, Microsoft Edge, Chromium a Google Chrome.

1.8.1 Vagrant

Vagrant[25] je nástroj, který vytváří a spravuje prostředí virtuálních strojů v jednoduchém pracovním postupu. Pro jednoduchou představu si ve vlatním počítači vytvoříte virtuální stroj, který bude mít určitý operační systém,

[18] Prostředek k vyjádření opakující se konstrukce nebo v jednom nebo více programovacích jazycích.

[19] Rozparsování a eliminace načítaného mrtvého kódu, který je v reálných aplikacích velmi frekventovaně ponechán.

aplikace a nástroje, které potřebujete a místo testování na vašem stroji uskutečníte testování na onom virtuálním. Tím si můžete ověřit, že jste nezapomněli na nějakou závislost a máte přehled o všech programech potřebných pro běh virtuálního stroje. Pro lepší přehled o běhu virtuálních strojů je dobré doinstalovat i nástroj VirtualBox.

Základní sloužkou Vagrantu je Vagrantfile, který je psaný v jazyce Ruby. Jedná se o konfigurační soubor virtuálního stroje, který se vytvoří sám při inicializaci. Ta se provádí pomocí `vagrant init 'ubuntu'`. Příklad takového Vagrantfilu může vypadat takto:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.network "private_network", ip: "192.168.33.10"
  config.vm.network :forwarded_port, guest: 3000, host: 3000

  config.vm.synced_folder "./floder", "/home/vagrant/floder"

  config.vm.provider "virtualbox" do |v|
    v.memory = 1024
    v.cpus = 2
  end

  config.vm.provision :shell, path: 'vagrant/bootstrap.sh'
end
```

Tímto konfiguračním scriptem nastavíme hlavní box jako Ubuntu v 64 bitové verzi. Dále vytvoříme privátní síť, která bude propojovat náš virtuální stroj a hostitelský počítač na IP adrese 192.168.33.10 a aplikaci na portu 3000 přepošle na port 3000 na hostitelský počítač. Dále vytvoříme sdílenou složku, upravíme virtuálnímu stroji paměť a počet jader procesoru, která může využívat. Poslední řádek konfiguračního souboru použije pro provisioning stroje soubor bootstrap.sh. Provisioning slouží k základnímu nastavení virtuálního stroje. V této ukázce se jedná o shell script, který nainstaluje příslušné prohlížeče, nodejs prostředí a další náležitosti.

Spuštění virtuálního stroje je pak dílem okamžiku a postačuje k tomu jednoduchý příkaz `vagrant up`. Automaticky se stáhne image pro příslušný box a provede se konfigurace virtuálního stroje.

1.8.1.1 Výhody Vagrantu

- Vývojové prostředí = produkční prostředí, což vede k minimalizaci chyb
- Jednotné prostředí pro všechny vývojáře na projektu
- Možnost spuštění několika prostředí najednou
- Možnost použití stejných skriptů pro konfiguraci jako na hostingu
- Licence je zdarma i pro komerční využití

1.8.1.2 Závěrem

Vagrant je velmi dobrý nástroj pro tvorbu a běh virtuálních strojů, které je tam možné izolovat pro testování a zajistit na nich pouze software, který potřebujeme.

1.8.2 Docker

Docker je přední světová platforma softwarových kontainerů. Slouží k sjednocení prostředí při vývoji aplikací pro jednotlivé vývojáře. Pracuje na principu oddělených konteinerů, které jsou od sebe izolované. Docker se používá pro Linuxové i Windowsové serverové aplikace.

K vytvoření image dockeru používáme Dockerfile, což je soubor s textovým popisem instrukcí. Během sestavení si docker jednotlivé příkazy cachuje a změna na konci konfiguračního souboru je velmi rychle zakomponována. Při změně na začátku se vše za změnou sestavuje nanovo.

Konfigurační soubor dockeru vypadá o něco složitěji, než ten pro Vagrant.

```
FROM selenium/base

RUN apt-get update \
  && apt-get install -y build-essential libxml2 libxml2-dev \
    libxslt1-dev nodejs npm ruby ruby-dev qt5-default \
    libqt5webkit5-dev xvfb imagemagick firefox \
    google-chrome-stable unzip \
  && apt-get clean && rm -rf /var/lib/apt/lists/* \
                            /tmp/* /var/tmp/* \
  && mkdir /app

RUN wget -O /tmp/chr-driver.zip \
              https://chromedriver/driver_linux64.zip \
   && wget -O /tmp/geckodr.tgz \
              https://github.com/.../geckodriver.tar.gz \
   && cd tmp \
   && unzip chr-driver.zip \
   && tar xzf geckodriver.tgz \
   && cd ~/ \
   && cp /tmp/chromedriver /bin/ \
   && cp /tmp/geckodriver /bin/

ADD xvfb.conf /etc/init

WORKDIR /app

ADD Gemfile /app

RUN gem install bundler && bundle install
ADD . /app
```

```
CMD ./runfirefox.sh
```

Pro sestavení dockeru slouží příkaz `sudo docker build -t dp ./`. Následné spuštění kontaineru se provede příkazem `docker run -rm dp`. V této ukázce je image pro sestavení Selenia a spuštění testů v prohlížeči Firefox. Instrukce `FROM selenium/base` označuje základní image, která bude použita pro tento kontainer. Další instrukce instalují příslušné balíčky, prohlížeče a drivery.

Veškeré kontainery jsou uloženy v cloudu a dostupné komukoli, kdo se do něj podívá. Může si tak váš kontainer naklonovat a upravit podle své vlastní potřeby. Tato myšlenka se mi pro mou práci úplně nelíbí a proto vychází jako vítěz Vagrant.

Kapitola 2

Návrh a realizace

2.1 Analýza a návrh řešení

Součástí této kapitoly je sepsání požadavků na systém, které vyplynuly z vlastních poznatků ze zkoumání teoretických možností, konzultací s vedoucím práce a mých zkušeností s technologiemi.

Výsledný systém bude odpovídat požadavkům a návrhu v této praktické části práce.

2.1.1 Požadavky

Z důvodu komplexního složení aplikace a následné přehlednosti jsem se rozhodl strukturovat požadavky dle částí mé aplikace podle jejich zaměření.

Mezi tyto části patří:

- **Systém pro automatické spouštění testů na více prohlížečích v rámci jednoho OS**
- **Webová aplikace pro tvorbu testů**
- **Webová aplikace pro tvorbu prostředí**
- **Webová aplikace pro spouštění a hodnocení testů přes více OS**

2.1.1.1 Požadavky na automatické spouštění testů

- Systém bude schopen pouštět jednotlivé testové úlohy v rámci jednoho stroje.
- Systém bude schopen poskytovat REST API pro získání výsledků.
- Systém bude mít univerzální implementaci, která nebude závislá na operačním systému.

- Systém bude fungovat na operačních platformách Linux a Windows.
- Systém bude podporovat prohlížeče Google Chrome, Chromium, Firefox, Opera a Microsoft Edge.

2.1.1.2 Část aplikace pro tvorbu testů

- Aplikace bude schopna vytvořit testovací soubor ve formátu JavaScript a HTML.
- Testovací soubory budou odpovídat zápisu testů pro BenchmarkJS.
- Bude možné zadávat HTML i JavaScriptovou část testů.

2.1.1.3 Část aplikace pro tvorbu prostředí

- Aplikace bude schopná vytvořit kostru virtuálního stroje.
- Aplikace bude přijímat konfigurační soubor ve formátu JSON.

2.1.1.4 Aplikace pro spouštění a hodnocení testů

- Skrz rozhraní bude možné zobrazit aktuální virtuální stroje.
- Skrz rozhraní bude možné zobrazit testovací úlohy.
- Skrz rozhraní bude možné zvolit testovací úlohy a virtuální stroje.
- Skrz rozhraní se bude možné dotázat na výsledek testů.
- Skrz rozhraní bude možné spouštět a zastavovat virtuální stroje.

2.1.2 Architektura systému pro automatické spouštění testů

Pro spouštění jednotlivých testů jsem zkoumal dvě varianty. Ze začátku práce jsem zkoumal možnosti Capybary a Selenia, které slouží i pro testování kompletní struktury HTML stránek, práce s DOM modelem a praktickým používáním webových prohlížečů. Druhou variantou bylo testování čistě JavaScriptu podle zadání pomocí platformy BenchmarkJS.

2.1.2.1 První varianta

První variantou bylo Selenium s Capybarou, které bylo schopné obdržet soubor ve formátu RSPEc testů a provést příslušné scénáře. Existuje již několik nástrojů, které jsou schopné tyto scénáře vytvářet a jednalo se tak vcelku o uživatelsky přijatelnou variantu.

Tento způsob testování obsahoval ovšem velké množství nedostatků. Prvním nedostatkem bylo započítávání doby startu prohlížeče do samotného času

testu, bez možnosti tento čas oříznout. Závažnějším nedostatkem byla reakce použité platformy pro jednotkové testování na zátěž stroje. Protože se jedná o čisté měření doby, která je potřebná pro zpracování testovaných úloh, vypadaly výsledky takto:

```
vagrant@fantozzi:~/spec$ firefox=true rspec capybara.rb

Capybara
  uses the web
  will the search work?

Finished in 11.75 seconds (files took 0.70181 seconds to load)
2 examples, 0 failures

vagrant@fantozzi:~/spec$ chrome=true rspec capybara.rb

Capybara
  uses the web
  will the search work?

Finished in 7.32 seconds (files took 0.58506 seconds to load)
2 examples, 0 failures
```

Obrázek 2.1: Měření přistoupení na stránku google.com a vyhledání výrazu

Po prvním testu jsem přidal zátěž na procesor a paměť RAM, což mělo celkové testování zpomalit.

```
vagrant@fantozzi:~/spec$ firefox=true rspec capybara.rb

Capybara
  uses the web
  will the search work?

Finished in 9.59 seconds (files took 0.46817 seconds to load)
2 examples, 0 failures

vagrant@fantozzi:~/spec$ firefox=true rspec capybara.rb

Capybara
  uses the web
  will the search work?

Finished in 37.42 seconds (files took 1.56 seconds to load)
2 examples, 0 failures
```

Obrázek 2.2: Test rozdílu času při zatížení výpočetního stroje pro Firefox

Na obrázku je přesně vidět rozdíl v testovacích časech, kdy na nezatíženém stroji trval test necelých deset vteřin a po přidání syntetického zatížení procesoru a ostatních komponent, vzrostl čas na necelých čtyřicet vteřin.

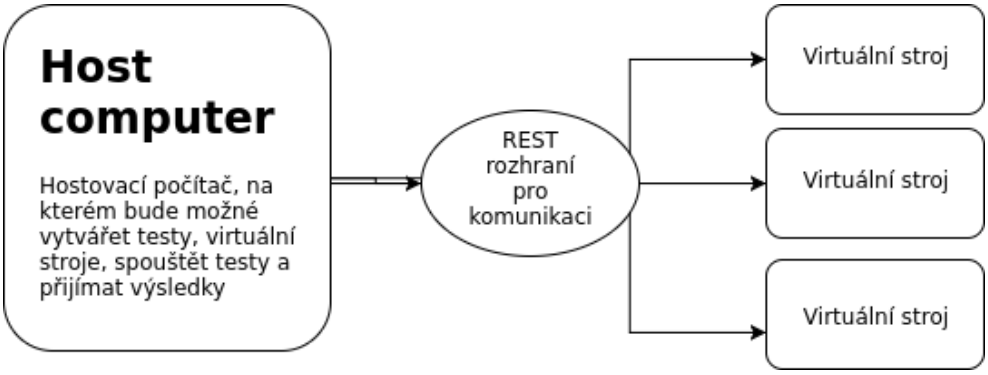
Toto zatížení by bylo možné určitým způsobem normalizovat podle času doby načtení, ale nejedná se o nijak přesný výsledek. Navíc by celé testování potřebovalo přibližně padesát instancí, ze kterých by bylo možné zjistit rozumnější hodnotu po statistickém zpracování.

2.1.2.2 Druhá varianta

Druhou variantou bylo použití testovací platformy BenchmarkJS. Testování přes BenchmarkJS funguje na principu spuštění speciálního scriptu v prohlížeči, který provede dané testování uvnitř prohlížeče.

Již samotná platforma zajišťuje statistické zpracování výsledků, několikanásobné spouštění testů a stanovení procentuální chyby v měření. První část aplikace byla sestavena pro testování na jednom stroji.

Hlavním cílem aplikace, která poběží na virtuálních strojích, byla možnost komunikace i ve virtuálním stroji. K tomu jsem použil NodeJS server[26], který poskytuje REST rozhraní. Pro lepší představu o architektuře se podívejte na diagram.



Obrázek 2.3: Hlavní diagram aplikace

Hostovací počítač rozeberu v dalších kapitolách. V této se jedná o návrh vystaveného REST rozhraní a aplikačního serveru uvnitř virtuálního stroje.

2.1.2.3 Jak testovat pomocí BenchmarkJS

Pro testování pomocí BenchmarkJS[22] je potřeba přidat do stránky HTML a JavaScriptový kód. HTML kód slouží jako prerekvizita pro testování, může se jednat například o několik DOM elementů, na kterých budeme chtít zkoušet různé způsoby jejich výběru. Dále mám připravené pole results, do kterého ukládám jednotlivé výsledky testů. Pro zpracování testů je možnost zapnutí asynchronního režimu, který umožní vykreslování částí výsledků postupně v rámci běhu testu.

V mé práci jsem ještě obalil celkovou strukturu testu výpisem do HTML a do konzole pro možnost zobrazení výsledků v průběhu a kontrolu, zda se

Tabulka 2.1: Komponenty serveru

Název komponenty	Účel komponenty
ExpressJS	Možnost práce s adresářovou strukturou jako REST zdrojem
Axios	Posílání HTTP požadavku
Open	Otevření prohlížeče na určité URL
bodyParser	Rozparsování výsledku v aplikaci
ejs	Renderovací engine
fs	Pro práci se soubory a složkami
child_process	Pro správu procesů
exec	Spouštění příkazů
rimraf	Práce se složkami na fylesystému
ps	Pro práci s procesy

testování někde nezaseklo. Po dokončení testování se nalezne nejrychlejší a nejpomalejší případ a výsledky se odešlou na routu `/results`.

2.1.2.4 NodeJS server

Pro možnost vystavení REST API jsem použil server napsaný v NodeJS. Potřeboval jsem server, který bude co nejméně zatěžovat příslušný stroj a zároveň bude schopný pracovat univerzálně na Windows i Linuxu. Komponenty serveru:

ExpressJS ExpressJS je framework, který umožňuje přehlednou organizaci webové aplikace do třívrstvé architektury na straně serveru. Pro renderování view jsem použil EJS, který má stejnou notaci jako ERB pro Ruby, se kterým mám zkušennosti. ExpressJS se dá použít i pro práci s databází, to jsem ale v mé diplomové práci nebotřeboval, neboť se na výsledky dotazuji, než dojde k vypnutí stroje.

Axios Axios jsem použil pro komunikaci mezi testovacím prostředím BenchmarkJS a aplikací na virtuálním stroji. Jedná se o jednoduchý mechanismus, jak poslat GET/POST požadavek na příslušné API rozhraní. V testovacím kódu používám Axios na vrácení výsledků, kde result reprezentuje výsledky uložené z testování.

```
axios.post('/results', { results: result });
```

Open Open je velmi šikovná zkratka za přímé volání binárky v systému. Umožňuje otevřít URL pomocí prvního parametru, druhý parametr je poté prohlížeč. Protože v mé práci pracuji s několika prohlížeči najednou, je velmi jednoduché ve všech pustit test.

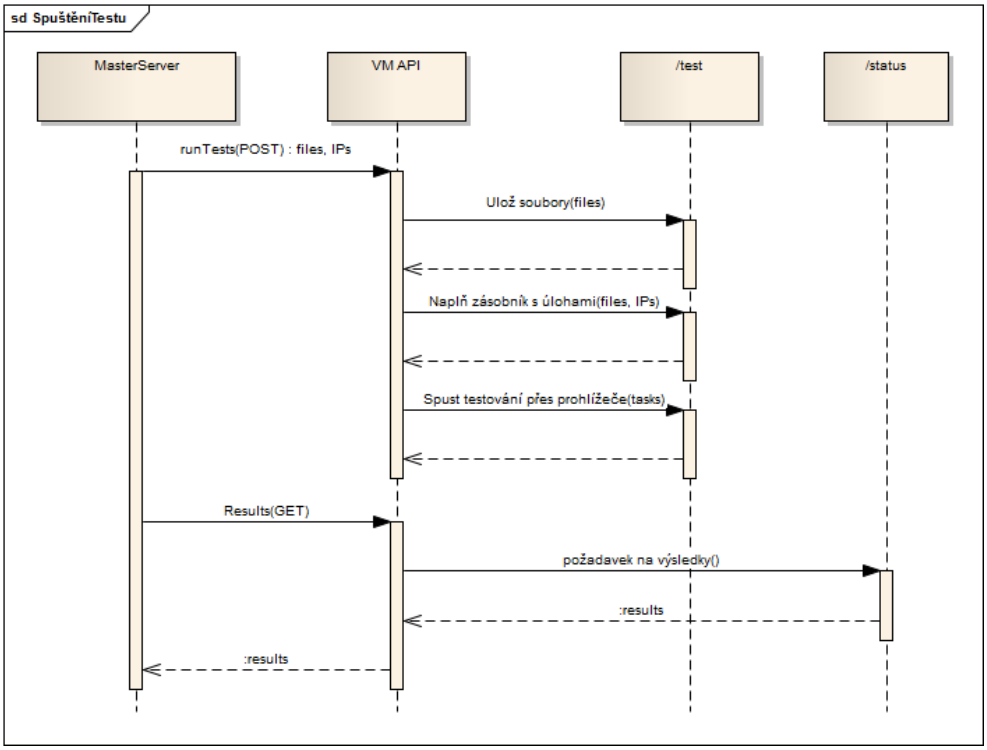
Rimraf a PS Jak jsem již zmínil na začátku práce, testování závisí na použitém stroji a na jeho zátěži. Proto jsem nemohl spustit všechny testy ve všech prohlížečích naráz, ale musel jsem vymyslet dávkové zpracování, které zajistí lepší rozložení zátěže na stroji a přesnější výsledky.

Pro ukončování prohlížečů jsem použil `exec` a `ps`. Zkoušel jsem použít i systémový `process`, ten ale nebyl úplně bezchybný a choval se jinak pod Windows a Linuxem. Testovací úlohy mám uložené v příslušných souborech, které otevírám v prohlížeči dávkově.

Pro zajištění spouštění správných úloh vždy smažu a znovu vytvořím složku, do které úlohy na začátku testování ukládám. K tomuto účelu jsem použil právě `rimraf`.

2.1.2.5 REST API

V této části bylo potřeba zajistit možnost spuštění testů a možnost dotazu na výsledky. Celé testování musí probíhat nerušeně bez zbytečného dotazování na API a snižování výpočetních prostředků pro prohlížeč a stroj. Na následujícím diagramu je nakreslená komunikace mezi hlavní aplikací a API.



Obrázek 2.4: Sekvenční diagram komunikace mezi hlavní aplikací a API

Spuštění testů Spouštění testů probíhá dávkově. Pro všechny testové úlohy a všechny testové prohlížeče uložím kombinace na zásobník, ze kterého odeberu vždy jednu instanci, kterou otestuji a po zapsání výsledků pokračuji na další, než je zásobník prázdný. Tento způsob mi umožňuje testovat jak na jednom stroji, tak i přes webové rozhraní, kdy si pošlu soubory v POST požadavku, uložím je a naplním zásobník.

Zde jsem narazil na jistou nedokonalost v návrhu operačních systémů a prohlížečů obecně. Jedná se o nejednotný název běžícího procesu a binárky, která spouští daný prohlížeč. Takto jsem musel pracovat s dvěma poli názvů, které jsem synchronizoval pomocí globálního indexu, abych byl schopen příslušný process prohlížeče ukončit po doběhnutí testu[27].

Pro spuštění testu jsem zvolil metodu POST a url `/tests/nazev_testu`. Jako první se provede uložení souborů do připravené složky. Dále následuje inicializace zásobníku a samotné spouštění testů. To probíhá otevřením prohlížeče na url pomocí `open('url' + testovací_soubor, prohlížeč)`. Po doběhnutí testu sám test pošle odpověď na url `/results`. Zde zkontroluji[28], ze kterého prohlížeče odpověď přišla a zabiji jeho process. Protože ne každý prohlížeč má právě jeden process, je potřeba vybrat jeho hlavní. Jedná se o první číslo procesu ve výpisu. Po ukončení prohlížeče počkám 100ms, než zapínám další test, kvůli spojené režii.

Po dokončení posledního testu vyčistím složku od testovaných souborů a připravím tak na nové testování. Kvůli asynchronnímu zpracování čistím složku až na konci testů místo na začátku, neboť nevím, jak velké soubory napadne uživatele nahrát a kvůli potřebnému timeoutu by mohlo docházet k nepředvídatelnému chování.

Poslední část tvoří GET `/status`, na kterém se vypíší výsledky. Tato URL slouží pro dotazování v průběhu i po dokončení testů na výsledky.

Výběr prohlížečů Součástí požadavků na aplikaci byla možnost výběru několika prohlížečů. V aplikaci jsem připravil tři soubory pro Linux-32bit, Linux-64bit a Windows 10. Každý soubor obsahuje připravenou konfiguraci pro dané prostředí.

V serveru pro Windows jsem musel upravit i spouštění prohlížečů. Implementace byla univerzální pro všechny dostupné prohlížeče kromě Microsoft Edge. Ten vyžadoval naprosto odlišný mechanismus zápisu pro spuštění. V příkladu je zde vidět univerzální spuštění pomocí open a proprietární spuštění MS Edge.

```
if (browsers[current.browser] != "microsoft-edge:")
open('url' + current.file, browsers[current.browser]);
else {
exec('start microsoft-edge:url' + current.file);
```

Zde je vidět odlišný zápis pro spouštění binárky, kdy je potřeba zapsat `microsoft-edge:` a poté url a parametry.

Spouštění serveru na virtuálním stroji Při vytváření virtuálních strojů pro testování je potřeba ručně nastavit příslušné prohlížeče a zajistit automatické spouštění tohoto serveru.

Pro Microsoft Windows je možné nastavit spuštění po přihlášení uživatele v plánovači úloh. Pro Linuxové distribuce lze server spustit po startu GNOME, který má také svůj plánovač aplikací po startu. Takto se spustí server a je možné zahájit komunikaci přes API.

Shrnutí automatického spouštění testů Pro potřeby spouštění testů na jednotlivých strojích jsem zvolil NodeJS server, který poskytuje REST API pro komunikaci s hlavní webovou aplikací. Na začátku je potřeba specifikovat prohlížeče, které budeme v daném prostředí testovat. To se provádí zápisem do pole `browsers`. Poté musíme upravit názvy procesů pro ukončení těchto prohlížečů, neboť si vývojáři nedali záležet s jednotnými názvy.

Připravenému serveru nastavíme automatické spouštění po startu operačního systému, nebo můžeme pustit server ručně a sledovat průběh testování přímo v konzoli. Tento způsob implementace zaručuje možnost testování i v rámci jednoho stroje, což nám zaručuje lepší výsledky, neboť neplýtváme výpočetní prostředky na běh virtuálních prostředí.

2.2 Hlavní webová aplikace

Hlavní webovou aplikaci jsem vyvíjel ve stejné technologii jako aplikaci na jednotlivé stroje. Použil jsem NodeJS a EJS framework pro šablonování. Základní návrhy uživatelského rozhraní jsem nakreslil v nástroji Balsamiq Mockups 3. V každé kapitole popíši základní rozhodnutí, funkčnost, návrh grafického rozhraní a shrnutí výsledku.

2.2.1 Část aplikace pro tvorbu prostředí

Pro možnost testování na více OS je potřeba umět vytvořit virtuální stroj. Pro vytvoření virtuálního stroje jsem použil balíček `node-vagrant`, který umí vytvořit soubor na základě JSON konfiguračního scriptu. Dále umí pracovat s provision systémem, který je důležitý pro instalaci nadstavbových komponent uvnitř virtuálního stroje.

Pro operační systém Microsoft Windows jsem připravil prostředí předem, neboť zde jsou velmi omezené možnosti základního nastavení a provisioningu. Pro možnost spuštění takto předem vytvořeného stroje přímo ve vagrantu, popřípadě VirtualBoxu, jsem doplnil aplikaci o výpis virtuálních strojů. Zde se vždy zobrazí název a stav virtuálního stroje, zda běží či ne. Poté je jej možné po zadání zapnout či vypnout.

Po zapnutí stroje je důležité chvíli počkat před samotným začátkem testování, neboť stroj musí stihnout naběhnout. Pro dodatečné nastavení virtu-

álního stroje je potřeba přihlášení přes konzoli a SSH, nebo přes VirtualBox GUI.

2.2.1.1 Co a jak nastavit

Pro sestavení prostředí je jako první potřeba vybrat správnou základní image z databáze vagrant boxů. Poté je potřeba upravit konfigurační soubor ve formátu JSON pro vygenerování souboru Vagrantfile.

Pro instalaci komponent a programů uvnitř virtuálního stroje je potřeba použít provisioning. Ten umožňuje například instalaci prohlížečů. Jediná nevýhoda tohoto řešení je instalace, u které je potřeba akce z grafického uživatelského rozhraní samotného stroje. V tomto případě se nedá do konfiguračního ani provision scriptu zanést část instalace a je potřeba ji dokončit ručně po nabootování do virtuálního stroje.

Podmínkou pro úspěšné spuštění prohlížečů je instalace libovolného grafického rozhraní. Pro moje testování jsem zvolil GNOME, neboť se jedná o vcelku systémově nenáročné prostředí a hlavně ho několik let aktivně používám. Možnou vyjímkou u konfigurace je testování prohlížečů v headless režimu. V tomto případě není grafické rozhraní potřeba, ovšem nemyslím si, že je dobré dělat benchmarky v jiném režimu, než ve kterém reálně prohlížeče používáme.

Windows Při sestavování Windows image jsem použil již sestavenou image Microsoft/
EdgeOnWindows10[29] jako základ pro mé rozšíření. Dále jsem chtěl nastavit síťové rozhraní[30] mezi hostovacím strojem a virtuálními Windows. Na tomto kroku se celé spuštění zaseklo a nepomohlo ani manuální přidání username a hesla pro ssh. Musel jsem proto vyřešit konflikt s nastavením ručně.

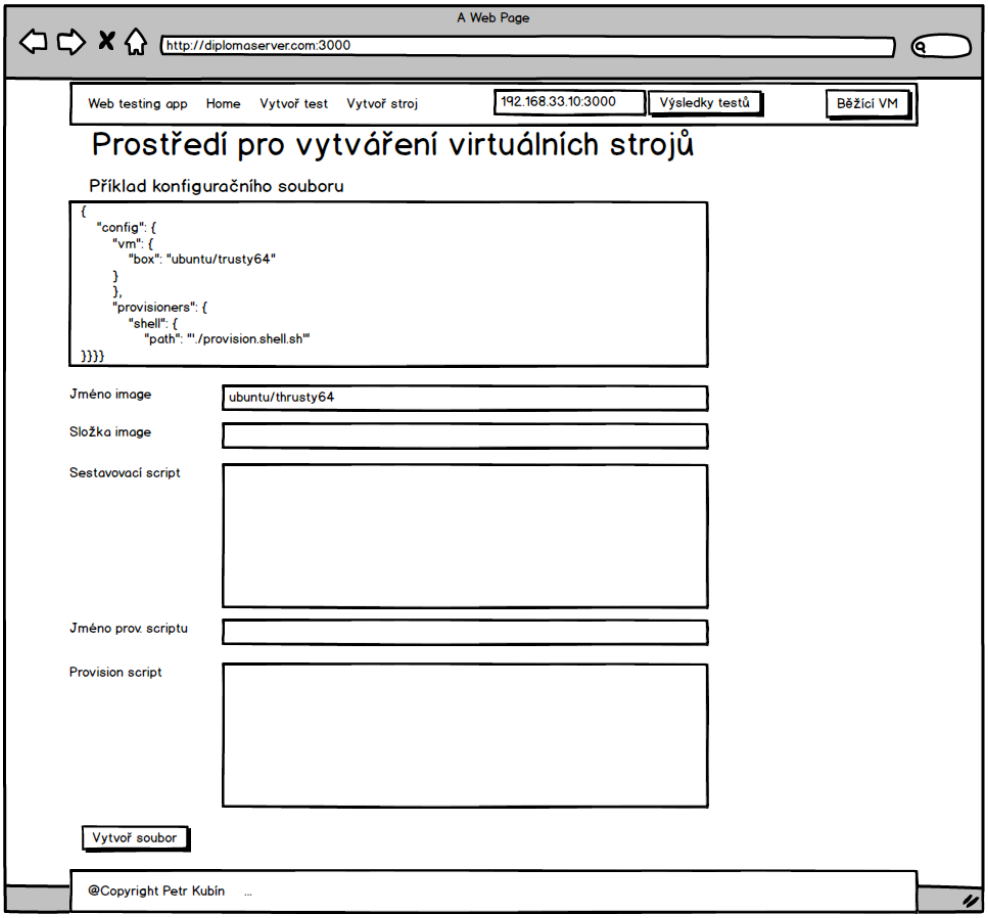
Pro toto ruční nastavení jsem byuži VirtualBox. Jako první je potřeba zkontrolovat verzi `User Additions`, která se musí shodovat s verzí v image. Dalším krokem je instalace všech prohlížečů po nabootování do image. Další krok v nastavování tvoří vytvoření sdílené složky, případně povolení kopírování, které umožní zkopírování serveru do image.

Posledním krokem je správné nastavení síťového rozhraní. Tento krok mi zabral celkem dlouhou dobu, neboť není zakomponovám do dokumentace. Bylo potřeba zvolit v nabídce `Settings->Network->Adapter1->Advanced->Port Forwarding` a správně nakonfigurovat síť. Takto začalo fungovat přeposílání dat ze serveru ve virtuálním stroji na hostující stroj.

Linux Nastavení u Linuxu bylo o poznání jednodušší. Veškeré nastavení šlo provést přes správně sestavený konfigurační soubor a jediná věc, která je potřeba udělat ručně, bylo automatické spouštění serveru na virtuálním stroji po startu grafického rozhraní.

2.2.1.2 Grafické rozhraní

Pro tvorbu prostředí bylo potřeba zpracovat jméno image, které chce uživatel použít. Poté musí uživatel zadat konfigurační script a případně provision script. Na začátku formuláře jsem umístil ukázkový konfigurační soubor.



Obrázek 2.5: Návrh uživatelského rozhraní pro tvorbu prostředí

2.2.1.3 Shrnutí tvorby prostředí

Pro možnosti spouštění a konfigurování virtuálních prostředí jsem použil balíčky `node-vagrant`[31] a `node-virtualbox`. První z balíčků mi umožnil sestavení kostry prostředí, které jsou potřeba kvůli komplexitě a podpory pro Microsoft Windows ještě ručně doupravit a původní myšlenka 100% automatizace přes webovou aplikaci narazila na kámen.

V případě, že by vývojáři Virtualboxu uvedli do produkce plné možnosti nastavení virtuálního stroje a doplnili ho přehlednou dokumentací pro příslušné platformy, byl bych pak schopný rozšířit mou aplikaci o toto vylepšení.

V době vývoje se mi nepovedlo nalézt mechanismus, který by umožňoval univerzální nastavení.

Tento fakt jsem proto kompenzoval možností vybrat již hotový virtuální stroj, který mohu spustit či zastavit pomocí druhého zmiňovaného balíčku. Tím jsem vyřešil nemožnost nastavení virtuálního stroje bez grafického rozhraní.

2.2.2 Část aplikace pro tvorbu testů

Pro vytvoření testu je potřeba sloučit JavaScriptový a HTML soubor. V HTML souboru si uživatel navolí přídavné knihovny, elementy pro testování, konfiguraci proměnných a podobné. V JavaScriptové části testu pomocí `suite.add()` přidá nový test.

Díky použití knihovny BenchmarkJS je poté přidávání testů otázkou okamžiku. Na stránkách `jsperf.com` je možné vytvořit a odladit svůj test na jednom stroji. Poté stačí vzít příslušné položky a zkopírovat do mého formuláře pro vytváření testů. Po sepsání testu stačí potvrdit formulář pro vytvoření souboru.

Na hlavní straně je poté seznam vytvořených testů, které je možné výběrem poslat k testování. Samotná distribuce testovacích souborů se provádí v těle POST požadavku. Zde jsem ještě zvažoval alternativu se sdílenými složkami, ale u nich je problém s různou cestou v adresářové struktuře. Například při vytvoření sdílené složky přes Vagrantfile do domovské složky ve virtuálním stroji, dojde ke správnému připojení a nasdílení dat. Při následném zapnutí přes virtualbox ovšem složka zmizí. Pokud detailněji projdete filesystem, zjistíte, že je daná složka místo v `home` adresáři najednou v `/media/sf_složka`. To je vcelku překvapivé, když je to v konfiguračním souboru nastaveno jinak. V případě Windows virtuálního stroje se jednalo o stejný problém a sdílení složek nefungovalo spolehlivě.

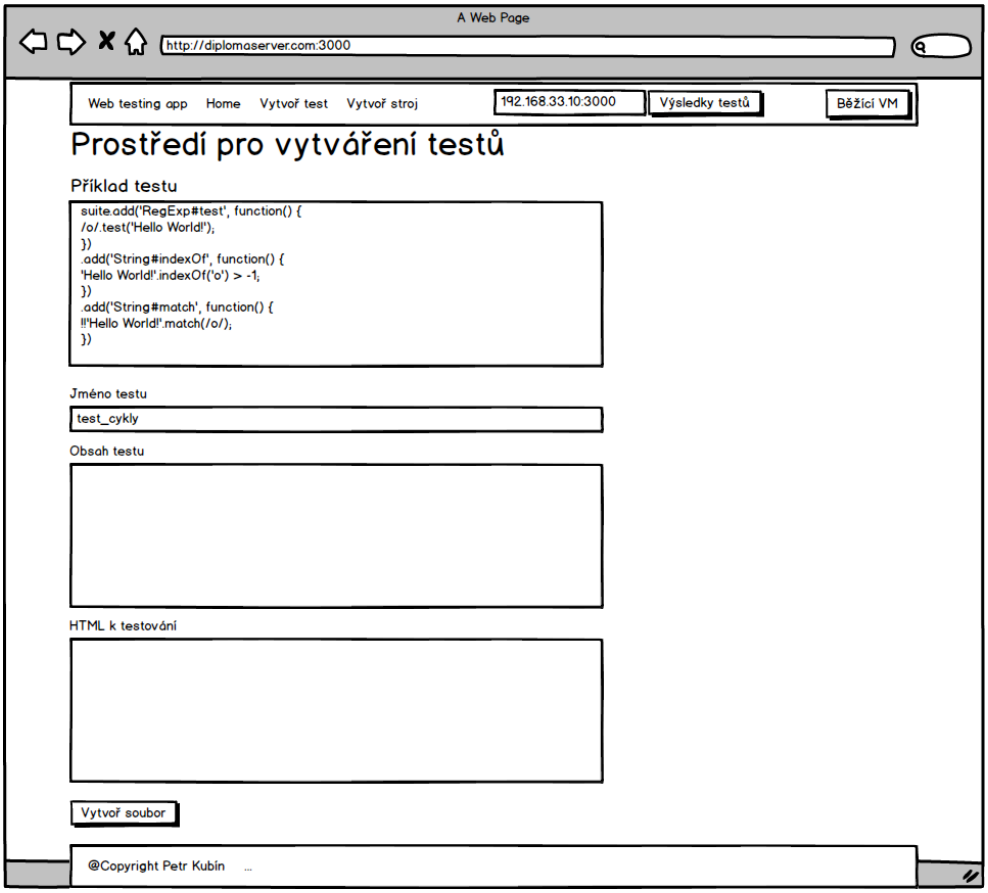
Vytváření výsledného testovacího souboru probíhá rozparsováním vstupních dat a následném spojení do jednoho výsledného html souboru, který obsahuje jak HTML tak JavaScript. Díky sloučení do jednoho testového souboru jsem docílil zjednodušení posílání na virtuální stroje pro následné testování.

2.2.2.1 Grafické rozhraní

Grafické prvky návrhu vytváření testů jsou jméno testu, JavaScriptová část testu a HTML část testu. Na začátek jsem umístil ukázku JavaScriptové části testu a poté následuje standardní formulář.

2.2.2.2 Shrnutí tvorby testů

Pro vytváření testů jsem zvolil jeden formulář, který odpovídá svou koncepcí zápisu testů v jsperf. Takto umožňuji uživatelům jednoduše vytvářet sadu



Obrázek 2.6: Návrh uživatelského rozhraní pro tvorbu testů

testů buď stejným mechanismem, na který jsou zbyklí, nebo překopírováním testových úloh do mé aplikace.

Výsledné sestavení testu probíhá v logice serveru hlavní aplikace a pro jeho funkčnost nebylo potřeba využít žádné knihovny. Bylo by dobré, kdyby vývojáři Vagrantu a VirtualBoxu sjednotili systém sdílení složek, což by umožnilo kompletní automatizaci, bez nutnosti serveru na virtuálním stroji a následné distribuování testových úloh. O samotné spuštění by se pak postaral script, který by například kontroloval obsah sdílené složky. Tím by se značně zjednudišil celý mechanismus.

Výsledné řešení používá k distribuci testů HTML požadavek, kdy je obsah testů součástí těla požadavku. Vyžaduje proto správné nastavení sítě, portů a serveru na straně virtuálního stroje.

Aplikace tak splnila všechny tři funkční požadavky, které na ni byly kladeny. Možnosti testování kompletních webových stránek a jejich nevýhody jsem uvedl v kapitole o testování JavaScriptu. Pro automatizované testování,

které doběhne v rozumné době, bude schopné ověřit limity daného prostředí a poskytovat statisticky zpracovatelné výsledky, jsem se musel omezit na testování pouze úryvků kódu a porovnání jejich výkonnosti. Samostřejmě že se dá napsat i složitější benchmark, ovšem testování kompletní práce s několika JS soubory ve velké aplikaci otestovat moje aplikace neumí.

2.2.3 Část aplikace pro spouštění a hodnocení testů

Nyní se dostáváme k hlavní části aplikace, která zabezpečuje celé testování. Zde bylo potřeba zabezpečit možnost dostat se na vytvoření testu a virtuálních strojů. Dále jsem chtěl mít možnost spustit již vytvořený virtuální stroj, případně ho moci ukončit. O tuto funkcionalitu se stará první polovina aplikace, kde je možné stroj zapnout/vypnout a dále zkontrolovat běžící virtuální stroje.

Kvůli nemožnosti zjistit detailní stav o běhu stroje přes webové rozhraní kvůli nedostatečné podpoře a chybějícímu API ze strany VirtualBoxu zobrazuje aplikace pouze stav běžící/neběžící, nikoli připravený pro testování. Existují částečně připravené nástroje jako například `npm-virtualbox`, nebo balíček `npm-vboxmanage`. Bohužel tyto nástroje nejsou dokončené a plně funkční.

Protože jsem v předchozích částech návrhu zvolil metodu testování pomocí posílání POST pažadavku na server, který běží uvnitř virtuálního stroje, bylo potřeba zajistit zadávání IP adres. První myšlenka směřovala k možnosti získání těchto adres přímo z konfiguračních souborů pro Virtualbox, případně VBoxManage. První řešení selhalo na nemožnosti zjistit, kde přesně ve filesystému se příslušný konfigurační soubor nachází, dále kvůli požadavku na univerzálnost řešení může dojít k vytvoření virtuálního stroje mimo aplikaci bez Vagrantfilu. To vedlo na druhou možnost získat IP adresu z VBoxManage. V dokumentaci k VBoxManage existuje i přepínač, ovšem rozhraní pro nodejs[32] nemá zabudovanou podporu pro práci se všemi vlastnosti VBoxManage.

Uživatel je tak nucen do nepříjemné situace, kdy si musí zapamatovat IP adresu, kterou nastavil při konfiguraci virtuálního stroje. Ta se zadává ve formátu `x.x.x.x:PORT`. V uživatelském rozhraní je připravena ukázka.

Implementace spuštění automatického testování spočívá v poslání POST požadavku na příslušnou IP adresu. Protože se při nastavení virtuálního stroje a jeho zapnutí alokují příslušné prostředky pro jeho běh samostatně, je možné provádět testování na více virtuálních strojích zároveň.

2.2.3.1 Rozbor práce s virtuálními stroji

Pro zjištění, který virtuální stroj je v systému a zda je aktivní, jsem použil `virtualbox` balíček z `npm`. Ten mi umožňuje vylistovat a zobrazit příslušné stroje včetně jejich stavu. Kostra použití vypadá takto:

```
virtualbox.list(function list_callback(vms, error) {
```

```
    if (error) console.log(error);
    var result = [];
    var vals = vms;
    for (var prop in vals) {
      if (vals.hasOwnProperty(prop)) {
        result.push({name: vals[prop].name,
                     state: vals[prop].running});
      }
    }
    machines = result;
    res.render('index',{files:files,machines:result});
  });
```

2.2.3.2 Spuštění testů

Na začátku požadavku si zpracuji soubory, které jsou pro testování vybrané. Poté je distribuuji přes IP adresy na příslušné stroje, kde se o samotné testování stará vlastní server, který na daném stroji běží. Zde je potřeba ručně zadávat IP adresy testovaných strojů. V budoucnu mám v plánu rozšířit mé řešení o možnost přímého vybrání virtuálních strojů z nabídky. K této funkcionalitě bude potřeba spolupráce ze strany vývojářů Vagrantu a VirtualBoxu pro umožnění lepší práce s virtuálním strojem přes NodeJS.

Zpracování dat z formuláře probíhá pomocí metody `createReadStream`, ve které si naplním obsah souborů do proměnné formData. Poté pošlu POST požadavky na všechny IP adresy, které uživatel zadal.

```
app.post('/runTesting', function(req, res) {
  var formData = {};
  machinesIPs = req.body.mytext;
  for (let i = 0; i < files.length; i++) {
   if (req.body[files[i]] == 'on')
    formData[files[i]]=fs.createReadStream(files[i]);
  }

  for (let i = 0; i < req.body.mytext.length; i++) {
    folder = '/test'
    request.post({ url: protocol.concat(IP[i], folder),
                   formData: formData },
      function(err, httpResponse, body) {
        if (err) {
          return console.error('upload failed:', err);
        }
      });
  }
```

```
    res.render('index',{files:files,machines:machines});
});
```

2.2.3.3 Řešení výsledků

Pro zobrazení výsledků jsem zvolil metodu dotázání na API běžící ve virtuálním stroji. Dané API zajišťuje vrácení výsledku ve formátu JSON, který zpracuji v mateřské aplikaci. Výsledek obsahuje testovaný soubor, operační systém a prohlížeč v hlavičce. Dále jsou zobrazeny přesné detaily o testovém prostředí včetně verze prohlížeče, vykreslovacího enginu a dalších. Poté následují informace o jednotlivých testových scénářích a na závěr je uveden vítěz a poražený.

V případě více kandidátů na vítěze se vezmou všichni kandidáti, kteří splňují určitou odchylku od nejlepšího výsledku. Tento jev může nastat například při porovnávání rychlostí jiných zápisů cyklů. Úkolem hlavní aplikace je tedy pouze dotazání na API a následné vykreslení výsledků, které provádím parsováním na frontendu aplikace, neboť se jedná o čistě vizuální vylepšení.

2.2.3.4 Grafické rozhraní

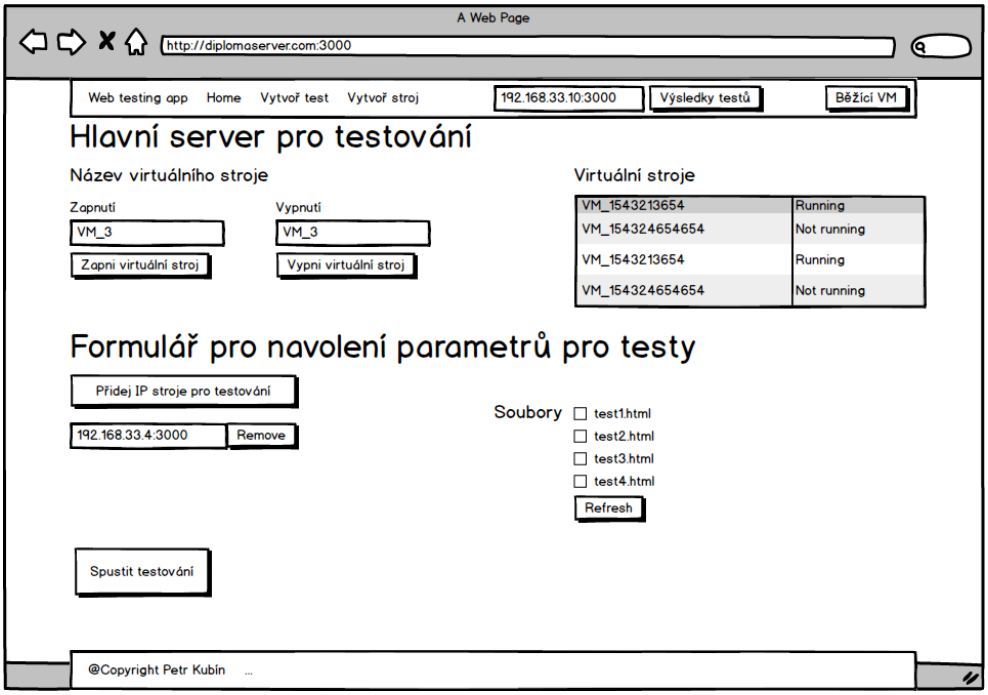
V hlavní aplikaci jsem musel poskládat prvky pro zapnutí a vypnutí stroje, kontrolu běžících strojů a část pro spuštění testování. Zvolil jsem rozložení pro tři sloupce, kdy v prvním sloupci je zapnutí stroje, v druhém je vypnutí a poslední sloupec slouží ke kontrole virtuálních strojů.

Část pro testování jsem vytvořil dvousloupcovou s formulářem pro IP adresy v levém sloupci a výběrem souborů v druhém sloupci. Toto rozložení mi přišlo uživatelsky pochopitelné, neboť se nejdříve uživatel musí postarat o virtuální stroje a poté přechází na samotné testování. Celý design tak reflektuje cestu uživatele skrz mou aplikaci.

Pro zpracování výsledků jsem zvolil znovu tři sloupce, které se při zmenšené obrazovce změní na jeden. Do záhlaví výsledku jsem umístil název testu, operační systém a prohlížeč. Dále jsem přidal tlačítko na přepínání viditelnosti detailů, neboť při vícerém testování je uživatel nepotřebuje znovu a znovu vídat. Poté následuje tabulka a na konci definition list s výsledky. Definition list jsem použil díky jeho HTML významu. Zde už je snímek z reálné aplikace, nikoli z návrhu v Balsamiqu.

2.2.3.5 Shrnutí hlavní webové aplikace

Pro hlavní aplikaci jsem zvolil znovu NodeJS server, který pracuje s virtuálními stroji přes jejich API rozhraní. Zobrazení virtuálních strojů mi umožnil balíček `virtualbox`, který je schopný zobrazit i stroje, které moje aplikace



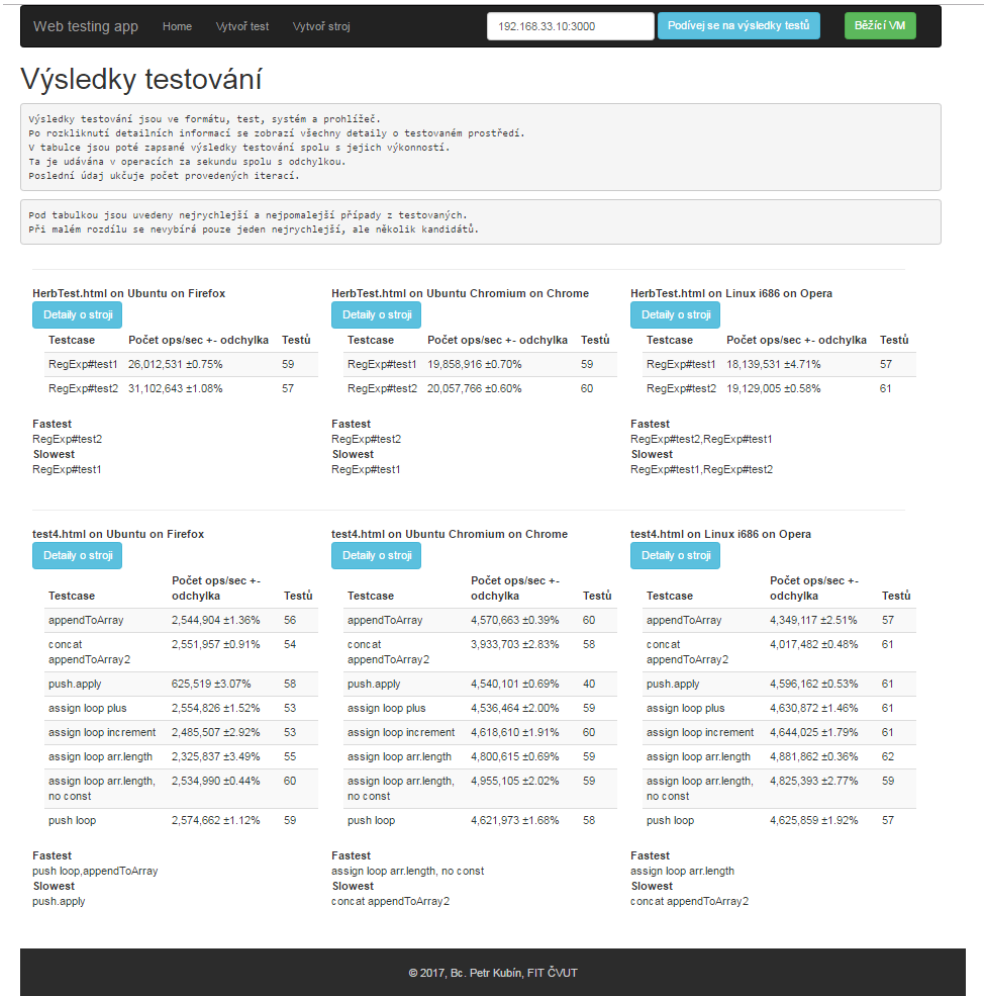
Obrázek 2.7: Návrh uživatelského rozhraní pro tvorbu testů

přímo nevytvořila. Zobrazení testovaných úloh slouží pro zvolení, kterou testovací úlohu pošle aplikace na virtuální stroj. Dále je možné zvolit virtuální stroje pomocí jejich IP adresy z vnitřní sítě.

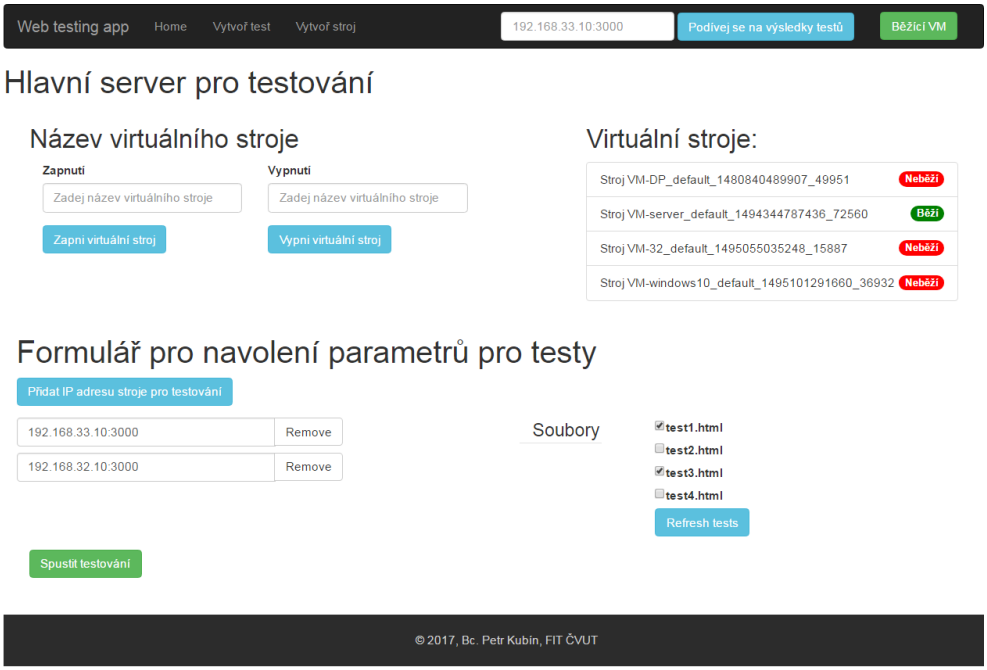
Pokud dojde k úpravě podpory NodeJS a Vagrantu, tak bych rád předělal tuto uživatelsky nepohodlnou variantu zadávání IP adres do podoby výběru přímo virtuálního stroje. Přes stejný systém je možné získat i výsledky testů. Aplikace si v tomto případě pamatuje poslední zadávané IP adresy při spouštění testování a pro výsledky není potřeba tyto adresy znovu zadávat. Sama aplikace se pak dotáže na příslušné IP adresy, které má uložené. Je možné zadat specifickou IP adresu pro zobrazení například jen jednoho testu. Tím se zapamatované adresy vynulují a aplikace použije pouze jedinou zadanou IP.

Posledním funkčním požadavkem je možnost spouštění a zastavování virtuálních strojů. Tento požadavek se povedlo díky balíčku virtualbox. Je tak možné zapínat a vypínat libovolné virtuální stroje, které má uživatel spustitelné na příslušném stroji.

Aplikace tak splnila všechny funkční požadavky, jen požadavek na spouštění testů na příslušném stroji je uživatelsky ne úplně příjemný. Zde přikládám ukázku výsledné aplikace.



Obrázek 2.8: Výsledný vzhled výsledku testů



Obrázek 2.9: Hlavní strana výsledné aplikace

Kapitola 3

Experimenty

3.1 Srovnání výsledků v rámci jednoho stroje

Pro začátek jsem začal zkoumat různá nastavení virtuálního stroje a porovnával jsem výsledky pro jednotlivé běhy. U nastavení virtuálního stroje jsem měnil velikost paměti RAM a dále počet procesorů. Ostatní nastavení či programy jsem neměnil. Pro splnění testování různých verzí prohlížečů je vidět v datailech testování, že například Opera odpovídá základním enginem staršímu Google Chromu. Tím se dá částečně porovnat závislost testování na určité verzi prohlížeče.

Pro testování jsem vždy vzal jeden operační systém a spustil jsem pro něj testování s různými parametry. Kvůli paměťovým nárokům a hardwaru mého počítače jsem si nemohl dovolit spustit veškeré testování naráz. Při výkonějším stroji by bylo možné testování spustit pro všechny nakonfigurované stroje, čímž se měření provádí paralelně a výsledek se dozvíme rychleji.

Pro veškeré měření jsem připravil sadu čtyř testovacích úloh, které se zabývají různým spektrem problémů, pro něž se JavaScript používá. Detailní výsledky jsou na přiloženém CD a v následujících kapitolách uvedu jejich shrnutí. Uvedená výkonnost je měřena v operacích/vteřinu (ops/sec).

3.1.1 Windows 10 64 bit

3.1.1.1 Zkoumání vlivu paměti RAM

Pro první nastavení jsem použil 2048MB RAM spolu se 4 jádry procesoru. Záměrně jsem zvolil více jader procesoru, neboť jsem chtěl co nejvíce eliminovat tuto proměnnou. Pro druhé testování jsem zvětšil paměť RAM na 3019MB, což bylo pro můj počítač maximum. V tabulkách výsledků uvedu pouze testové případy a jejich algoritmy, ve kterých byl vidět rozdíl.

Firefox První test neodhalil žádné závislosti v rámci zvětšení paměti. Druhý test ukázal jistou závislost v prvním případě, který byl schopný zpracovat

Tabulka 3.1: Tabulka výsledků pro Windows 10 a Firefox

Testovaný algoritmus	první stroj ops/sec	druhý stroj ops/sec
(a > b) - (a < b)	5,710 +-2.36%	8,977 +-3.62%
(a < b) ? -1 : (a > b) ? 1 : 0	8,502 +-6.70%	9,843 +-3.44%
a == b ? 0 : a < b ? -1 : 1	11,368 +-1.19%	11,929 +-1.55%
For loop, basic	107,719 +-1.64%	134,033 +-1.20%
For loop, cached	95,807 +-3.00%	126,993 +-0.30%
For loop, reversed	89,584 +-1.78%	117,289 +-0.87%
For loop, jQuery	7,070 +-4.84%	9,907 +-1.17%
For loop, underscore	545 +-4.06%	778 +-0.79%
Foreach loop	143,389 +-2.26%	131,059 +-1.12%
For in loop	357 +-4.79%	385 +-4.10%
While loop, basic	113,011 +-3.96%	153,585 +-1.98%
While loop, cached	121,362 +-0.87%	153,749 +-1.16%
While loop, reversed	77,474 +-1.37%	95,626 +-0.80%
appendToArray	2,274,796 +-2.29%	2,964,343 +-1.07%
concat appendToArray2	3,024,969 +-1.29%	3,273,198 +-0.49%
push.apply	759,865 +-1.68%	794,776 +-1.09%
assign loop plus	2,720,102 +-6.66%	2,973,902 +-0.80%
assign loop increment	1,629,900 +-6.09%	2,738,080 +-0.50%
assign loop arr.length	1,577,787 +-3.99%	2,758,111 +-1.45%
assign loop arr.length, no const	1,679,809 +-2.58%	2,977,210 +-0.41%
push loop	1,870,648 +-3.29%	3,463,677 +-0.40%

1.5krát více operací než s méně pamětí. Pro třetí testovaný vzorek, který se skládá z různých zápisů smyček a porovnávání jejich rychlosti, bylo jasně patrné zrychlení ve všech případěch pro stroj s více pamětí RAM. Také došlo k promíchání nejrychlejších výsledků.

Při posledním testu se ukázalo znovu, že stroj s více paměti je schopný spočítat více operací za vteřinu. Následně došlo k promíchání nejrychlejších výsledků.

Chrome V prvním testu došlo k mírnému zrychlení přes všechny tři testové případy. Pořadí výsledků zůstalo beze změn.

V druhém případě nastalo zrychlení ve všech třech testovaných úkolech. Největší rozdíl byl v druhém testovaném bodě a jednalo se o 45% nárůst výkonu.

Pro třetí testovaný scénář se opět potvrdilo zrychlení a i promíchání celkových vítězů.

V posledním testu došlo k překvapivé změně i u nejpomalejšího algoritmu, který místo concat appendToArray byl rázem assign loop plus. Pro přesné

Tabulka 3.2: Tabulka výsledků pro Windows 10 a Google Chrome

Testovaný algoritmus	první stroj ops/sec	druhý stroj ops/sec
(a > b) - (a < b)	17,479 +-2.02%	25,036 +-1.02%
(a < b) ? -1 : (a > b) ? 1 : 0	26,934 +-3.66%	39,060 +-1.77%
a == b ? 0 : a < b ? -1 : 1	34,772 +-1.19%	38,402 +-1.19%
For loop, basic	63,147 +-1.68%	89,098 +-0.53%
For loop, cached	63,402 +-1.07%	89,304 +-0.47%
For loop, reversed	69,752 +-3.97%	89,615 +-0.27%
For loop, jQuery	2,999 +-1.56%	3,085 +-0.27%
For loop, underscore	1,269 +-1.72%	1,305 +-0.98%
Foreach loop	4,031 +-2.18%	4,946 +-0.45%
For in loop	1,915 +-2.20%	2,937 +-1.00%
While loop, basic	10,456 +-2.59%	15,610 +-0.56%
While loop, cached	11,880 +-1.91%	17,987 +-0.35%
While loop, reversed	9,964 +-47.24%	14,055 +-53.49%
appendToArray	3,785,904 +-3.23%	4,889,168 +-1.49%
concat appendToArray2	960,933 +-57.51%	4,345,526 +-0.63%
push.apply	3,315,783 +-1.95%	4,874,925 +-1.95%
assign loop plus	3,740,691 +-2.55%	4,375,077 +-1.97%
assign loop increment	3,777,796 +-1.55%	4,461,958 +-1.12%
assign loop arr.length	4,109,464 +-1.96%	4,826,423 +-1.04%
assign loop arr.length, no const	4,038,180 +-1.44%	4,785,070 +-1.17%
push loop	3,898,310 +-1.85%	4,654,535 +-1.05%

názvy se podívejte na přiložené CD. Dále bylo překvapivé, že všechny testy se srovnaly na celkem podobné hranici a nedocházelo k žádným extrémním rozdílům.

Opera V prvním testu došlo k zrychlení ve všech třech případech. V druhém testu se jednalo o 50% zrychlení počítané přes všechny tři testované případy.

Třetí test také probíhal o něco rychleji a u posledního testu nebyl scénář o nic jiný než v předchozích. Došlo také k promíchání nejrychlejších a nejpomalejších scénářů.

Edge Pro první test nebylo z výsledků patrné žádné extra zlepšení. Pouze první případ ukázal lehký nárůst výkonu. V druhém testu byly výsledky neznatelně odlišné.

Třetí test neukázal žádné závislosti na paměti RAM pro Microsoft Edge. Změnil se pouze nejrychlejší algoritmus, který si ovšem ani v prvním případě nevedl špatně. Poslední test znovu potvrdil proházení výsledků a zrychlení ve všech případech.

Tabulka 3.3: Tabulka výsledků pro Windows 10 a Operu

Testovaný algoritmus	první stroj ops/sec	druhý stroj ops/sec
(a > b) - (a < b)	13,346 +-3.58%	25,512 +-1.32%
(a < b) ? -1 : (a > b) ? 1 : 0	17,988 +-4.43%	40,527 +-0.71%
a == b ? 0 : a < b ? -1 : 1	21,714 +-2.75%	38,772 +-1.26%
For loop, basic	72,810 +-4.10%	88,894 +-0.74%
For loop, cached	79,458 +-1.36%	89,327 +-0.61%
For loop, reversed	80,649 +-0.79%	89,729 +-0.30%
For loop, jQuery	2,621 +-1.74%	3,054 +-0.61%
For loop, underscore	1,105 +-1.57%	1,257 +-2.53%
Foreach loop	4,102 +-4.47%	4,315 +-6.27%
For in loop	2,043 +-1.12%	2,957 +-1.62%
While loop, basic	11,100 +-1.17%	12,096 +-0.82%
While loop, cached	15,142 +-2.96%	16,612 +-3.20%
While loop, reversed	12,569 +-49.31%	12,684 +-66.64%
appendToArray	4,123,722 +-3.72%	4,909,734 +-1.64%
concat appendToArray2	3,835,806 +-1.08%	4,393,743 +-0.49%
push.apply	4,498,737 +-0.81%	4,997,979 +-0.45%
assign loop plus	3,913,038 +-1.94%	4,442,646 +-1.64%
assign loop increment	3,926,990 +-2.11%	4,475,904 +-2.05%
assign loop arr.length	4,228,459 +-1.97%	4,897,838 +-1.85%
assign loop arr.length, no const	4,218,758 +-1.82%	4,914,503 +-1.21%
push loop	4,123,960 +-2.24%	4,668,894 +-1.10%

3.1.1.2 Zkoumání vlivu počtu procesorů

Pro nastavení počtu procesorů jsem zvolil v první variantě 2 procesory a v druhé dvojnásobek (4).

Firefox V prvním ani druhém případě nedošlo k viditelnému zlepšení. Testované časy byly téměř totožné. Třetí případ ukázal opačný jev než paměť RAM, kdy stroj s vyšším počtem procesorů dosáhl horších výsledků. Ani v posledním testu se nepotvrdila žádná závislost.

Chrome Testování pro další prohlížeč bylo v prvním případě téměř totožné. Druhý a třetí případ ukázal zpomalení pro vyšší počet procesorů. Při posledním testu nedošlo k žádnému výraznému rozdílu v testovaných časech.

Opera Pro první dva testy nastalo již očekévané zpomalení. Třetí test byl v první části rychlejší a v druhé pomalejší pro stroj s vyšším počtem proce-

sorů. V posledním testu vyhrál výkonnější počítač nad slabším s velmi malým rozdílem v měřené výkonnosti.

Edge U prohlížeče Microsoft Edge se projevil rozdíl pouze v druhém testu, který dopadl lépe pro stroj s menším počtem procesorů. První test a dále poslední dva testy dopadly téměř totožně.

3.1.2 Ubuntu 64 bit

Pro Ubuntu ve verzi 64 bit jsem dané testové prostředí nenastavoval, neboť jsem měl jeho 32 bitového kolegu. Výsledky tohoto virtuálního stroje budu porovnávat až přes celkové výsledky testů.

3.1.3 Ubuntu 32 bit

3.1.3.1 Zkoumání vlivu paměti RAM a počtu procesorů

Protože samotné zvýšení počtu procesorů v předchozím případě nemělo nijak prokazující výsledky, rozhodl jsem se sloučit zvýšení paměti RAM a počtu procesorů do jednoho bodu. Z předchozího testování mám pocit, že nová jádra procesoru byla schopna naplno využít zbývající paměť a ještě by nějakou paměť navíc snesla. To se potvrdilo při předchozí diskuzi o vlivu paměti RAM, kdy oba testované stroje měli vyšší počet procesorů.

Firefox V prvním testu nebyl nijak znatelný rozdíl a druhý test poskytl téměř identická data. Pro třetí test byla data znovu téměř totožná a i výsledné nejrychlejší a nejpomalejší testy byly totožné. Pouze poslední test v prvních dvou testovaných algoritmech zaznamenal výraznější zrychlení pro stroj s více pamětí a procesorových jader.

Chrome Pro prohlížeč Chrome byly výsledky téměř identické pro všechny testované úlohy a nebylo možné určit žádnou závislost.

Opera Pro prohlížeč Opera byly první tři testové úlohy jako přes kopírák. Poslední test byl ovšem rychlejší pro výkonější konfiguraci systému. Zrychlení se pohybovalo okolo 12.5 %.

3.1.4 Hodnocení závislosti na paměti RAM

Během testování se projevila určitá závislost výkonnosti na velikosti paměti RAM. Samotné zvětšení paměti bez zvýšení počtu vláken procesoru ovšem nemělo velmi výrazný vliv, neboť danou paměť nebyl schopný procesor využít. Pro virtuální stroj s Windows 10 bylo počátečních 2048MB paměti téměř na spodní hranici únostnosti a proto zvýšení na 3019MB značně pomohlo procesoru při odkládání během výpočtů.

Pro verzi s Ubuntu 32 bit nedocházelo k téměř žádné výrazné změně. Je to způsobené tím, že tento operační systém nespotřebovává takové množství výpočetních prostředků pro svůj běh a následná paměť RAM již nebyla efektivně využita. Proto je dobré při sestavování virtuálního stroje zjistit jeho limity a nastavit pameť RAM na úroveň, kdy se s ní nebude plýtvat a zároveň nebude chybět. Při velmi špatném nastavení paměti dojde k chybě při spuštění prohlížeče, neboť sám prohlížeč má zajištěné ošetření při nedostatku výpočetních prostředků.

3.1.5 Hodnocení závislosti na paměti počtu procesorů

Počet procesorů neměl až tak zásadní vliv při mých testovaných úlohách a nastavení stroje. Nejspíše by se tento jev mohl projevit při zpracovávání paralelních úloh či běhu několika prohlížečů současně. Je ale vždy dobré nastavit počet procesorů na rozumnou mez, aby se při výpočtu zkloubil jejich výkon s přidělenou RAM pamětí.

3.2 Srovnání výsledků v rámci více strojů

Pro srovnání výsledků v rámci několika virtuálních strojů jsem použil konfiguraci s 2048MB RAM a 4 jádry procesoru. Společné prohlížeče jsou Firefox, Chrome a Opera. Microsoft Edge není dostupný pro Linux a tak ze srovnání vypadává.

3.2.1 Firefox

Druhý test Druhý test vyhrálo Ubuntu 64bit. Oproti Windows10 mělo v první metrice dvojnásobný benchmark. Ubuntu 32bit bylo přesně mezi 64bitovými systémy.

Třetí test V testování smyček byly obě linuxové distribuce velmi vyrovnané. Windows10 zaostal ve většině jednotlivých testů.

Čtvrtý test Poslední test dopadl přesně jako předchozí.

3.2.2 Chrome

Druhý test Google Chrome dopadl pro Linuxové distribuce téměř totožně. Windows 10 zaostalo za oběma konkurenty.

Třetí test Zde dopadlo testování jako přes kopírák. Linuxové distribuce předčili Windows10.

Čtvrtý test Poslední test dopadl stejně jako předchozí.

3.2.3 Opera

Pro prohlížeč Opera dopadly výsledky totožně jako pro Chrome. Žádné změny v pořadí mezi jednotlivými virtuálními stroji zde nebyly.

3.3 Srovnání prohlížečů v rámci jednoho stroje

Pro srovnání prohlížečů v rámci jednoho stroje jsem zvolil Ubuntu 64bit a Windows10. Pro další srovnání se můžete podívat na přiložené CD a projít si výsledky. Jednotky jsou znovu operace/vteřinu s procentuální odchylkou při měření.

3.3.1 Ubuntu 64bit

Tabulka 3.4: Srovnání prohlížečů v rámci jednoho stroje první test

Firefox ops/sec	Chrome ops/sec	Opera ops/sec
2,426,774 +-9.60%	8,272,902 +-0.52%	5,513,893 +-1.91%
15,857,516 +-2.08%	22,464,880 +-0.81%	32,565,613 +-0.96%
1,585,503 +-12.63%	6,076,346 +-7.92%	4,904,042 +-1.43%

První test V prvním testu exceloval prohlížeč Google Chrome pro průměrný výsledek. Nejrychlejší byla ovšem Opera pro prostřední benchmark s dvojnásobnou rychlostí než Firefox.

Tabulka 3.5: Srovnání prohlížečů v rámci jednoho stroje druhý test

Firefox ops/sec	Chrome ops/sec	Opera ops/sec
9,638 +-1.98%	26,418 +-1.00%	35,367 +-16.00%
11,506 +-2.05%	39,585 +-1.88%	42,882 +-1.57%
12,642 +-1.82%	38,235 +-1.88%	44,527 +-1.24%

Druhý test V druhém testu byla Opera jasným vítězem. O trochu horší benchmark byl naměřen u Google Chromu a poslední zůstal Firefox.

Třetí test Ve třetím testu zabodoval Firefox. Opera a Google Chrome poskytovali srovnatelné výsledky.

Čtvrtý test Poslední test vyhrál Google Chrome. Překvapivou metrikou se stala metoda push.apply, kde činil rozdíl úctihodný šestinásobek v rozdílu výkonu mezi Chromem a Firefoxem. Opera se umístila mezi těmito prohlížeči. Pro lepší orientaci v měření zde přiložím ukázku kódu push.apply:

Tabulka 3.6: Srovnání prohlížečů v rámci jednoho stroje třetí test

Firefox ops/sec	Chrome ops/sec	Opera ops/sec
134,374 +-0.42%	90,481 +-0.42%	85,547 +-4.96%
134,744 +-0.43%	89,613 +-0.59%	67,102 +-0.50%
134,135 +-0.47%	89,606 +-0.41%	90,193 +-0.34%
6,525 +-5.25%	2,875 +-5.43%	3,938 +-6.91%
1,257 +-0.70%	1,394 +-1.36%	1,566 +-0.97%
133,014 +-0.41%	5,001 +-0.28%	6,745 +-0.31%
527 +-3.31%	3,443 +-1.48%	2,612 +-1.37%
133,763 +-0.31%	15,863 +-0.53%	89,941 +-0.29%
134,364 +-0.29%	18,107 +-0.38%	89,975 +-0.32%
89,066 +-0.39%	19,168 +-0.90%	67,063 +-0.49%

Tabulka 3.7: Srovnání prohlížečů v rámci jednoho stroje čtvrtý test

Firefox ops/sec	Chrome ops/sec	Opera ops/sec
3,449,720 +-0.66%	4,971,265 +-0.47%	3,923,398 +-2.91%
3,445,586 +-0.64%	3,959,605 +-5.60%	3,401,449 +-5.84%
775,113 +-0.98%	4,732,314 +-2.97%	2,967,451 +-2.90%
3,357,167 +-0.84%	3,971,343 +-7.25%	2,950,963 +-4.07%
3,295,001 +-0.56%	3,695,198 +-9.64%	2,998,789 +-2.50%
3,323,097 +-0.39%	4,634,437 +-9.33%	3,161,354 +-3.47%
3,314,520 +-0.58%	4,834,713 +-2.22%	3,207,842 +-2.75%
3,480,495 +-0.49%	4,747,280 +-1.40%	3,231,979 +-4.62%

```
var arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
var arr2 = ['A','B','C','D','E','F','G','H','I','J'];
.add('push.apply', function() {
  var arr3 = []
  Array.prototype.push.apply(arr3, arr1);
  Array.prototype.push.apply(arr3, arr2);
})
```

Kapitola 4

Testování

Testování je nedílnou součástí tvorby každého software. Už od jednotkových testů, které slouží k odhalení chyb v algoritmizaci či logice aplikace až po intergační a uživatelské testování. V mé práci se zmíním o provedení Heuristické analýzy, díky které jsem odhalil několik chyb.

4.1 Heuristická analýza

Jednou z forem testování bez přítomnosti uživatelů je heuristická analýza[33][34]. Je založena na splnění pravidel heuristicky. Těch existuje vcelku velké množství. Pro mou práci jsem zvolil Nielsonovu heuristiku, která je nejběžněji používaná.

Pro správné otestování pomocí heuristiky je potřeba zajistit experty. Správný expert by měl být obeznámen v vývojem webových aplikací a měl by rozumět tématu. Pro možné nepřesnosti v překladu evedu názvy jednotlivých bodů heuristiky v angličtině.

4.1.1 Jednotlivé body heuristické analýzy

1. Visibility of system status

   - **Aplikace neindikuje chybové stavy, uživatel se musí dívat do konzole serveru:**
     Indikaci chybových stavů jsem doplnil pomocí flash messages.
   - **Není vidět aktuální průběh testu:**
     Kvůli paměťovým nárokům na přenos grafického rozhraní do webové aplikace jsem neimplementoval tuto funkcionalitu. Průběh testu může uživatel sledovat přes VirtualBox nebo přímo v GUI virtuálního stroje, kde mu testy probíhají přímo před očima bez jeho zásahu.

- **U výsledků se musí uživatel vracet tlačítkem zpět v prohlížeči:**
  Zde jsem doplnil navigační lištu aplikace a upravil vzhled pro lepší orientaci ve výsledcích a možnost cestování v aplikaci.

2. Match between system and the real world

   - **Zde experti nenašli žádné prohřešky.**

3. User control and freedom

   - **Vracení se na hlavní stránku z výsledků pomocí zpět v prohlížeči:**
     Tuto opravu jsem vyřešil hlavní navigační lištou v celé aplikaci.
   - **Vypnutí testování se provádí pomocí vypnutí virtuálního stroje:**
     Testování nelze přerušit jinak a nepřijde mi to jako nijak velký nedostatek.

4. Consistency and standards

   - **Nedostatečně vysvětlené pojmy pro nezkušené uživatele:**
     Doplnil jsem specifické pojmy speciálním popisným HTML tagem.

5. Error prevention

   - **Chybí indikace ve webové aplikaci, je pouze v konzoli:**
     Tento nedostatek jsem opravil pomocí flash messages pod hlavní navigační lištou.

6. Recognition rather than recall

   - **Nutnost pamatování IP adresy stroje:**
     Tuto nevýhodu jsem zmiňoval již v samotném návrhu. Po přepracování doplňků pro nodejs a sjednocení práce s virtuálními složkami bude teprve možné zajistit bezpečné vytvoření kompletního virtuálního stroje a jeho spouštění podle jména.

7. Flexibility and efficiency of use

   - **Aplikace si po spuštění testů pamatuje IP adresy, které se nemusí znovu ručně zadávat při přechodu na výsledky.**

8. Aesthetic and minimalist design

- **Aplikace by potřebovala rozšířit o několik nápověd pro uživatele neznalé problematiky:**
  Nápovědy jsem přidal ke všem specifickým pojmům.

9. Help users recognize, diagnose, and recover from errors

   - **Apliakce neobsahuje moc chybových zpráv:**
     Byly doplněny flash messages, které chybové a success zprávy uživateli zobrazí.

10. Help and documentation

    - **Chybí help, nejčastěji byly uživatelé nesví z provision scriptu:**
      Jako návod pro použití mé diplomové práce slouží tento text. Dále jsem doplnit určité informace, které při používání aplikace uživatelům pomohou.

Závěr

Cílem práce bylo vytvořit webovou aplikaci umožňující konfiguraci a sestavení testovaných prostředí, testovaných úloh a spouštění automatizovaných testů v těchto prostředích. Po návrhu a prvním prototypu aplikace jsem provedl heuristickou analýzu pro odhalení chyb.

Pro mé řešení jsem použil NodeJS server jak na straně hlavní webové aplikace, tak pro API na straně virtuálních strojů. Tím jsem zabezpečil univerzální rozhraní pro běh nezávislý na operačním systému, který je schopný komunikace s hlavní aplikací.

Pro front end aplikace jsem použil Bootstrap, JQuery a vlastní kaskádové styly. Výsledkem je webová aplikace schopná konfigurace virtuálních strojů s omezeními, které se týkají nastavení uvnitř virtuálního stroje, k nimž je potřebné grafické rozhraní. Do webové aplikace jsem v případě sjednocení či rozšíření od vývojářů Vagrantu a VirtualBoxu schopen doimplementovat příslušné části, které by zabezpečili 100% automatizaci a nezávislost na ručním doladění virtuálních strojů.

Aplikace bude dostupná v produkční verzi na githubu pro ostatní webové vývojáře a doufám, že jim pomůže při testování a pochopení fungování virtuálního stroje jako celku spolu s prohlížeči.

Literatura

[1] Statcounter global stats - browser, os, search engine including mobile usage share_2017. 2017, [Cited 2017-03-17]. Dostupné z: http://gs.statcounter.com

[2] The history of the web - w3c wiki_2017. 2017, [Cited 2017-03-06]. Dostupné z: https://www.w3.org/wiki/The_history_of_the_Web

[3] Google, V.: The evolution of the web. 2017, [Cited 2017-05-06]. Dostupné z: http://www.evolutionoftheweb.com

[4] Honneffer, D.: Opera Software ASA - Opera version history. 2017, [Cited 2017-05-22]. Dostupné z: http://www.opera.com/docs/history/presto/#o1

[5] Index of /pub/opera-developer/_2017. 2017, [Cited 2017-02-15]. Dostupné z: http://get.geo.opera.com/pub/opera-developer

[6] Seth, G.; Foresti, A.: Microsoft Edge's JavaScript engine to go open-source - Microsoft Edge Dev Blog. 2017, [Cited 2017-04-13]. Dostupné z: https://blogs.windows.com/msedgedev/2015/12/05/open-source-chakra-core/#vQleIVkpCeoxBAQA.97

[7] History of the mozilla project_2017. 2017, [Cited 2017-03-12]. Dostupné z: https://www.mozilla.org/en-US/about/history/details

[8] Timeline - mozillawiki_2017. 2017, [Cited 2017-03-04]. Dostupné z: https://wiki.mozilla.org/Timeline

[9] Levy, S.: INSIDE CHROME: THE SECRET PROJECT TO CRUSH IE AND REMAKE THE WEB. 2017, [Cited 2017-05-13]. Dostupné z: https://www.wired.com/2008/09/mf-chrome

[10] The webkit open source project_2017. 2017, [Cited 2017-03-15]. Dostupné z: https://webkit.org/project

[11] Gecko faq_2017. 2017, [Cited 2017-03-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Gecko/FAQ

[12] Blink: a rendering engine for the chromium project_2017. 2017, [Cited 2017-03-13]. Dostupné z: https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html

[13] Thompson, S.: Retiring Octane. 2017, [Cited 2017-04-13]. Dostupné z: https://v8project.blogspot.cz/2017/04/retiring-octane.html?m=1

[14] Spidermonkey 1.8.8_2017. 2017, [Cited 2017-03-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Releases/1.8.8#Platform_support

[15] Ratschan, S.: Formální Metody a Specifikace (LS 2016) Praktické ověřování správnosti programů 2. 2016, [Cited 2017-05-13]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-FME/_media/lectures/lecture_9_practical2_v3.pdf

[16] Computer architecture: a quantitative approach, third edition_2017. 2017, [Cited 2017-04-18]. Dostupné z: http://www.iuma.ulpgc.es/~nunez/clases-micros-para-com/clases-mpc-slides-links/Copy%20of%20(Ebook20Pdf)%20Computer%20Architecture%20A%20Quantitative%20App.pdf

[17] Segal, L.; Plante, N.: Module: Capybara — Documentation for capybara (2.13.0). 2017, [Cited 2017-04-18]. Dostupné z: http://www.rubydoc.info/gems/capybara/Capybara

[18] Jusufbegovic, B.: The Basics of Capybara and Improving Your Tests — SitePoint. 2017, [Cited 2017-05-21]. Dostupné z: https://www.sitepoint.com/basics-capybara-improving-tests

[19] Bynens, M.; Dalton, J.-D.: Bulletproof JavaScript benchmarks. 2017, [Cited 2017-05-06]. Dostupné z: https://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks

[20] Bynens, M.: JsPerf: JavaScript performance playground. 2017, [Cited 2017-02-20]. Dostupné z: https://jsperf.com

[21] Bynens, M.; Dalton, J.-D.: Benchmark.js. 2017, [Cited 2017-02-15]. Dostupné z: https://benchmarkjs.com

[22] Hossain, M.: Benchmark.js: how it works - monsur.hossa.in. 2017, [Cited 2017-05-12]. Dostupné z: http://monsur.hossa.in/2012/12/11/benchmarkjs.html

[23] Kieffer, R.: broofa/jslitmus. 2017, [Cited 2017-02-05]. Dostupné z: https://github.com/broofa/jslitmus

[24] Jetstream 1.1 — in depth analysis_2017. 2017, [Cited 2017-02-12]. Dostupné z: http://browserbench.org/JetStream/in-depth.html

[25] Introduction - vagrant by hashicorp_2017. 2017, [Cited 2017-03-15]. Dostupné z: https://www.vagrantup.com/intro/index.html

[26] File system | node.js v8.1.2 documentation_2017. 2017, [Cited 2017-02-22]. Dostupné z: https://nodejs.org/api/fs.html

[27] Window.close() not working in firefox_2017. 2017, [Cited 2017-02-4]. Dostupné z: https://stackoverflow.com/questions/28250054/window-close-not-working-in-firefox

[28] Raboy, N.: Get Remote HTML Data And Parse It In Express For NodeJS. 2017, [Cited 2017-05-05]. Dostupné z: https://www.thepolyglotdeveloper.com/2015/05/get-remote-html-data-and-parse-it-in-express-for-nodejs

[29] Creating a windows 10 base box for vagrant with virtualbox | huestones_2017. 2017, [Cited 2017-02-16]. Dostupné z: http://huestones.co.uk/node/305

[30] Linkletter, B.: How to emulate a network using VirtualBox. 2017, [Cited 2017-02-01]. Dostupné z: http://www.brianlinkletter.com/how-to-use-virtualbox-to-emulate-a-network

[31] Mujagic, E.: node-vagrant. 2017, [Cited 2017-02-08]. Dostupné z: https://www.npmjs.com/package/node-vagrant

[32] Zgadzaj, M.: Change(b)log: Benchmarking Node.js - basic performance tests against Apache + PHP. 2017, [Cited 2017-04-18]. Dostupné z: http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php

[33] Zikovský, P.: Návrh UI, prototypy, winder semestr 2016, lecture. 2016, [Cited 2017-04-13]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-NUR/_media/lectures/x02-navh_a_prototyping.pdf

[34] Zikovský, P.: Testing without users, winter semester 2016, lecture. 2016, [Cited 2017-04-13]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-NUR/_media/lectures/x03-semestralka_testing_without_users.pdf

Příloha A

Seznam použitých zkratek

GUI Graphical user interface

XML Extensible markup language

IE Internet Explorer

MS Misrosoft

JS JavaScript

JIT Just In Time

HTTP Hypertext Transfer Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

OO Objektově orientované

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `MasterServer` ..... hlavní aplikační server
  - `Ubuntu64bit` ..... image Ubuntu64 bit použitá v experimentální části
  - `Ubuntu32bit` ..... image Ubuntu32 bit použitá v experimentální části
  - `Windows10` ..... image Windows10 použitá v experimentální části
  - `experimenty` ..... uložené výsledky experimentů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Kubin_Petr_2017.pdf` ..... text práce ve formátu PDF