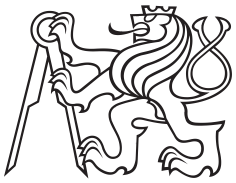
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Doplněk prohlížeče pro administraci nástroje Pi-hole

Jan Kolovecký

Školitel: Ing. Martin Ledvinka
Obor: Softwarové inženýrství a technologie
Květen 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kolovecký** Jméno: **Jan** Osobní číslo: **475384**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Doplněk prohlížeče pro administraci nástroje Pi-hole

Název bakalářské práce anglicky:

Pi-hole admin browser plugin

Pokyny pro vypracování:

1. Seznamte se s nástrojem Pi-hole, jeho administrací a možnostmi jejího aplikačního využití.
2. Analyzujte možnosti administrace nástroje Pi-hole. Porovnejte jejich výhody a nevýhody.
3. Navrhněte aplikaci pro administraci Pi-hole, pomocí které bude možné snadno spravovat lokální instanci z klientského prohlížeče.
4. Naimplementujte navrženou administrační aplikaci.
5. Ověřte použitelnost vytvořeného řešení pomocí uživatelského testování./ 1. Become familiar with Pi-hole, its administration and the possibilities of its usage.
2. Analyze the available means of Pi-hole administration. Compare their pros and cons.
3. Design a Pi-hole administration tool which will be used to manage the local Pi-hole instance from a client's browser.
4. Implement the designed solution.
5. Validate the viability of your solution via usability testing.

Seznam doporučené literatury:

[1] Pi-hole LLC, Pi-hole documentation, 2020
[2] P. Mehta, Creating Google Chrome Extensions, Apress, 2016
[3] MDN contributors, Browser Extensions, 2020

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Ledvinka, skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **12.02.2021** Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2022**

Ing. Martin Ledvinka podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. | Datum převzetí zadání

Podpis studenta

Poděkování

Děkuji vedoucímu práce, panu Ing. Martinovi Ledvinkovi, za ochotu vést mou práci, jeho rady a čas strávený na průběžných konzultacích i mimo ně. Mé poděkování patří také Kamile Topinkové za její nezastupitelnou pomoc při jazykové korektuře.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a v souladu s Metodickým pokynem o dodržování etických principů pro vypracování závěrečných prací, a že jsem uvedl všechny použité informační zdroje.

V Praze, 17. května 2021

Jan Kolovecký

Abstrakt

Tato práce se zabývá platformou Pi-hole a vytvořením doplňku do prohlížeče, přes který by bylo možné spravovat lokální instanci Pi-hole. Kromě samotného návrhu a implementace doplňku tato práce obsahuje popis administrace Pi-hole i stručný úvod do vývoje doplňků do prohlížečů. Cílem této práce je popsání potřebných nástrojů, vytyčení důležitých pojmů, navržení, vytvoření a otestování doplňku do prohlížeče, přes který bude možno administrovat instanci Pi-hole.

Klíčová slova: Pi-hole, doplněk do prohlížeče, blokování reklam, DNS sinkhole, Raspberry Pi, Java, Quarkus, Docker, GraalVM, QEMU

Školitel: Ing. Martin Ledvinka

Abstract

This thesis researches the topic of the Pi-hole platform and creation of a browser plugin, through which user would be able to administrate local instance of Pi-hole. In additon to design and creation of the plugin, this thesis contains description of Pi-holes administration and brief introduction into development of browsers plugins. The goal of thesis is to define needed instruments, define critical terms, design, implement and test plugin through which it would be possible to administrate instance of Pi-hole.

Keywords: Pi-hole, browser addon, browser extension, add-blocking, DNS sinkhole, Raspberry Pi, Java, Quarkus, Docker, GraalVM, QEMU

Title translation: Pi-hole admin browser plugin

Obsah

Obrázky

Tabulky

Kapitola 1
Úvod

1.1 Předmluva

Internet je v dnešní době zásadní částí lidského života. Vždyť průměrný Čech na něm stráví měsíčně desítky hodin [1]. Ale při prohlížení internetu např. přes webový prohlížeč jsou nám kromě námi chtěných dat prezentována i data, která nevyhledáváme - reklamy. Reklamy jsou zásadním zdrojem příjmů pro provozovatele webových stránek. Pro ně platí jednoduchá přímá úměra mezi počtem reklam na stránce a ziskem. Toto ovšem vede k nekomfortu uživatele, jelikož reklamy nejenže zabírají podstatnou část obrazovky, ale dokáží i značně ovlivnit výkon jeho zařízení. I společnost *Google*, která je jedním z největších hráčů na trhu s internetovými reklamami, ve svém prohlížeči *Google Chrome* (resp. v jeho jádře *Chromium*) zavedla funkce, které reklamu zablokují při nadměrném vytěžování prostředků [2, 3].

V roce 2015 vznikla za účelem blokování nechtěné reklamy platforma *Pi-hole* [4]. Blokování reklamy funguje na principu *DNS (Domain Name System) sinkhole* - instanci *Pi-hole* nastavíme jako DNS server, a pokud na instanci přijde DNS dotaz na server, který distribuuje reklamy, *Pi-hole* vrátí *IP (Internet Protocol)* adresu *0.0.0.0* - což znemožní načtení reklamy.

Pi-hole je možno ovládat přes *CLI (Command-line interface)* rozhraní či pomocí webového rozhraní. Cílem této práce je navrhnout a implementovat třetí možnost. Ovládání pomocí *doplňku (plugin)* v prohlížeči, který urychlí a usnadní základní ovládání *Pi-hole*.

1.2 Motivace

Existence nástrojů k blokování reklamy je samozřejmě provozovatelům stránek známa. S blokováním reklam tedy bojují různými způsoby. Ať již častou změnou DNS adres, ze kterých se reklamy distribuují, tak detekcí blokace reklam.[1] Proto je nutné mít snadný přístup k administraci *Pi-hole*, jak

[1] Viz diskuze k detekci blokace reklam na stránce Stack Overflow `https://stackoverflow.com/questions/4869154/how-to-detect-adblock-on-my-website` [cit. 2020-11-29], či samotná existence knihovny na detekci blokace reklam `https://www.npmjs.com/package/just-detect-adblock` [cit. 2020-11-29].

k přidávání nových záznamů do *blacklistu*, tak k dočasnému pozastavení blokování reklam.

Pokud si vezmeme do úvahy znalosti průměrného uživatele internetu, současný přístup k ovládání *Pi-hole* není ideální. Možnost ovládání pomocí *CLI* je často nepohodlná a zbytečně zdlouhavá i pro administrátora sítě (nutnost fyzického přístupu k zařízení, kde *Pi-hole* běží, či nutnost navazání např. *SSH spojení*). Webové rozhraní je sice uživatelsky přívětivější, ale obyčejného uživatele zahltí možnostmi a různými ukazateli. Navíc si uživatel musí pamatovat *IP adresu* serveru, či ji mít někde uloženou.

Pokud obyčejnému uživateli neposkytneme rychlé a jednoduché ovládání, uslyší administrátor sítě jen negativní komentáře o problémech s nedostupnými servery a nefungujícími stránkami.

Zde vchází do hry možnost ovládání přes doplněk v prohlížeči. Mezi jeho hlavní výhody je možné zařadit :

- Rychlý přístup do ovládání pomocí jednoho kliknutí, bez potřeby opustit současnou stránku.
- Základní množina nejdůležitějších příkazů pro snadnou orientaci.
- Jednoduchá instalace pro administrátora sítě pomocí platformy *Docker* či spuštěním jednoho instalačního skriptu.
- Jednoduchá instalace pro uživatele přes obchod s doplňky do prohlížeče.

Kapitola 2
Technický slovníček

Tato práce využívá velké množství různorodých technologií a principů, z tohoto důvodu je zde zařazena kapitola s jejich popisem a vysvětlením.

2.1 Pi-hole

Jak už bylo uvedeno v úvodní kapitole, *Pi-hole* je prostředek k blokování reklam. Reklamy blokuje tak, že nahrazuje lokální *DNS server*, a pokud přijde *DNS dotaz* na doménu, která distribuuje reklamy, tak odpoví adresou *0.0.0.0*.

Pi-hole je nejčastěji používán na počítačích *Raspberry Pi* (2.2) (odtud také pochází název **Pi**-hole), ale nic nebrání instalaci *Pi-hole* na libovolném počítači[1] s dostatečným výpočetním výkonem a síťovým připojením.

2.1.1 Moduly Pi-hole

Pi-hole se skládá ze tří modulů, každý poskytuje jinou funkčnost. Tyto tři moduly je možné vidět v schématickém diagramu 2.1.

Pi-hole FTLDNS

Pi-hole FTL(DNS) je klíčovým modulem, který, jak název napovídá, obstarává samotné funkce DNS serveru, a blokování nežádoucích domén. Pro uživatele pomocí *API (Application Programming Interface)* nabízí komplexní statistiky o fungování instance jako počty dotazů a blokovaných domén, nejčastěji blokované domény a více.

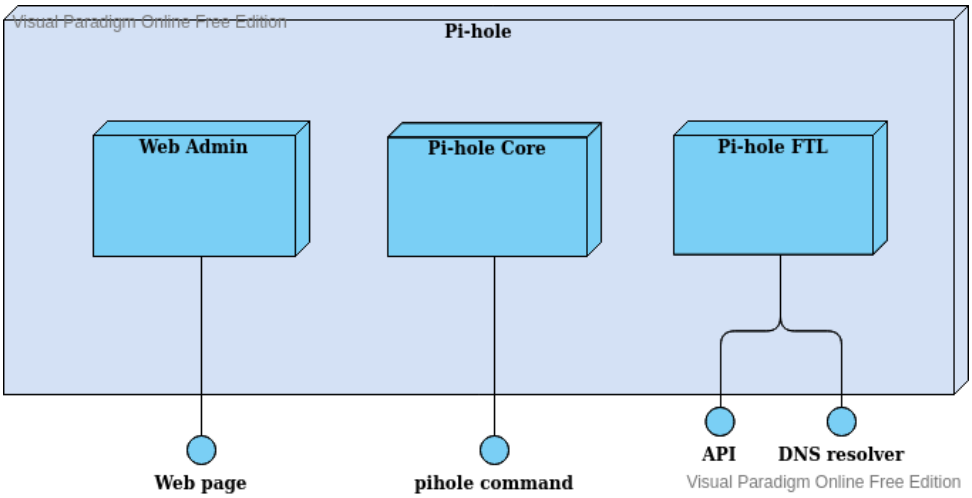
Pi-hole core

Pi-hole core je modulem, který poskytuje *pihole command* (3.1), tudíž je spíše podpůrným modulem pro *Pi-hole FTLDNS*. Obstarává logování, nástroj *gravity* (2.1.4), verzování a slouží jako middleware pro ovládání *Pi-hole FTLDNS*

[1] Minimální systémové požadavky a podporované operační systémy jsou k nalezení zde: `https://docs.pi-hole.net/main/prerequisites/` [cit. 2021-01-06].

Pi-hole Web Admin

Web Admin poskytuje uživatelské rozhraní pro ovládání Pi-hole přes prohlížeč, více informací v sekci 3.2.8.



Obrázek 2.1: *Pi-hole* moduly

2.1.2 Blacklist

Blacklist je seznam blokovaných domén. *Pi-hole* DNS server bude na DNS dotazy na domény z *blacklistu* odpovídat adresou *0.0.0.0*[2] .

2.1.3 Whitelist

Je možné, že se na *adlistu* (2.1.4), který odebíráme, vyskytne doména, kterou blokovat nechceme. I pro tyto situace existuje *whitelist*, seznam domén, u kterých DNS dotazy nebudou blokovány.

2.1.4 Gravity

Gravity je jedním z klíčových nástrojů Pi-hole. *Gravity* umožňuje automatické přidávání domén do *blacklistu* (2.1.2).

Gravity načte všechny seznamy reklam (2.1.4) z *Adlist Table* (2.1.5), stáhne jejich obsah, jednotlivé domény setřídí, odstraní duplicity a uloží je do *Gravity Table* (2.1.5).

Gravity poté provede restart DNS serveru (3.2.5), a ten si načte domény z *Gravity Table* (2.1.5) do *blacklistu*.

[2]Zde se jedná o jemnou nepřesnost v zájmu jednoduchosti. U *Pi-hole* lze nastavit několik režimů blokace - odpovídání adresou 0.0.0.0, odpovídání adresou, na které se Pi-hole nachází, či prázdnou odpovědí. Odpovídání adresou 0.0.0.0 je výchozí nastavení. Všechny režimy jsou popsány v dokumentaci, konkrétně zde: `https://docs.pi-hole.net/ftldns/blockingmode/` [cit. 2021-05-04]

Adlist

Adlist je volně dostupný seznam domén k zablokování ať již pomocí Pi-hole, či jiného adblock programu.

Doména se na *adlist* může dostat z několika důvodů, kromě domén, které rozesílají[3] reklamy, může být do seznamu umístěna doména, která například uživatele na internetu sleduje[4], či doména, která obsahuje malware či jiný nebezpečný obsah.

Velký problém tohoto konceptu je ten, že spoléháme na třetí osobu/y a jejich uvážení. Veřejné *adlist* mohou tedy často obsahovat domény, které jsou mimo jiné tzv. *false positive* nebo jsou již nepoužívané. Pesimisté mohou dokonce uvažovat o existenci seznamů, jejichž úkolem je škodit (např. blokace domény pro aktualizaci antivirových programů). Bohužel, či spíše naštěstí, se mi nepovedlo dohledat žádnou zmínku o takovémto škodlivém seznamu[5].

2.1.5 Domain database

Domain Database je základní databáze Pi-hole skládající se z několika tabulek (*tables*).

Mezi nejdůležitější tabulky patří : **Domainlist , Adlist Table, Gravity Table a Client Table.**

Domainlist

Tabulka obsahující všechny domény z *white/blacklistu* (2.1.3, 2.1.2). Záznamy v tabulce jsou upravovány pomocí nástroje *Gravity* 2.1.4, či pomocí příkazu `pihole` (3.2.1).

Tabulka kromě samotných domén, které mohou být zadány i v *RegEx*(*Regular Expression*) tvaru, obsahuje například jejich typ (*Whitelist/ blacklist, RegEx*), datum vytvoření a poslední datum úpravy.

Adlist Table

Tabulka obsahující množinu URL adres *Adlist* seznamů (2.1.4), ze kterých *Gravity* (2.1.4) čerpá domény k přidání do *Domainlistu* (2.1.5).

Gravity Table

Tabulka obsahující domény, které byly zkompletovány nástrojem *Gravity* (2.1.4).

[3]Pojem "rozesílají"je zde možná použit příliš expresivně, jsou tím myšleny domény, které jsou součástí nějaké *Content Delivery Network* (*CDN*), v tomto případě *CDN* pro reklamu

[4]Je myšleno sledování pomocí tzv. Tracking cookies (Third party cookies), více informací například v tomto článku: `privacy.net/stop-cookies-tracking/` [cit. 2020-11-19].

[5]Podařilo se mi ale nalézt příspěvěk o objevení se neškodné domény **files.slack.com** na jednom z populárních adlistů. Toto zabránilo uživatelům komunikační platformy *Slack* v jejím používání. Příspěvěk lze nalézt zde : `https://www.reddit.com/r/pihole/comments/lc2pah/if_youve_been_having_issues_with_images_etc_on/` [cit. 2021-05-04]

Při každém spuštění nástroje *Gravity* (3.2.4) je tato tabulka zahozena a vybudována znovu.

Primární funkcí této tabulky je možnost jednoznačného určení, ze kterých *adlist* seznamů (2.1.4) zablokovaná doména pochází.

Client Table

Tabulka obsahující množinu klientů DNS serveru (jejich IP adres).

2.2 Raspberry Pi

Raspberry Pi Foundation vznikla v roce 2009 a po dvou letech vývoje uvedla do prodeje svůj primární produkt - počítač *Raspberry Pi*. *Raspberry Pi* byl původně určen k propagaci počítačových věd do škol, postupem času jich ale bylo prodáno přes 20 milionů. Díky nízké ceně a multifunkčnosti zaujal pevné místo ve vzdělávacích institucích, ale i mezi nadšenci do technologií [5].

Raspberry Pi je plnohodnotný počítač přibližně o velikosti platební karty (*Raspberry Pi model B* má rozměry 85.60mm x 56mm x 21mm). Kromě klasických komponent jako procesor, GPU, RAM, obsahuje i např. GPIO headers pro ovládání periferií.

Raspberry Pi ale není jen počítač, je to i rozsáhlý soubor hardware komponent, softwaru a aktivní komunity.

2.3 WebExtensions API

WebExtensions API je prostředí pro vývoj a fungování doplňků v prohlížeči Mozilla Firefox, které má za úkol sjednotit vývoj doplňků do prohlížeče.

Tohoto cíle dosahuje používáním stejných jmenných konvencí a poskytováním stejné funkčnosti jako *Extensions API* od *Google Chrome*.

2.3.1 Match pattern

Match pattern v *WebExtensions API* prostředí je způsob, kterým můžeme jednoznačně identifikovat skupiny *URL (Uniform Resource Locator)*. Můžeme si jej představit jako zjednodušený *regulární výraz*, který je aplikován na *URL*.[6]

Vezmeme-li například *match pattern* `https://*/api/`, budou vyhovovat URL jako `https://fel.cvut.cz/api/` či `https://google.com/api`. Nevyhovovat budou naopak `http://fel.cvut.cz/api` či `http://fel.cvut.cz/`.

Match pattern se používá například u modulu doplňku *content script* (4.4.3), kde určuje množinu stránek, na kterých bude *content script* aktivní.

[6] Více informací a příkladů zde: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/match_patterns` [cit. 2020-11-19].

2.4 Google Chrome vs Chromium

Chromium je plnohodnotný open source prohlížeč, který je zamýšlený jako společné jádro pro různé prohlížeče. Toto jádro využívá například *Google Chrome*, *Opera* či *Microsoft Edge*.

Goole Chrome tedy využívá toto jádro a rozšiřuje ho o různé funkce a napojuje jej více do svého ekosystému (*Chromium* samotné je zapojeno do Google ekosystému, například přihlašování do prohlížeče probíhá přes Google účet, či sdílí obchod s doplňky)[6].

2.5 Web-ext

Web-ext je nástroj spravovaný Mozilla project, který má za úkol zjednodušit vývoj, testování a nasazování doplňků.[7] Nástroj umožňuje několik zajímavých vývojových funkcí, jako znovu-načtení doplňku v prohlížeči, když se změní jeho zdrojový kód (*hot reload*), nebo analýzu kódu.

Pomocí *web-ext* můžeme také doplněk zabalit, podepsat a odeslat do **addons.mozilla.org**, kde můžeme doplněk zveřejnit uživatelům pro nainstalování (Ať již jako nový doplněk, či jen novou verzi zveřejněného doplňku).

2.6 GraalVM

GraalVM je *JavaVM* s vlastním *Java SDK (Software development kit)*, který podporuje i jiné programovací jazyky. Hlavní předností, kterou *GraalVM* oproti *Java VM* nabízí, je *Ahead-of-Time compilation*, což je přeložení byte kódu do strojového kódu před distribucí spustitelného souboru. To znamená, že Java aplikace může běžet bez *JavaVM* [7]. *GraalVM* je zatím v raném stádiu svého vývoje, s čímž je potřeba při volbě této technologie počítat.

2.6.1 Native image

Aplikace přeložená pomocí *Ahead-of-Time compilation* se nazývá *native image*. *Native image* není, narozdíl od *Java VM*, nezávislý na platformě[8]. To znamená, že pro každou platformu je potřeba vygenerovat zvláštní *native image*.

Quarkus a native image

Quarkus (*Framework* pro programovací jazyk *Java*, více v oddílu 5.3) podporuje velmi jednoduchou kompilaci do *native image* pomocí *GraalVM*. Pokud chceme vytvořit *native image*, například pomocí systému *Maven* (2.9), stačí jeden příkaz `./mvnw package -Pnative`. Tento příkaz nám vygeneruje v

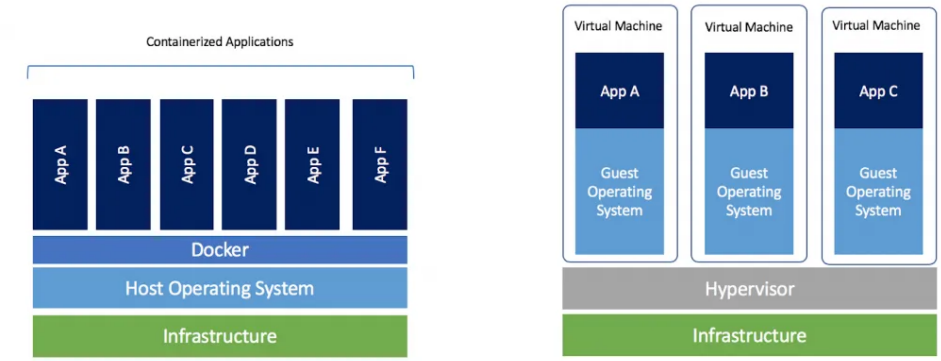
[7] Dostupný z `https://github.com/mozilla/web-ext` [cit. 2020-11-10].

[8] Slovo platforma (*platform*) má v informatice mnoho definic a podob. Proto v této práci bude slovo platforma označovat dvojici instrukční sady a operačního systému - například *AMD64/linux*.

složce `target` spustitelný binární soubor. Velkou výhodou je, že kompilace probíhá v *Docker kontejneru* pomocí speciálního *Docker image* (2.7.1), tudíž jedinou závislostí je *Docker* a *Maven*, bez potřeby instalovat *GraalVM*.

2.7 Docker

Docker je nástroj na virtualizaci na úrovni operačního systému. Funguje na podobném principu jako *virtuální počítače* (*VM*), s rozdílem, že jednotlivé instance sdílí *jádro* (*kernel*), tudíž jsou méně náročné na provoz (schématický rozdíl viz schéma 2.2).



Obrázek 2.2: Zjednodušené schéma rozdílů mezi *Docker* a *VM* [8]

2.7.1 Docker image

Docker image (*obraz*) je vzor, ze kterého lze vytvářet jednotlivé *Docker containers*. Zjednodušeně můžeme přirovnat *Docker image* ke třídě v objektově orientovaném přístupu.

2.7.2 Docker container

Docker container (*kontejner*) je konkrétní instance, která při stvoření vychází z *Docker image*. *Docker container* můžeme přirovnat k konkrétní instanci třídy.

2.7.3 Multiarch Docker image

I když jedním ze základních Docker principů je princip *Build once, Run anywhere (Sestav jednou, spusť kdekoliv)*, tento princip naráží, pokud chceme jeden obraz používat na různých architekturách, protože Docker obraz téměř vždy obsahuje zkompilované binární soubory pro specifickou architekturu. Také naráží i na různých operačních systémech, jelikož Docker využívá části operačního systému. Takovýto Docker obraz budeme v této práci označovat jako *platform-specific image* či *obraz závislý na platformě*.

Jako řešení existuje *Docker multiarch image (obraz)*. *Multiarch* obraz se v podstatě skládá z několika *platform-specific* obrazů spojených jedním *Docker manifestem* (2.7.5). Pokud chceme vytvořit kontejner z takovéhoto *multiarch* obrazu, Docker automaticky vybere a spustí vhodný *platform-specific* obraz podle aktuálního prostředí.

2.7.4 Tvorba multiarch obrazu

Existují dva hlavní způsoby tvorby *multiarch* obrazu [9].

Docker buildx

Buildx je Docker plugin, který rozšiřuje základní funkcionalitu nástroje Docker CLI rozhraní, mimo jiné podporuje jednoduchou tvorbu *multiarch* obrazů[9].

Tvorba *multiarch* obrazu je možna jedním příkazem

```
docker buildx build --platform <seznam-platforem> .
```

Buildx paralelně vytvoří všechny *platform-specific* obrazy, které pak spojí pod jeden *multiarch* obraz.

Docker manifest

Tvorba přes *manifest* je původním způsobem tvorby *multiarch* obrazů. Nejdříve musíme vytvořit jednotlivé *platform-specific* obrazy. U nich jejich cílovou platformu určíme přepínačem `--build-arg` (např. `--build-arg ARCH=amd64`).

Z takto vytvořených obrazů poté vytvoříme *multiarch* obraz příkazem

```
docker manifest create <jmeno vysledneho obrazu>
                       --amend
                  <jméno platform-specific obrazu>
                       --amend
              <jméno jiného platform-specific obrazu>.
```

Každý *platform-specific* obraz tedy připojíme přepínačem `--amend`.

Tato metoda využívá toho, co *multiarch* obraz je - jen soubor několika *platform-specific* obrazů. Oproti *Buildx* metodě má jasnou nevýhodu ve složitosti. Ale její hlavní výhoda, kvůli které je používána v této práci, je možnost pro každý obraz použít vlastní *Dockerfile* (viz 2.7.6).

2.7.5 Docker manifest

Docker manifest je *JSON (JavaScript Object Notation)* soubor, který obsahuje metadata o jednom Docker obrazu. U *platform-specific* obrazů obsahuje například informace o jednotlivých vrstvách jako *digest* vrstvy (digest je *hash* vrstvy dle *Docker v2* formátu) či velikost vrstvy (příklad platform-specific manifestu C.0.2). Pokud se jedná o *multiarch* obraz, obsahuje seznam

[9] *Docker buildx* se bohužel nachází v rané části vývoje, tudíž není zaručena jeho stabilita a způsoby jeho použití se mohou bez varování změnit.

manifestů jednotlivých *platform-specific* obrazů, ze kterých se skládá a metadat o nich (příklad multiarch Docker manifestu C.0.3).

2.7.6 Dockerfile

Dockerfile je předpis, podle kterého se vytváří jednotlivé Docker obrazy. Jedná se o soubor instrukcí potřebných k vytvoření Docker obrazu.

Docker jako takový funguje na principu vrstev. Každá instrukce v *Dockerfile* je vlastně jedna vrstva, která se nabalí na tu předchozí. Zároveň se každá vrstva ukládá, aby se mohla případně přepoužít (např. pokud změníme jen poslední vrstvu, nebude se celý obraz tvořit znovu, ale využije se již uložených vrstev.)

Příklad souboru Dockerfile vytvořeného v rámci projektu se nachází v příloze C.0.4.

2.7.7 Pojmenovávání Docker obrazu

Každý Docker obraz musí mít unikátní jméno ve tvaru

```
<registry>/<repository-name>/<name>:<tag>
```

Jednotlivé části jména mají tyto významy :

- **registry** - URL adresa systému, který ukládá a spravuje jednotlivé obrazy. Pokud není specifikováno, je použita výchozí adresa `registry-1.docker.io`.
- **repository name** - jméno repozitáře, ve kterém je obraz uložen.
- **name** - jméno obrazu.
- **tag** - tag se používá odlišení různých obrazů se stejným jménem. Jeho hlavní použití je k odlišení různých verzí jednoho obrazu (např. *v1.0,v1.1, latest* apod.).

Například obraz s nejnovější verzí serveru této práce má jméno **kulda22/alanine:latest**.

2.8 QEMU

QEMU je emulátor a virtualizátor, který umožňuje spouštět programy na jiných plaftormách, než pro které byly kompilovány.

QEMU má dva hlavní módy :

- Systémová emulace - je emulován celý operační systém, bez ohledu na jakou architekturu byl kompilován.
- Programová emulace - *QEMU* umožňuje spouštět jednotlivé programy, které byly kompilovány pro jiné platformy.

Druhého módu využíváme v této práci v sekci 7.2.3. Docker kontejner zaregistruje sám sebe v službě *binfmt_misc*[10]. *Binfmt_misc* je služba, pomocí které v Linux jádru specifikujeme, jakou aplikací se má spustit konkrétní formát spustitelných souborů. V našem případě je jako aplikace registrován Docker kontejner, ve kterém je nainstalován *QEMU*. Pokud tedy spustíme program kompilovaný pro jinou platformu, spustí se v kontejneru. V našem případě spouštíme kontejner pro jinou platformu, tedy de facto spouštíme kontejner v kontejneru.

2.9 Maven

Maven je nástroj, který má na starosti dvě stránky vývoje software - závislosti (*dependencies*) a sestavení (*build*). Je primárně určen pro programovací jazyk Java, ale je kompatibilní s více jazyky (např. C#, Scala, Ruby).

Maven je založen na principu životního cyklu aplikace. Životní cyklus aplikace se skládá z několika fází (*compile, test, package ...*). Každou tuto fázi můžeme ovlivnit pomocí doplňků (*plugin*), které mění / doplňují standardní průběh.

2.9.1 Maven goal

Maven goal je konkrétní cíl doplňku. Slouží, abychom mohli spustit nějaký *Maven* příkaz i mimo fáze životního cyklu aplikace (nebo jej spustit spolu s nimi).

2.10 CI/CD

Pojem *CI/CD* se skládá ze dvou pojmů - *CI* jako *Continuous Integration* a *CD* jako *Continuous Delivery*. Tyto dva pojmy spoluvytváří myšlenkový postup, který preferuje krátké vývojové cykly a časté vydávání nových verzí, které přináší malé změny, před občasným vydáváním verzí, které přináší mnoho změn najednou. Prim v *CI/CD* hraje automatizace, automatizuje se testování, automatizuje se tvoření spustitelných souborů (*build*) a nasazování [10] .

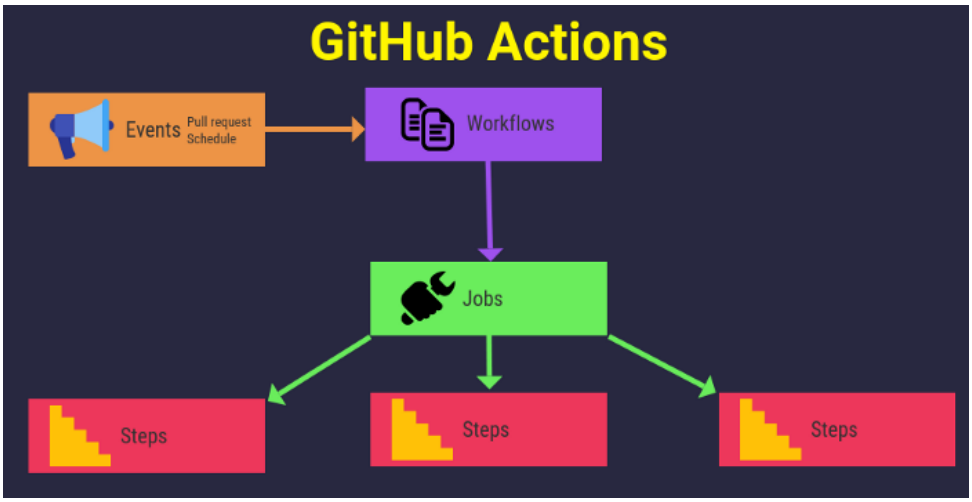
Zjednodušeně by se dalo říci, že ideálním průběhem podle *CI/CD* principů by bylo udělat změnu v kódu, a poté ji zanést do verzovacího systému. Ten změnu otestuje, vytvoří spustitelný soubor a ten nasadí.

2.10.1 GitHub Actions

GitHub Actions je nástroj komerčního verzovacího systému *GitHub*. Tento nástroj umožňuje spouštění různých *scénářů (workflows)* pomocí posluchačů na události v repozitáři (např. *push*, vytvoření *issue*).

[10] Více informací o celém procesu zde: `https://github.com/multiarch/qemu-user-static`[cit. 2021-05-03]

Každý scénář se skládá z několika *úkolů (jobs)*, které se skládají z postupných kroků. Schéma je možné vidět na diagramu 2.3.



Obrázek 2.3: Zjednodušené schéma anatomie GitHub Actions [11]

Silnou stránkou *GitHub Actions* je možnost využití *Actions* - to jsou samostatné přepoužitelné bloky napsané třetí stranou, které využíváme v úkolech. Tyto bloky mají za úkol zjednodušit vytváření scénářů. Slouží například k přípravě prostředí, volání *API* různých nástrojů apod.

Každý scénář je definován speciálním souborem v repozitáři. Obsah takového souboru je možno vidět v úryvku 2.1.

```
# When the workflow should start
on:
  push:
    branches: master

    ...

jobs:
    build:
    # Defines basic enviroment
    runs-on: ubuntu-latest
    # List of steps to execute
    steps:
          # Get the repositery's code - using action
      - name: Checkout
        uses: actions/checkout@v2
      - name: Set up JDK 11
        uses: actions/setup-java@v1
        # Action's arguments
        with:
          java-version: 11
    # Run executes command in bash
      - name: Build with Maven
        run: mvn -Dquarkus.package.type=fast-jar package
```

Listing 2.1: Úryvek ze souboru který definuje *GitHub Actions scénář*

2.11 BASH Injection

BASH Injection je bezpečnostní slabina podobná *SQL injection*. V podstatě jde o zranitelnost, která útočníkovi umožní spouštět vlastní *BASH* příkazy. Jednoduchá ukázka tohoto útoku je vidět v 2.2. Samozřejmě, ukázaný útok není nebezpečný, ale místo `ls -la` můžeme vykonat například `bash -c "cat ~/.ssh/id_rsa &>/dev/tcp/<ip>/<port>"`, což například odešle *privátní RSA klíč* na útočníkův server.

```
/// intended function
String userInput = "cvut.cz"


execute("pihole -b " + userInput);
/// Pi-hole blacklists domain cvut.cz - OK


/// BASH Injection

/// ";" is command delimiter in BASH
String maliciousInput = "cvut.cz; ls -ls"

execute("pihole -b " + userInput);
/// Pi-hole blacklists domain cvut.cz AND lists all files and folders in directory
```

Listing 2.2: Ukázka jednoduchého útoku pomocí *BASH injection*

Kapitola 3
Administrace

Jak již bylo řečeno v úvodu, Pi-hole nabízí dvě rozhraní pro administraci. Tato rozhraní nabízejí možnost komplexní administrace *Pi-hole*, a pro potřeby této práce se zaměříme na popis *CLI rozhraní* a jeho schopností.

3.1 Popis CLI rozhraní

CLI rozhraní *Pi-hole* je realizováno pomocí příkazu `pihole [COMMAND]`. Pokud máme nainstalovanou *Pi-hole* pomocí Docker platformy, je nejjednodušším způsobem vykonat příkaz `docker exec [container name] pihole [COMMAND]`, kterým Docker vykoná uvedený příkaz v uvedeném kontejneru.

Přes CLI rozhraní voláme přímo *Pi-hole core* (2.1.1), pokud chceme administrovat *web admin* rozhraní (3.2.8), musíme přidat přepínač `-a` za příkaz `pihole`.

CLI rozhraní je svým rozsahem příkazů určeno spíše k uzpůsobování fungování *Pi-hole*, pro dohled nad fungováním *Pi-hole* je vhodnější webové rozhraní (3.2.8).

Tento i následující odstavce vychází z oficiální dokumentace Pi-hole[1].

3.2 Příkazy CLI rozhraní

3.2.1 Přídání záznamů do Whitelistu a Blacklistu

Základním příkazem pro Pi-hole je přidávání domén do *black-/whitelistu* (2.1.2,2.1.3).

Pokud chceme přidat doménu v *RegEx (Regular expression)* tvaru, je potřeba použít přepínač `-regex`.

Pokud nezadáváme doménu v *RegEx* tvaru a chceme zablokovat i všechny poddomény, použijeme přepínač `--wild`, Pi-hole doménu přemění v *RegEx* výraz, který bude obsahovat všechny poddomény.

Po každém přidání záznamu proběhne reload DNS serveru - DNS server se restartuje a znovu se načtou blokované domény spolu s přidaným záznamem.

[1] Dostupná z `https://docs.pi-hole.net` [cit. 2020-11-01]

Blacklisting

Příkazem ve tvaru `pihole -b [domain name]` přidáme adresu na *blacklist*.

Whitelisting

Příkaz ve tvaru `pihole -w [domain name]` použijeme, pokud chceme přidat doménu na *whitelist*.

3.2.2 Enable & Disable DNS blocking

K pozastavení blokování domén na daný časový úsek můžeme použít příkaz ve tvaru `pihole disable [time]`. Čas volíme v sekundách nebo minutách.

Přesný formát příkazu je `pihole disable ?m`, kde '?' je počet minut na pozastavení, respektive `pihole disable ?s`, kde '?' je počet sekund na pozastavení.

Pozastavit můžeme i na dobu neurčitou příkazem `pihole disable`. Příkazem `pihole enable` blokování domén zase obnovíme.

3.2.3 Enable & Disable logging

Pi-hole ve výchozím nastavení loguje všechny *DNS* dotazy, nad kterými pak vypracovává statistiky dostupné např. přes webové rozhraní (3.2.8).

Pokud chceme logování zastavit, použijeme příkaz `pihole logging off`. Toto ale zároveň smaže i historii logů, takže tímto příkazem ztratíme veškeré statistiky.

Pokud chceme zabránit ztrátě historie DNS dotazů, můžeme použít příkaz `pihole logging off noflush`, který pozastaví logování, ale zanechá jejich historii.

Logování obnovíme příkazem `pihole logging on`.

3.2.4 Gravity update

Po instalaci *Pi-hole* je zaregistrována služba (*cron*), která mimojiné každý týden v neděli (mezi 03:00 a 05:00) provede *Gravity update* (2.1.4) pomocí příkazu `pihole updateGravity`.[2]

Pokud ale chceme spustit update manuálně, můžeme použít `pihole -g`, nebo `pihole updateGravity`.

3.2.5 Restart DNS

Pomocí příkazu `pihole restartdns` je možné provést restart *Pi-hole* DNS serveru.

DNS server si po restartu znovu načte všechna nastavení a blokované domény.

[2] Diskuze k tomuto tématu na stránkách Pi-hole fóra `https://discourse.pi-hole.net/t/gravity-update-when-does-it-run-exactly/20098/2` [cit. 2020-11-1].

3.2.6 Pihole version

Verzi *Pi-hole* zobrazíme příkazem `pihole -v`. Dostaneme takovýto výstup :

```
pi@alanin:~ $ pihole -v

Pi-hole version is v5.1.2 (Latest: v5.1.2)
AdminLTE version is v5.1.1 (Latest: v5.1.1)
FTL version is v5.2 (Latest: v5.2)
```

3.2.7 Status

Pi-hole disponuje jednoduchým příkazem na zjištění stavu. Pomocí příkazu `pihole status` *Pi-hole* zobrazí, jestli běží DNS service, a jestli je DNS blokování zapnuto.

3.2.8 Webové rozhraní

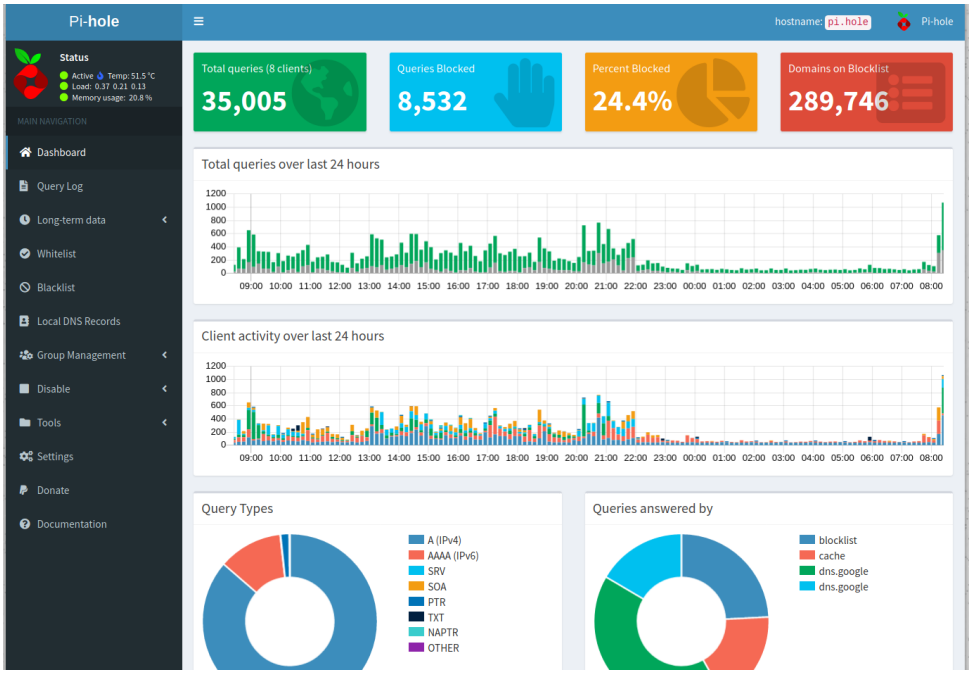
Webové rozhraní (Web Admin) umožňuje snadný vhled do fungování Pi-hole statistikami jako počty DNS dotazů za časový úsek či přehled nejčastěji blokovaných domén.[3]

Kromě sledování statistik je možná administrace *Pi-hole* či vyhledávání nad *Domain database* (2.1.5).

Množina funkcí *webového rozhraní* pokrývá většinu funkcí CLI rozhraní a poskytuje některé funkce, které CLI rozhraní neumožňuje. Například audit DNS adres - možnost označení DNS adresy, které naznačuje, že jsme u ní provedli audit a nenalezli jsme důvod k jejímu zablokování.

Na obrázku 3.1 je možno vidět UI rozhraní.

[3] Webové rozhraní poskytuje balíček `https://github.com/pi-hole/AdminLTE` [cit. 2020-11-03].



Obrázek 3.1: Pihole web admin rozhraní

Kapitola 4
Tvorba doplňků

Tato kapitola se zaměřuje na návrh a vývoj doplňku do prohlížečů a na obecné principy jejich funkčnosti. Ve vývoji a používání doplňků je vidět neustálý konflikt mezi expresivitou doplňků, bezpečím a soukromím jejich uživatelů.

4.1 Úvod

Přibližně před šesti lety měl programátor na výběr z několika způsobů tvorby doplňků do prohlížeče. Mohl volit mezi *Overlay extension*, *Bootstrapped* či *Add-on SDK*, který se nejvíce podobá současnému přistupu [12].

Všechny postupy výše již nejsou v dominantních prohlížečích podporovány (např. u *Mozilla Firefox* nejsou podporovány od verze 57, která vyšla na konci roku 2017 [12].)

V současné době existuje u každého z dominantních prohlížečů na trhu jen jediný přístup k tvorbě doplňků - *WebExtensions API* v případě *Mozilla Firefox* a *Extension API* v případě *Google Chrome*[1].

Pro potřeby této práce budeme používat názvosloví a principy tvorby doplňků pomocí *WebExtensions API* pro prohlížeč *Mozilla Firefox*.

Následující odstavce vycházejí z internetové dokumentace MDN web docs[2].

4.2 Princip

Vývoj doplňků probíhá pomocí technologií *Javascript*, *CSS (Cascading Style Sheets)* a *HTML (HyperText Markup Language)*. Volba těchto technologií není náhodná, jelikož samotný obsah webových stránek je tvořen stejnými technologiemi, a doplněk často tento obsah upravuje či rozšiřuje.

Každý doplněk je tvořen několika moduly, které spolu komunikují pomocí API prohlížeče. Tyto moduly reagují na události uživatele a prohlížeče.

[1] Dokumentace dostupná na : `https://developer.chrome.com/extensions/api_index` [cit. 2020-11-10].

[2] Dostupné z `https://developer.mozilla.org` [cit. 2020-11-10].

4.3 Manifest

Nejdůležitější součástí doplňku je soubor *manifest.json*. Je to jediný soubor, který se musí nacházet v každém doplňku.

Manifest definuje základní metadata jako název, popis, jméno vývojáře či verzi. Kromě metadat obsahuje např. i množinu povolení, která vyžaduje od uživatele či odkazy na ikony.

Další důležitou součástí je definování součástí doplňku, jako například seznam *background scripts* či HTML soubor, který má být vykreslován jako modul *options page*.

Příklad *manifestu* přímo z *Mozilla Web Docs* se nachází v příloze C.0.1.

4.4 Moduly doplňku

4.4.1 Toolbar button

Toolbar button je předem definovaná akce, která nastane po kliknutí na ikonu doplňku v nástrojové liště napravo od vyhledávací řádky prohlížeče. Příklad nástrojové lišty a různých ikon *toolbar button* je možno vidět na obrázku 4.1.

Obrázek 4.1: Nástrojová lišta prohlížeče Mozilla Firefox

Po kliknutí na *toolbar button* mohou nastat dva scénáře.

Browser Action

Prohlížeč vytvoří událost, na kterou může být navázán posluchač. Posluchač může provést definovanou akci (např. otevření nové záložky či změnu obsahu stránky).

Popup

Prohlížeč vykreslí dodaný HTML soubor, se kterým může uživatel interagovat. Do HTML souboru může být vložen i Javascriptový soubor, který zajišťuje komunikaci s ostatními moduly. Pro potřeby návrhu je důležité chování prohlížeče, který tento javascriptový skript vykonává jen po dobu, po kterou je *popup* otevřený, jakmile jej uživatel uzavře, prohlížeč skript přestane okamžitě vykonávat.

4.4.2 Background script

Background script je množina Javacsriptových skriptů, které běží v pozadí nezávisle na otevřených stránkách či oknech. Tyto skripty je možno používat k uchovávání stavu doplňku, či k vykonávání časově náročných akcí.

4.4.3 Content script

Jedním z nejdůležitějších modulů je *content script* modul. Zjednodušeně si jej můžeme představit jako Javascriptový skript, který je vložen do HTML každé stránky. Zde může mít stejnou funkčnost jako skript, který byl stažen s HTML stránky. Navíc má přístup k podmnožině WebExtensions API.

Samozřejmě realita je složitější a omezená. Ke každému skriptu je v manifestu nutno připojit i vlastnost *matches*, která pomocí *match patternu* (2.3.1) určuje, na kterých stránkách bude *content script* spuštěn.

Content script je navíc oddělen od skriptů webové stránky, největším omezením je nemožnost přímého přístupu k proměnným stránky. Toto omezení lze obejít, ale z hlediska bezpečnosti uživatele to není doporučováno [13].

4.4.4 Notifikace

K upozornění uživatele na např. provedenou akci či změnu stavu je možno využít *notifikace*. *Notifikaci* vytvoříme pomocí *WebExtension API*. Zajímavostí je, že tyto notifikace jsou tvořeny operačním systémem, na kterém je prohlížeč spuštěn, takže na každém operačním systému budou notifikace vypadat jinak.

4.4.5 Options page

Options page je HTML dokument, pomocí kterého uživatel může přizpůsobovat doplněk. Tento dokument může uživatel zobrazit v `about:addons`, či jej můžeme zobrazit z skriptu zavoláním funkce `runtime.openOptionsPage()`.

4.4.6 Ostatní

Kromě výše uvedených modulů existují mnohé další. Například modul upravující našeptávaní v adresové řádce či modul pro přidání položky do kontextového menu po kliknutí pravým tlačítkem na stránce.

4.5 Komunikace mezi moduly doplňku

Komunikace mezi moduly doplňku by měla probíhat pouze pomocí funkcí *WebExtensions API* k tomu určených. Jsou možné dva způsoby komunikace.

4.5.1 One-off messages

Jednodušším způsobem jsou takzvané *one-off messages*. Jsou to jednorázové zprávy, jenž obsahují Javacsriptový objekt. Je to jednoduchý způsob komunikace například mezi *background script* a *content script* a naopak. Vysílající strana volá funkci `browser.runtime.sendMessage(payload)`, přijímající strana musí mít zaregistrovaný posluchač `browser.runtime.onMessage.addListener(callback-function)`.

4.5.2 Connection-based messaging

Connection-based messaging je další způsob pro komunikaci mezi moduly. Po vytvoření spojení obě strany vlastní objekt, pomocí kterého čtou i posílají zprávy. Další výhodou je, že pokud jedna strana komunikuje s více stranami najednou (např. jeden *background script* obsluhuje několik *content script*), je snadno určitelné, od kterého skriptu zpráva přišla.

4.6 Uchovávání stavu

Uchování stavu přes vypnutí prohlížeče je řešeno pomocí *local storage* - rozhraní podobné *Web Storage API*, které využívá Javascript ve webových stránkách. Je využíváno například k přechovávání nastavení doplňku.

4.7 Permissions

Doplňky jsou v zásadě bezpečnostní slabinou. Stahujeme si do prohlížeče kód, který může číst i modifikovat jak samotný obsah stránek, tak požadavky a odpovědi při jejich komunikaci se serverem.

Proto je implementován systém povolení (*permissions*). Možnost přístupu k určitým metodám, které by byly snadno zneužitelné, je podmíněna souhlasem uživatele při instalaci (aktualizaci) doplňku. Úmysl přístupu k těmto metodám je nutno explicitně uvést pomocí záznamu *permissions* v manifestu.

4.8 Multiplatformnost

Multiplatformnost rozšíření je zásadním tématem pro vývojáře. Při pohledu na podíl prohlížečů na trhu vidíme jasné tři leadery trhu. *Chrome* a prohlížeče na bázi *Chromium* (např. *Edge, Opera*), *Safari* a *Mozilla Firefox* [14]. Pokud vynecháme *Safari*, zůstanou nám dva prohlížeče, pro které, když vyvineme dvě verze doplňku, pokryjeme více než 75 % trhu.

Autoři jak *Mozilla Firefox*, tak *Google Chrome* při vývoji svých prohlížečů mají na mysli multiplatformnost doplňků. I proto je WebExtensions API z velké části kompatibilní s Extensions API. Pokud si vývojář není jistý kompatibilitou, může navštívit MDN Web Docs[3], najít si danou funkci, kde se nachází tabulka kompatibility, a rozdíly mezi *WebExtensions API* a *Extensions API*.

V následujích odstavcích bych se rád zaměřil na pár vybraných rozdílů mezi prohlížeči.

[3]Dostupné z `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/` [cit. 2020-11-19]

Jmenné prostory

Pokud chceme volat funkce *Extensions API* na prohlížečích *Google Chrome*, *Chromium* či *Opera*, musíme použít jmenný prostor `chrome.<function>` (např. `chrome.runtime.sendMessage(message)`)

Prohlížeče *Mozilla Firefox* (pro *WebExtensions API*) či *Edge* (Pro *Extensions API*) zase používají `browser.<function>`.

Firefox pro usnadnění vývoje ale podporuje i `chrome.*` jmenný prostor, i když je doporučováno používat `browser.*`.

Asynchronní zpracovávání událostí

Podstatným rozdílem je i proces zpracování asynchronních událostí. *Chrome* a prohlížeče na bázi *Chromium* používají *callback*.

U *Firefox* je použití návrhového vzoru *callback* zastaralé. Standardním způsobem zpracování je použití návrhového vzoru *promise* [4].

Pro použití *promise* i u prohlížečů na bázi *Chromium* je možno použít například knihovnu *webextension-polyfill* (6.2.3).

Rozdíly mezi prohlížeči na bázi Chromium

Rozdíly jsou i mezi prohlížeči na bázi *Chromium*, často některé funkce přímo chybí (funkce *privacy* u *Edge*).

Safari

Safari od společnosti *Apple* jsem při úvahách a porovnáních výše vynechal, jelikož vývoj doplňků pro Safari je sice konceptuálně stejný, ale implementačně značně rozdílný. Proto doplněk vytvořený pro Safari rozhodně není multiplatformní. *Apple* ale nedávno umožnil vytvoření doplňku pro *Safari* z zdrojových kódů doplňku, který je určen pro jiný prohlížeč pomocí nástroje *Xcode* [15].

Z výše uvedených důvodů je zřejmé, že ačkoliv je jasná tendence vývojářů webových prohlížečů sjednotit vývoj doplňků, stále existují markantní rozdíly. Proto je při návrhu doplňku vhodné si jasně vymezit podporované prohlížeče, a při vývoji důkladně testovat na všech podporovaných prohlížečích.

Pro rozsah této práce jsem si zvolil za cíl vytvořit doplněk pro prohlížeče: *Mozilla Firefox*, *Google Chrome* a *Chromium*.

[4] *Promise* je návrhový vzor, který se používá u asynchronních operací. Každá asynchronní operace nám vrací *promise* (*slib*), který má metodu *then*. Do této metody vložíme funkci, která bude zavolána po úspěšném dokončení operace. Oproti vzoru *callback* je *promise* čitelnější při řetězení více asynchronních operací.

Kapitola 5
Návrh řešení

Po představení programu *Pi-hole*, popisu vývoje a technologií potřebných k vývoji doplňku webového prohlížeče, lze nyní přejít k vlastnímu návrhu řešení.

5.1 High level návrh

Začneme high-level návrhem. Uživatel bude mít ve svém preferovaném prohlížeči nainstalovaný doplněk. Doplněk bude moct ovládat např. přes popup, po zadání povelu doplněk pošle příkaz na server. Server příkaz zpracuje, pokud je to potřeba, změní stav lokálně běžící *Pi-hole*, a v odpovědi vrátí informaci o úspěšném či neúspěšném provedení příkazu. Doplněk tuto informaci zpracuje a zobrazí uživateli informaci o výsledku operace.

5.2 Funkce

Po prozkoumání možností administrace *Pi-hole* (3) a uživatelských zkušenostech jsem vymezil množinu funkcí, které by doplněk měl umožňovat.

- Vypnutí / zapnutí **DNS blokování** jak na určitý čas, tak bez časového omezení (3.2.2).
- Vypnutí / zapnutí **logování** jak na určitý čas, tak bez časového omezení ((3.2.3)).
- Přidání domény na **white/blacklist** (3.2.1).
- Spuštění **update gravity** (3.2.4).
- Spuštění **restartu DNS** serveru (3.2.5).
- Zobrazení **verze** Pi-hole (3.2.6).
- Zobrazení **stavu** Pi-hole (3.2.7)

5.3 Serverová část

Jako nejjednodušší volba se nabízí použít *REST API*, které poskytuje přímo *Pi-hole* (poskytuje jej modul *FTLDNS*, který je popsán v části 2.1.1), ale jeho rozsah je značně omezený, spíše slouží jako zdroj informací o stavu *Pi-hole*. Pokud bychom chtěli použít *Pi-hole REST API*, museli bychom poté funkce našeho doplňku uzpůsobit kapacitám tohoto *API*.

Druhou volbou je vytvoření vlastního serveru, se kterým bude doplněk komunikovat. Server by tyto požadavky zpracoval, a pomocí příkazu `pihole` by je předal *Pi-hole* na vykonání. Protože příkaz `pihole` je skutečně mocný v možnostech administrace *Pi-hole* (viz kapitola 3), mohli bychom implementovat všechny funkce. Jasnou nevýhodou je nutnost vytvoření tohoto serveru, a další vytížení prostředků na zařízení. Proto je třeba vybrat řešení, které zatíží zařízení co nejméně.
Jelikož chci implementovat všechny funkce, jedinou volbou je vytvoření vlastního serveru.
Na výběr je z mnoha technologií, zde porovnám pár z nich.

5.3.1 Javascript - Node.js

Node JS je *framework* pro Javascript pro psaní aplikací mimo prohlížeč. Interně je postaven na prostředí *V8 engine* od *The Chromium Project*.[1] Nejčastěji je používán pro serverové aplikace, kde na klientské straně běží Javascriptový *framework* (*React, Vue, Angular* či další). *Node JS* využívá pro všechny *IO* (*Input/Output*) operace asynchronní zpracování, tudíž např. otevření souboru neblokuje vlákno, které by jinak čekalo, než se soubor otevře. Velkou výhodou i nevýhodou zároveň je faktická nemožnost využít více vláken. *Node JS* bude vždy využívat jen jedno vlákno, což sice vede k nižšímu vytížení procesoru, ale dokáže vést k vyšším latencím při vyšším vytížení [16, 17].

Výhody

- Malá náročnost na výpočetní prostředky.
- Asynchronní zpracovávání *IO* operací.
- Javascript je snadný jazyk na naučení.
- Velký počet knihoven.
- Aktivní komunita.

Nevýhody

- *Node JS* je založen na *callback* architektuře. Často to může vést k nepřehlednému kódu.
- Při nárůstu požadavků vyšší latence.

[1] Více o *V8 engine* zde : `v8.dev/` [cit. 2020-11-22].

- Prudký vývoj - často se mění, zpětná kompatibilita knihoven není zaručena.

5.3.2 Go - Revel

Go (či *Golang*) je poměrně mladý jazyk[2], který spadá pod křídla společnosti *Google*. I když není na trhu tolik rozšířený, je mezi vývojáři velmi populární, v roce 2020 byl jazykem, který by se chtělo naučit největší procento vývojářů [18, 19]. *Revel* je *full-stack web framework* pro *Go*, který nabízí mnoho funkcí důležitých pro vývoj a fungování *REST API* (jako například *filtrování*, *routing*, *interceptors*) bez složitého nastavování.

Výhody

- Malá náročnost na výpočetní prostředky.
- Nízká latence i při vysokém počtu požadavků.
- Snadný jazyk na naučení.

Nevýhody

- Menší počet knihoven.
- *Garbage collector* dokáže značně zpomalit celou službu [20].

5.3.3 Java - Spring Boot

Java je jedním z nejrozšířenějších jazyků, používá se na komplexní systémy s velkým počtem požadavků. *Spring*[3] je *framework*, který se snaží poskytnout komplexní řešení pro vývoj různorodých aplikací. Nadstavba *Spring Boot* umožňuje rychlejší vývoj aplikací, jelikož vývojáři dodá aplikaci, kterou pro základní použití není potřeba složitě konfigurovat.

Výhody

- Velké množství knihoven.
- *Spring Boot* umožňuje rychlý vývoj.

Nevýhody

- Velká náročnost na výpočetní prostředky.
- Značná velikost konečné aplikace.

[2] První vydání bylo v roce 2012, viz. `golang.org/project/` [cit. 2020-11-22].
[3] Domovská stránka projektu : `https://spring.io/`[cit. 2020-11-22]

5.3.4 Java - Quarkus

Quarkus[4], jakožto *framework* pro *Java*, přebírá všechny výhody a nevýhody tohoto jazyka. *Quarkus* je *framework* zaměřený na *microservices* a *Docker* (2.7). Proto je relativně nenáročný na výpočetní prostředky. Jedním z hlavních prostředků, kterým dosahuje tohoto cíle, je možnost použití *GraalVM*.

Výhody

- Velké množství knihoven.
- Malá velikost konečné aplikace.
- Malá náročnost na výpočetní prostředky.
- Důraz na *Docker*.
- Nativní podpora reaktivního programování.
- Při vývoji snadno umožňuje *hot deployment*.

Nevýhody

- Malé množství dokumentace a návodů.
- Omezená funkcionalita

Z vypsaných možností jsem se nakonec rozhodl pro kombinaci *Java* a *Quarkus*. Jazyk *Java* jsem zvolil, jelikož jsem s ním poměrně dobře obeznámen, a vyhovuje mi statické typování. *Quarkus* jsem zvolil z důvodů jeho zaměření na *Docker* a malé náročnosti na výpočetní prostředky.

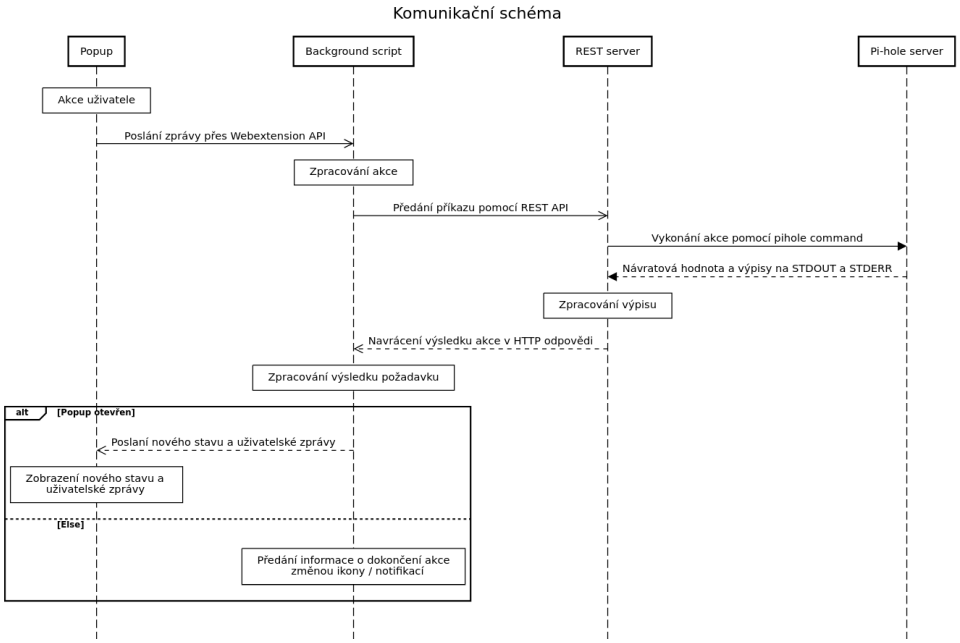
5.4 Komunikace

Další důležitou součástí je komunikační kanál. Existuje sice více možností jako *gRPC*, *SOAP* či *GraphQL*, ale z hlediska jednoduchosti, čitelnosti a rozšířenosti jasně vede *REST API* [21]. Další výhodou je, že na klientské straně nejsou potřeba žádné další knihovny, jelikož požadavky na *REST API* lze jednoduše posílat z nativní funkce *fetch*.
Schéma komunikace je graficky znázorněno v 5.1.

5.4.1 Quarkus a REST API

Quarkus pro *REST API* používá knihovnu *RESTeasy*, pro vytvoření *REST endpointu* pouze stačí přidat příslušnou anotaci k veřejné metodě třídy. *Serializace* i *deserializace* probíhá automaticky, lze samozřejmě dodat vlastní objekty, které *serializaci* či *deserializaci* provedou.

[4]Stránka projektu zde : `quarkus.io` [cit. 2020-11-22].



Obrázek 5.1: Komunikace mezi moduly

5.5 Klientská část

Jako klientská část bude vytvořen doplněk do prohlížeče, i když použitím *REST* a zveřejněním jeho schématu nebude těžké vytvořit a používat např. mobilní aplikaci jako klienta. Uživatel bude doplněk ovládat pomocí *toolbar popup*, který bude příkazy předávat *background script*. *Background script* bude hlavní součástí celého doplňku, bude komunikovat se serverem, udržovat stav a upozorňovat uživatele.

5.5.1 Upozornění uživatele

Pokud uživatel popup zavře, máme před sebou dilema, jak uživatele upozornit na změnu stavu. Nabízí se možnost použití notifikace (4.4.4), ale to je relativně invazivní způsob upozornění. Při časté změně stavu by to mohlo uživatele doslova zahltit, a tím odradit od používání doplňku. Tento způsob by tedy bylo dobré používat jen u těch uživatelů, kteří tuto metodu budou preferovat (například explicitním povolením v nastavení doplňku).

Druhou možností je upozornit na změnu stavu změnou ikony v nástrojové liště. Je to poměrně často užívaná metoda, kterou lze snadno implementovat. Nová ikona by na sobě mohla mít například červený puntík, což je vizuálně dostatečně odlišitelné. Tato metoda by tedy byla primárním způsobem, jak uživatele upozornit na změnu stavu.

5.5.2 Udržování stavu

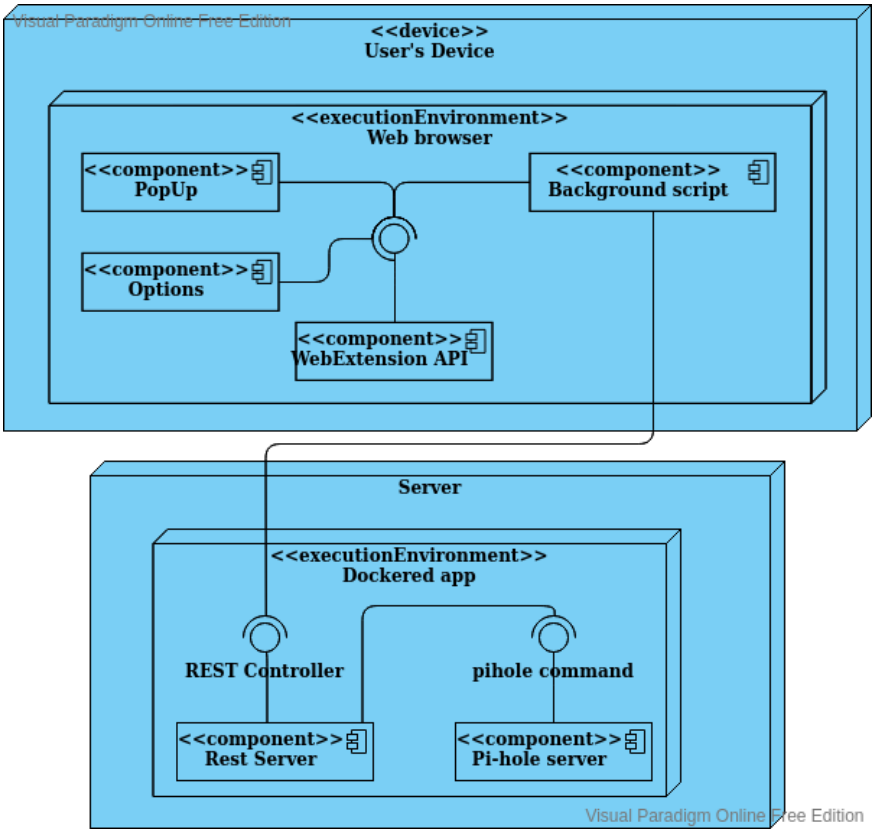
Toolbar popup nemá žádný způsob jak přechovávat stav mezi zapnutími. Proto je jedinou možností udržovat stav v *background script*, a *popup* se vždy při spuštění zeptá na stav, který pak zobrazí.

5.6 Nasazení

Klientská část aplikace bude distribuována a instalována přes obchod s doplňky prohlížeče. Serverová část bude distribuována dvěma způsoby, buď jako *Docker image* (2.7.1), pokud bude uživatel využívá instalaci *Pi-hole* pomocí *Pi-hole Docker image* (B.3). V této podobě by distribuovaný *Docker obraz* vycházel z oficiálního *Pi-hole obrazu*, a tudíž by obsahoval jak *Pi-hole*, tak *REST* server. V diagramu 5.2 můžeme vidět schéma nasazení pomocí Docker obrazu.

V případě, že má uživatel nainstalovanou *Pi-hole* přímo bez virtualizace (viz B.4), byl by k dispozici instalační skript, který by stáhl spustitelný *REST* server a nastavil by prostředí.

Podporované operační prostředí bude stejné jako u *Pi-hole*. To znamená Linux s 32- či 64-bitovým procesorem.



Obrázek 5.2: Zjednodušený diagram nasazení pro případ distribuce přes Docker

Kapitola 6
Implementační část

Implementace práce vycházela z kapitoly Návrh (5). Byly vytvořeny dvě separátní části, které spolu komunikují. Tato kapitola se nejdříve zabývá implementací serverové části, a poté přechází k implementaci doplňku jako klientské části aplikace.

6.1 Server

Jako server byla naimplementována webová aplikace využívající programovací jazyk *Java* a *framework Quarkus*. Server vystavuje *REST* rozhraní, pomocí kterého s ním komunikuje klientská aplikace.

6.1.1 Nástroje pro vývoj

Na začátku projektu byla jako verzovací systém použita školní instance systému *GitLab*, ale během projektu jsem přešel na komerční systém *GitHub*. Tento přechod nastal jak kvůli zachování projektu do budoucna (na školním systému by došlo ke smazání po ukončení studia), tak kvůli *CI/CD* - *GitHub* totiž zdarma poskytuje virtuální stroje (*runner*) na *CI/CD* (*GitHub Actions* - více informací v sekci 2.10.1).

Dalším důležitým nástrojem byl systém *Maven* (2.9). *Maven* byl zvolen oproti konkurenčním řešením kvůli větší podpoře komunity při kombinaci s *Quarkus*, a mé zkušenosti s ním.

6.1.2 Architektura

Server má jednoduchou, tří-vrstvou architekturu. Klienti komunikují s serverem pomocí *REST* rozhraní. To je implementováno v *CommandController*. Tento *kontroler* volá jednotlivé *služby* (*service*), které mají na starosti logiku. A tyto *služby* volají jednotlivé *příkazy* (*Command*), které pomocí *CommandExecutor* vykonávají *BASH* příkazy.

REST rozhraní

REST rozhraní je implementováno pomocí knihovny *RESTEasy*[1]. Tato knihovna umožňuje jednoduchou tvorbu *REST* rozhraní pomocí *anotací*. Stačí *veřejnou* (*public*) metodu označit anotací metody (např. *@GET* nebo *@PUT*) a metoda bude automaticky zavolána, pokud přijde *HTTP* (*Hypertext Transfer Protocol*) dotaz se správnou metodou (viz. ukázka 6.1)

```
@HEAD
public Response heartbeat() {
    return Response.ok().build();
}
```

Listing 6.1: Ukázka jednoduché tvorby REST rozhraní pomocí anotací

Jak již bylo řečeno, *CommandExecutor* nadále volá metody *service* vrstvy, kterým předává parametry z *HTTP* dotazů.

Vrstva služeb

Vrstva služeb (*service layer*) je poměrně jednoduchá, jelikož není potřeba vykonávat složitou logiku. Služby tedy hlavně určují, jaké *příkazy* volat, popřípadě jich volají několik a skládají z nich ucelené informace.

Příkazová vrstva

Základem celého serveru jsou *příkazy* (*command*). Pomocí nich se instance *Pi-hole* ovládá, či se zjišťuje její stav. Každý příkaz má *CommandExecutor* (vykonávač příkazů), kterému předá textový příkaz k provedení. *CommandExecutor* příkaz vykoná a vrátí výstupy z něj (výstup z standardního výstupu, z chybového výstupu a návratový kód). *Command* se poté na základě výstupů rozhodne, jestli nastal požadovaný stav, a pokud ne, vyhodí výjimku.

6.1.3 BASH Injection

Jelikož při přidávání domén do *Black/White-listu* dostáváme textový vstup od uživatele, který pak používáme jako argument v příkazové řádce, je nutno se obávat *BASH injection* (2.11). Pokud bychom *BASH injection* nezabránili, mohli bychom útočníkům poskytnout snadný útočný vektor. A jelikož není implementovaná žádná forma *autorizace/ autentizace*, tento vektor by byl poskytnut každému, kdo se nachází v síti, kde běží implementovaný server. *BASH injection* lze zabránit umístěním zpětného lomítka před speciální znaky. Toto provádíme pomocí *RegEx výrazu*.

6.1.4 Quarkus versus Spring Boot

Jelikož jsem v návrhu (5.3) uvažoval mezi *Quarkus* a *Spring Boot*, a protože mám zkušenosti s *Spring Boot* platformou, rád bych zde uvedl pár rozdílů mezi nimi, které jsem vypozoroval při implementaci.

[1] Více informací o knihovně zde: `https://resteasy.github.io/`

Hot deployment

Jak již bylo napsáno při výběru technologií, *Quarkus* podporuje *Hot deployment* bez jakéhokoliv nastavování. Stačí spustit *Maven goal* (2.9.1) `quarkus:dev`, který po každé změně v zdrojovém kódu[2] upraví běžící aplikaci. *Spring Boot* toto sice také umožňuje, ale je potřeba podpora *IDE* (*Integrated development environment*) a speciální knihovna. Při vývoji mi *hot deployment* značně urychloval postup, a pokud je k dispozici bez speciálního nastavování, beru jej jako zásádní výhodu vybraného řešení.

Exception handling

Spring Boot má podle mého názoru lépe vyřešené zpracování *výjimek* (*exceptions*). Pokud výjimku nezachytáváme v kódu, můžeme ještě zadefinovat speciální metody, které výjimku zpracují, pokud "vybublá"až k *REST* rozhraní. *Spring Boot* nabízí možnost zadefinování speciální třídy, která dědí z `ResponseEntityExceptionHandler`. Zde poté můžeme implementovat veřejné metody, které výjimky budou zpracovávat. Příklad takové metody je např v 6.2.

```
/// muzeme urcit ktere vyjimky metoda zpracovava, pokud tato anotace chybi bude
  zpracovavat vsechny nezpracovane vyjimky

@ExceptionHandler({IllegalArgumentException.class, NullPointerException.class})

public final ResponseEntity<Object> handleExceptionInternal(java.lang.Exception ex) {

    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                         .body("Internal exception, please report this to admin" );
}
```

Listing 6.2: Ukázka jednoduché metody, která zpracovává výjimku v Spring Boot

Quarkus se vydal obdobnou cestou, ale na každou třídu výjimky je potřeba jedna třída. To podle mého názoru vede k množství zbytečného kódu, což vede k nepřehlednosti, zvláště u větších projektů. Příklad takové třídy je možné vidět v příkladu 6.3.

```
@Provider
public class PiholeAlreadyDisabledHandler implements
  ExceptionMapper<PiholeAlreadyDisabledException> {

    @Override
    public Response toResponse(PiholeAlreadyDisabledException exception) {
            return Response.status(Response.Status.CONFLICT)
                    .entity(new CommandResponse(false, "Pihole is already disabled!"))
                    .build();

    }
}
```

Listing 6.3: Ukázka jednoduché třídy, která zpracovává výjimku v Quarkus

[2]Přesnější vyjádření je po každé změně zdrojových souborů a nějakém vnějším *IO* požadavku (např. *HTTP* požadavek)

6.2 Plugin

Jako klientská aplikace byl naimplementován doplněk do prohlížeče využívající *WebExtensions API* a *Extension API*. Kromě implementace funkční části aplikace proběhla i implementace uživatelského rozhraní.

6.2.1 Nástroje pro vývoj

Mezi zásadní nástroje pro vývoj patřil *Web-ext* (2.5). S jeho pomocí jsem při vývoji testoval fungování doplňku v prohlížeči *Mozilla Firefox*. *Chromium* zase umožňuje nahrání *nezabaleného* doplňku (*load unpacked extension*), které také podporuje *hot reload*. Pomocí nástroje *Web-ext* jsem zdrojové kódy i *zabaloval* (2.5), formát, do kterého *Web-ext* doplňky zabaluje, akceptují oba dva obchody, což zjednodušuje jejich distribuci.

Dále byl použit balíčkovací systém *NPM*, který primárně slouží k získávání knihovny *webextension-polyfill* (6.2.3).

6.2.2 Architektura

Plugin se skládá ze tří částí. *Background skriptu*, který zodpovídá za komunikaci se serverem a udržování stavu. *Popup části*, která působí jako ovládací panel pro uživatele. A nakonec *nastavení* (*options*), kde uživatel nastavuje např. *IP* adresu serveru. Popup i nastavení posílají akce uživatele background skriptu, který je ukládá, či posílá na server.

Popup

Popup jako takový slouží ke dvěma účelům - *informování* o stavu a *poskytování* rozhraní pro akce uživatele.

Po kliknutí uživatele na ikonku doplňku mohou nastat dva scénáře. Pokud doplněk není připojený k serveru (zjišťuje se pomocí hearthbeat - 6.2.6), zobrazí se dialog podobný nastavení, kde uživatel může změnit *IP* adresu a port. Pokud je doplněk připojen k serveru, zobrazí se hlavní ovládací panel (snímek 8.4). Jelikož chceme ponechat komunikaci mezi doplňkem a serverem na minimu, aktuální stav *Pi-hole* se získává až po otevření. V kombinaci s relativně pomalým *pihole command* (na *Raspberry Pi 3 B+* trvá získání stavu až tři sekundy) se tedy po otevření doplňku dostáváme do situace, kdy máme neaktuální data. Uživatel je na toto upozorňován načítací obrazovkou.

Background skript

Background skript je mozkem doplňku. Zpracovává zprávy od ostatních dvou částí doplňku, které jsou posílány pomocí *one-off messages* (4.5.1). Na základě těchto zpráv buďto mění svůj stav (změna *IP* adresy), posílá požadavky na server (příkaz na vypnutí *DNS* blokování), či jen poskytuje některá data (aktuální stav).

Nastavení

Nastavení nabývá podoby jednoduchého formuláře, kde uživatel upřesňuje nastavení doplňku. Uživatel může nastavit: *IP adresu serveru*, *port*, na kterém server poslouchá a zda si přeje aby doplněk využíval systémové *notifikace*.

6.2.3 Webextension-polyfill

Pro zjednodušení implementace byla využita knihovna *Webextension-polyfill*[3]. Jak bylo uvedeno v kapitole o vývoji doplňků (4.8), *Chromium* používá návrhový vzor *callback* při zpracovávání asynchronních událostí, ale u *Mozilla Firefox* je toto zastaralé. Aby nebylo nutné udržovat dvě verze aplikace, byla využita tato knihovna, která do *Chromium* tuto funkcionalitu doplňuje.

6.2.4 Mozilla Firefox versus Chromium

Jelikož jsem v kapitole o vývoji doplňků věnoval podstatnou část multiplatformnosti spíše z teoretického hlediska, rád bych nyní věnoval pár řádků hledisku praktickému.

Po použití knihovny *Webextension-polyfill* odpadly jakékoliv rozdíly mezi *WebExtensions API* a *Extension API*. Po celou dobu vývoje jsem nenašel žádný funkční rozdíl, což mne příjemně překvapilo.

Kde ale byly rozdíly, byla vzhledová stránka doplňku. Oba prohlížeče mají vlastní, odlišný, *user-agent* vzhled[4], tudíž sjednotit vzhled a velikost nebylo bezbolestné.

6.2.5 Toast messages

Informace o výsledcích uživatelských akcí jsou uživateli předávány pomocí jednoduchých zpráv (*toast messages*), které se nachází v popup. Tyto zprávy po definovaném čase zmizí, aby nedošlo k zahlcení uživatele. Pokud není popup otevřený v čase vytvoření zprávy, zpráva se uloží do fronty zpráv, a uživateli se ukáže po otevření doplňku. Pokud má uživatel povolené notifikace, obdrží navíc i notifikaci s textem zprávy.

6.2.6 Heartbeat

Aby doplněk mohl jednoduše určit, jestli existuje spojení se serverem, byl naimplementován jednoduchý *heartbeat endpoint*. Pokud *HTTP* požadavek na tento *endpoint* proběhne v pořádku, je server online a doplněk s ním může komunikovat. Pokud žádost neproběhne v pořádku, server je offline a doplněk přechází do *offline* řežimu.

[3] Dostupné zde :`https://github.com/mozilla/webextension-polyfill/` [cit. 2020-11-20].

[4] *User-agent* je soubor *CSS* pravidel specifický pro každý prohlížeč. Pokud v *CSS* pravidlech není definována vzhledová vlastnost, použije se pravidlo z tohoto souboru

6.2.7 CORS

CORS (*Cross-Origin Resource Sharing*) je bezpečnostní mechanismus, který zabraňuje stránkám posílat *HTTP* požadavky "jménem" klienta na jiné domény. Aby mohla webová stránka posílat *HTTP* požadavky na jinou než vlastní doménu, musí dotazovaná doména explicitně povolit *CORS* v *HTTP hlavičce* (*header*). Pokud není povolen, prohlížeč požadavek zamítne, a stránce se doména jeví jako nedostupná.

V serverové části projektu je tedy *CORS* povolen, *CORS* hlavičku server posílá se všemi *HTTP* odpověďmi. Zajímavá věc ale nastává, pokud server dostupný není. Pokud doplněk otevřeme a přejdeme do vývojářské konzole background skriptu[5] v prohlížeči *Mozilla Firefox*, uvidíme, že prohlížeč hlásí

```
'Cross-Origin Request Blocked: The Same Origin Policy disallows
reading the remote resource at http://192.168.1.0:8221/alanine.
(Reason: CORS request did not succeed).'
```

(viz obrázek 6.1). *Chromium* hlásí chybu, kterou bychom očekávali spíše, a to

```
'HEAD http://192.168.50.91:8221/alanine
net::ERR\_ADDRESS\_UNREACHABLE'
```

U prohlížeče *Firefox* toto chování naznačuje, že prohlížeč kontroluje *CORS* skutečně u všech žádostí, bez ohledu na výsledek, či zda požadavek skutečně odešel.

Obrázek 6.1: *Firefox* konzole hlásící chybu *CORSfailed* pokud server není dostupný

6.2.8 Vykonávání cizích skriptů

Doplněk využívá službu *Font Awesome*, která mimojiné bezplatně poskytuje mnoho ikon, které lze použít v HTML obsahu. Ale ke svému fungování potřebuje *Font Awesome* skript, který umožní vykreslení těchto ikon. URL adresa skriptu se vkládá do HTML hlavičky, a skript se stahuje při načítání stránky (v tomto případě po otevření popupu). Toto je ale u doplňku veliký bezpečnostní problém. Jelikož zdrojové kódy doplňku vždy prochází bezpečnostní kontrolou (viz kapitola 7.1), takovéto stahování skriptů z neznámých zdrojů bez omezení by znamenalo snadné obejití této kontroly. K povolení vykonávání externích skriptů tedy musíme do manifestu umístit položku

```
"content_security_policy": "script-src 'self' 'unsafe-eval'
    https://kit.fontawesome.com; object-src 'self'".
```

[5] Konzoli background skriptu lze zobrazit pomocí stránky about:debugging

Tato položka vyjmenovává povolené zdroje skriptů a povolené zdroje HTML objektů (HTML objekt je například vložený obrázek nebo video). Více o této bezpečnostní politice na MDN web docs[6].

Při bezpečnostní kontrole je tedy zřejmé, z jakého zdroje jsou skripty stahovány, a pokud zdroj není dostatečně důvěryhodný, doplněk bude zamítnut.

[6] `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/content_security_policy` [cit. 2021-4-23]

Kapitola 7
Nasazení

Důležitou součástí životního cyklu každé aplikace je i nasazení. Nasazením je v této práci myšlena distribuce programu mezi uživatele a instalace na jejich zařízeních. Jelikož byly naimplementovány dvě aplikace, v této kapitole bude popsáno nasazení obou dvou částí.

7.1 Doplněk

Distribuce doplňků probíhala přes obchody s doplňky *Mozilla Addons* a *Chrome Web Store*. Postup při využívání obou platforem byl podobný, avšak s drobnými, ale zajímavými rozdíly. Nejdříve bych tedy popsal obecně společný postup, a poté se zaměřil na rozdíly jak v publikování nových doplňků / nahrávání nových verzí, tak v možnostech správy.

7.1.1 Obecný průběh

Pokud chceme námi vytvořený doplněk zveřejnit, musíme absolvovat pár kroků.

1. Vytvoření vývojářských účtů pokud jimi není disponováno.
2. Zabalení zdrojových kódů doplňku - vytvoření *ZIP* archivu. Toto je možné automatizovat použitím nástroje *Web-ext* (2.5).
3. Nahrání zabaleného doplňku do obchodu.
4. Vyplnění potřebných informací a prohlášení.
5. Odeslání doplňku k bezpečnostní kontrole. Pokud doplněk kontrolou neprojde, je potřeba doplněk upravit a pokračovat od kroku dva.
6. Zveřejnění doplňku veřejnosti.

Nahrání nové verze i nahrání nového doplňku mají stejný průběh, liší se pouze vyplňované údaje o doplňku.

7.1.2 Chrome Web Store

U *Chrome Web Store* je zajímavý rozdíl už v prvním kroku. Registrace vývojářského účtu je totiž zpoplatněna poplatkem 5 USD. Další zajímavostí je nutnost použít *Google Chrome* při nahrávání zabalených zdrojových kódů. Při nahrávání přes *Chromium* totiž obchod hlásí neznámou chybu (*Chromium verze 90.0.4430.72*).

Pokud vývojář překoná všechny tyto nástrahy, dostane se k vyplňování informací, kde je nutno zodpovědět velké množství otázek jak o budoucím nakládání s daty uživatelů, tak k vyžadovaným povolením (*permissions*). Před nahráním je také nutné vyplnit množství prohlášení, např. že nebudeme poskytovat data o uživatelích třetím stranám.

Poté je nutné počítat s bezpečnostní kontrolou, na jejíž výsledek je nutno čekat o poznání déle než u *Mozilla Addons* - nejkratší čas čekání byl pět hodin. A je o poznání přísnější, např. v rámci projektu odhalila povolení, které nebylo v kódu využíváno[1]. Kontrola u *Mozilla Addons* toto neobjevila.

7.1.3 Mozilla Addons

Mozilla Addons nabízí oproti *Chrome Web Store* jednoduchý, až strohý způsob jak publikovat a rozšiřovat svůj doplněk. Ve všech krocích je potřeba udělat to, co vývojář očekává, bez nepříjemných překvapení.

Také je o mnoho rychlejší bezpečnostní kontrola - vždy proběhla do tří hodin od odeslání.

7.1.4 Statistiky

Obě dvě platformy nabízí podobné statistiky o doplňku. Je možné zobrazit statistiku počtu stahování a počtu aktivních uživatelů - viz 7.1. Také zobrazuje počty uživatelů dle - verze prohlížeče, jazyku uživatele a operačních systémech - 7.2.

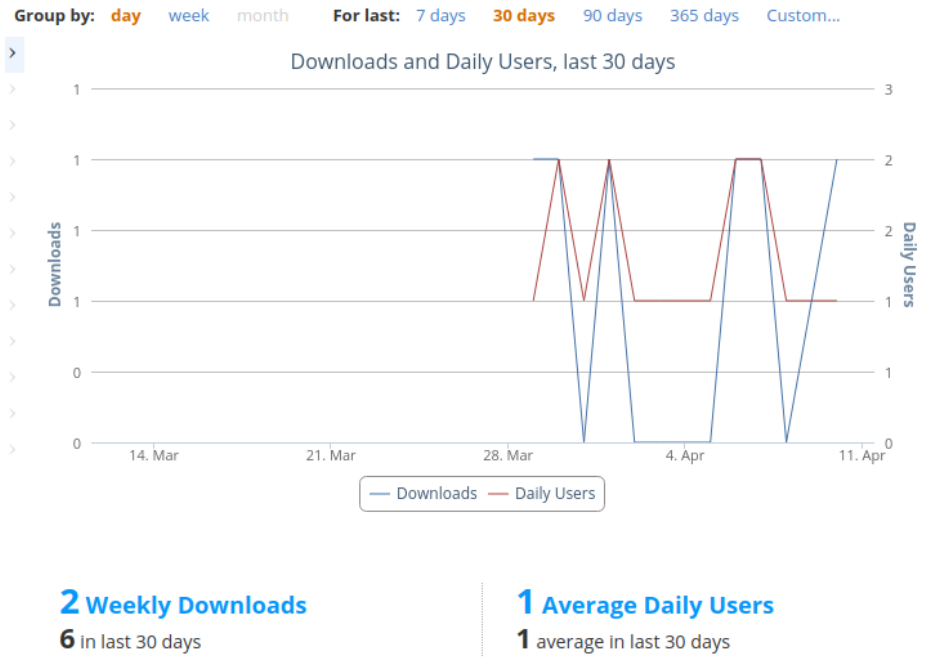
7.1.5 Uživatelský pohled

Již jsme si popsali pohled vývojáře na využívání obchodů s doplňky. Ale jak je na tom uživatel? Jak je pro něj složitá instalace, odinstalace a update na novou verzi ?

Instalace

Instalace je velmi jednoduchá. Stačí najít záznam doplňku v zvoleném obchodě, a kliknout na tlačítko instalace. Pokud doplněk vyžaduje povolení (*permissions*), musí uživatel explicitně potvrdit jejich povolení. Poté se doplněk nainstaluje a je připraven k použití.

[1] V kódu byla funkce, která zjišťovala, jestli je popup otevřen. Ta volala externí funkci, u které jsem se domníval (špatně jsem pochopil dokumentaci), že potřebuje povolení **activeTab**.



Obrázek 7.1: Příklad grafu počtu stažení a aktivních uživatelů v Mozzilla Addons.

Odinstalace

Odinstalace je také velmi snadná. Stačí na doplněk v nástrojové liště kliknout pravým tlačítkem myši a vybrat možnost *odstranit z prohlížeče*.

Update

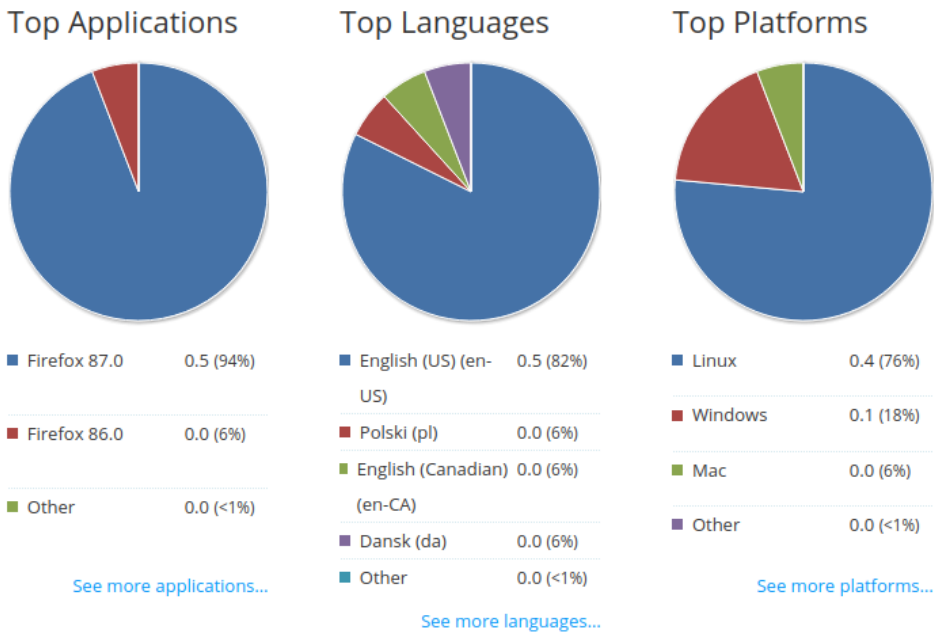
Update nastává pro uživatele automaticky, ale může si jej spustit manuálně v nastavení doplňku.

7.2 Serverová část

Distribuce serverové části byla o mnoho složitější než distribuce doplňku do prohlížeče. Dle návrhu (5.6) je nutno zajistit distribuci a instalaci dvěma cestami. Instalací pomocí *Docker image* a pomocí *instalačního skriptu*.

Oba dva způsoby ale sdílejí jeden problém, který bohužel nebyl odhalen v návrhu, a jehož přehlédnutí značně zkomplikovalo celou distribuci. Původním cílem bylo distribuovat *native image* (2.6.1) kvůli jeho menší zátěži na výpočetní výkon zařízení. Ovšem *GraalVM* v současné době nepodporuje *32-bit* architekturu[2]. A ačkoliv většina nejnovějších *Raspberry Pi* má *64-bitový* processor, *64-bitová* verze doporučovaného operačního systému *Raspberry Pi*

[2]Seznam podporovaných platform zde : `https://www.oracle.com/tools/technologies/graalvm-supported-platforms.html` [cit. 2021-05-06]



Obrázek 7.2: Příklad grafu počtu uživatelů dle verze prohlížeče, jazyku uživatele a operačního systému v Mozzila Addons.

OS je v rané fázi testování. Protože vyřazení *32-bitové* architektury by tedy vyřadilo podstatnou část potencionálních uživatelů, je potřeba distribuovat jak *native* verzi (pro *AMD64* a *ARM64* architektury), tak *Java* verzi pro ostatní (např. *ARMV7* architekturu).

7.2.1 Společné požadavky

Požadavky native image verze a Java verze se od sebe v zásadě neliší. Pouze Java verze potřebuje k fungování funkční instalaci *Java JRE* (*Quarkus* požaduje verzi 8 či 11+). Obě verze mají jako závislost *Pi-hole*. A poté mechanismus, který server po každém startu operačního systému spustí.

7.2.2 Instalační skript

Instalační skript jako mechanismus ke spuštění serveru po každém startu operačního systému využívá unix služby *cron*.

Instalace

Instalace probíhá spuštěním instalačního skriptu. Tento skript je odvozen od instalačního skriptu *Pi-hole*, jelikož instalační skript *Pi-hole* dokáže rozlišit architekturu, což je nutné pro stažení správné verze serveru.

Instalační skript je přímočarý, i když kvůli rozpoznávání architektury poměrně velký (v počtu příkazů). Nejprve zkontroluje přítomnost příkazu `pihole`, poté vytvoří potřebnou složku, a podle architektury stáhne buď

native verzi, nebo Java verzi z *GitHub* repozitáře. Pokud je nutno stáhnout Java verzi, je zkontrolována i přítomnost příkazu `java`. Následně se nastaví služba *cron* na spuštění serveru po každém startu operačního systému. A nakonec je server spuštěn.

Odinstalace

Pro pohodlnost uživatele je dodán i odinstalační skript. Ten je také jednoduchý, upraví nastavení *cron* služby a poté smaže všechny soubory a složky serveru.

Update

Instalace nových verzí serveru není automatizována. Byla zvolena primitivní metoda, kdy uživatel nejdříve server musí odinstalovat, a poté znovu nainstalovat.

7.2.3 Docker install

Tvorba instalace pomocí nástroje *Docker* byla složitější než tvorba instalačního skriptu. To z několika důvodů, které si rozebereme později.

Instalace

Instalace, jak bylo cílem, napodobuje instalaci *Pi-hole* obrazu. Jelikož při vytváření vycházíme[3] z oficiálního *Pi-hole* obrazu, může uživatel použít stejné nastavení, pouze musí povolit jeden port navíc.

Odinstalace

Odinstalace probíhá naprosto stejně jako u *Pi-hole* obrazu. *Odinstalace* je prosté smazání kontejneru.

Update

I *update* na novější verzi obrazu je stejný jako u originálního obrazu. Stačí kontejner smazat, provést příkaz `docker pull`, který stáhne nejnovější verzi obrazu, a poté znovu nainstalovat.

Tvorba ARM64 obrazu

Hlavním problémem byla chybějící možnost nastavit u *GraalVM* cílovou platformu *native image*. *GraalVM* tedy kompiluje *native image* pouze na platformu, na které je kompilace spouštěna. Tudíž je snadné vytvořit *AMD64* variantu, ale pro tvorbu *ARM64* varianty by bylo potřeba dalšího prostředí. Což nelze uznat za vhodné řešení.

[3] Docker obrazy mají možnost rozšiřovat jiné obrazy. Pro představu podobné dědění u objektového programování.

Po delším výzkumu bylo nalezeno řešení skládající se ze dvou částí. První částí je poskytnutí vlastního Docker obrazu[4] při tvoření native image (viz. 2.6.1). Tento Docker obraz je *platform-specific* pro *ARM64*, tudíž GraalVM vytváří native image pro *ARM64*. Ale tento obraz nelze spustit na jiné architektuře než *ARM64*. V tuto chvíli přichází *QEMU* (2.8). *QEMU* umožňuje emulaci pro jednotlivé Docker kontejnery, tudíž nám umožňuje spustit *ARM64* Docker obraz nezávisle na architektuře prostředí. Cenou za tuto emulaci je značná časová náročnost tohoto řešení. Kompilace native image bez *QEMU* trvá obvykle kolem 4 minut, kdežto kompilace s emulátorem *QEMU* trvá kolem 30 minut. Ale značně zjednodušuje vytváření Docker obrazů např. v *CI/CD* (2.10).

S6 Overlay

Pi-hole Docker využívá pro jako správce procesů v kontejneru *S6 overlay*[5]. *S6 overlay* je zjednodušeně služba, která se stará o životní cyklus uživatelských aplikací. Po startu kontejneru vykoná námi dodaný skript, a při vypínání vykoná jiný, námi dodaný skript. [6] Tento správce byl tedy použit jako alternativa *cron* služby, která byla využita v 7.2.2.

Docker Image a nové verze Pi-hole

Jelikož náš Docker obraz obsahuje i *Pi-hole*, je nutné po každém vydání nové verze *Pi-hole* znovu vytvořit a nahrát celý obraz. Toto ale značně usnadňuje *CI/CD* (viz. 2.10), které celý proces vytváření obrazu automatizuje. Uživatel si poté může stáhnout novou verzi obrazu s novou verzí *Pi-hole*. Toto s sebou nese i nečekanou výhodu, jelikož si tímto uživatel stáhne i nejnovější verzi serveru.

7.2.4 Vytížení serveru

Zásadní nevýhodou zvoleného řešení separátního serveru je hrozba zvýšeného vytížení serveru. Tudíž bylo nutné porovnat vytížení serveru s implementovaným serverem a *Pi-hole* a vytížení pouze s *Pi-hole*.

Měření bylo provedeno simultánně na *Raspberry Pi 3B+*. Obě dvě varianty běžely v Docker kontejneru.

[4] Dockerfile pro tento obraz je k nalezení zde: `https://github.com/Kulda22/alanine/blob/master/Dockerfile.nativebuilder` [cit. 2021-05-07]. V podstatě se jedná o jednoduchý Docker obraz, který má nainstalovaný *GraalVM native image kompilátor* a má ho nastavený jako *entrypoint* (příkaz, který se zavolá při spuštění Docker kontejneru). Má tedy stejné vlastnosti a chování jako výchozí Docker obraz (ten je používán pokud nespecifikujeme svůj vlastní), ale je *platform-specific*.

[5] Github projektu: `https://github.com/just-containers/s6-overlay/`[cit. 2021-05-05]

[6] Příklady těchto skriptů jsou ve zdrojovém kódu ve složce s6-fastjar. (Např zde : `https://github.com/Kulda22/alanine/blob/master/s6-fastjar/alanine/run` [cit. 2021-05-07])

NAME	CPU %	MEM %	MEM Usage [MB]	NET OUTPUT [MB]
Pihole	34.14	2.03	18.81	2.76
alanine	3.48	6.08	56.27	0.51

Tabulka 7.1: Tabulka průměrného vytížení prostředku při uživatelských akcích

K měření byl použit skript *Docker Stats Graph*[7], který využívá Docker příkazu `stats`. Tento příkaz poskytuje data o každém kontejneru. Mezi data patří: *procentuální vytížení procesoru, využití paměti v MB, procentuální vytížení paměti, síťové i diskové vstupy/ výstupy (I/O) a součet počtu procesů a vláken jádra operačního systému*[8]. Skript vybraná data ukládá do *CSV (Comma-Separated Values)* souboru. Tato data byla poté zanesena do grafu pomocí *Python skriptu* ze stejného repozitáře.

Byla provedena dvě měření. První, při kterém byly vykonány[9] uživatelské akce - *vypnutí a zapnutí DNS blokování, vypnutí a zapnutí logování, přidání domény do whitelistu a blacklistu*. Druhé měření bylo klidové - bez akcí uživatele.

Výsledky byly velmi zajímavé. Tabulky 7.2 a 7.1 ukazují průměrné vytížení prostředků. Grafy 7.3, 7.4, 7.5 a 7.6 ukazují využití CPU / paměti (při uživatelských akcích byla data snímána každou sekundu, bez uživatelských akcí každých pět sekund). Kontejner s *Pi-hole* a serverem je pod názvem *alanine*, kontejner pouze s *Pi-hole* je pod názvem *Pihole*

Jak je vidět v tabulce a na grafech, *Pi-hole* značně ztrácí při uživatelských akcích - sestavení celé HTML stránky je značně náročnější než jednoduchá *REST* odpověď. Na grafu 7.3 je možné si všimnout prvního vrcholu u *alanine* v čase 12:54:00. V tomto čase doplněk žádal server o stav - pomocí `pihole status` (3.2.7), což je operace náročná na čas i výpočetní prostředky.

Pokud ale uživatel žádné akce nekoná, využití CPU je srovnatelné. Tento test nejspíše ovlivnily externí faktory (soudím z velkého množství vysokých lokálních maxim v grafu) - DNS dotazy, vnitřní akce Pihole apod., ale i tak můžeme říci, že server v této situaci procesor znatelně více nezatěžuje.

Jasnější situace je u využití paměti. Tady je rozdíl konstantně trojnásobný v prospěch samotné *Pi-hole*. Ale využití 60 MB paměti je stále přijatelné, vzhledem k tomu, že i např. *Raspberry Pi Zero* má 512MB RAM [5].

Závěrem tedy můžeme říci, že server není náročný na výpočetní prostředky, ba naopak je dokáže ušetřit při aktivním používání.

7.2.5 CI/CD

Součástí práce bylo i vytvoření jednoduchého *CI/CD* (2.10). Bylo vytvořeno pomocí *GitHub Actions* (2.10.1). Byly vytvořeny dva *scénáře (workflows)*.

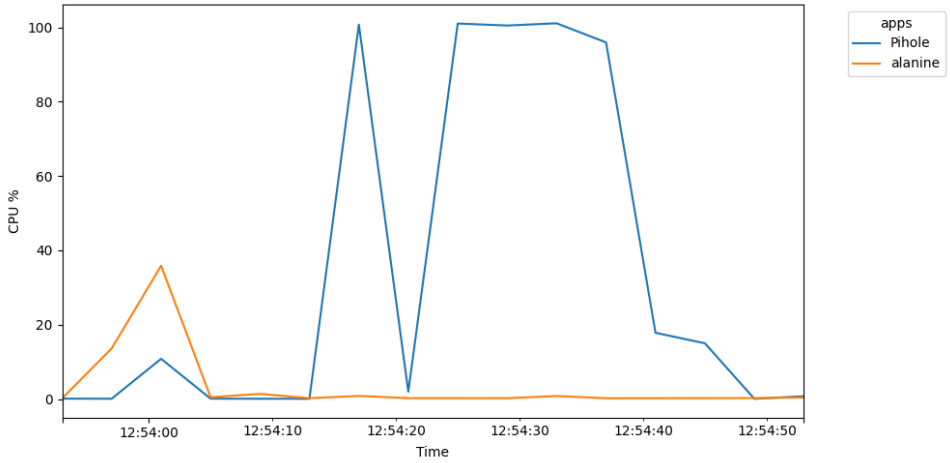
[7] Dostupný z `https://github.com/sylhare/docker-stats-graph` [cit 2021-04-27]

[8] Více informací o příkazu zde: `https://docs.docker.com/engine/reference/commandline/stats/` [cit. 27.4.2021]

[9] U kontejneru pouze s *Pi-hole* byly akce vykonány pomocí webového rozhraní, u kontejneru s naimplementovaným serverem pomocí doplňku.

NAME	CPU %	MEM %	MEM Usage [MB]	NET OUTPUT [MB]
Pihole	0.88	1.99	18.44	1.84
alanine	0.78	6.07	56.26	0.45

Tabulka 7.2: Tabulka průměrného vytížení prostředku bez uživatelských akcí



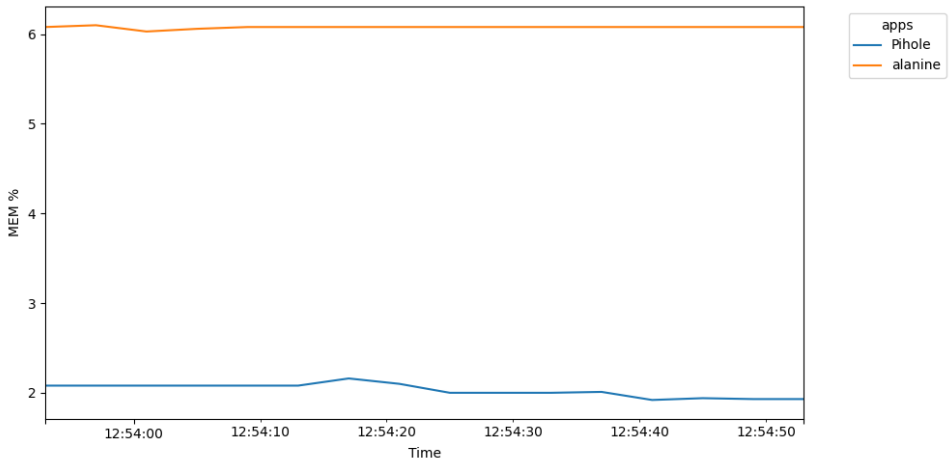
Obrázek 7.3: Graf využití CPU v procentech při uživatelských akcích

Jeden scénář nastává, když nastane *Git push* nebo *pull request* na *development* (vývojové) větvi. Ten kód otestuje pomocí integračních testů, a poté vytvoří vývojové Docker obrazy a nahraje je do Docker repozitáře.

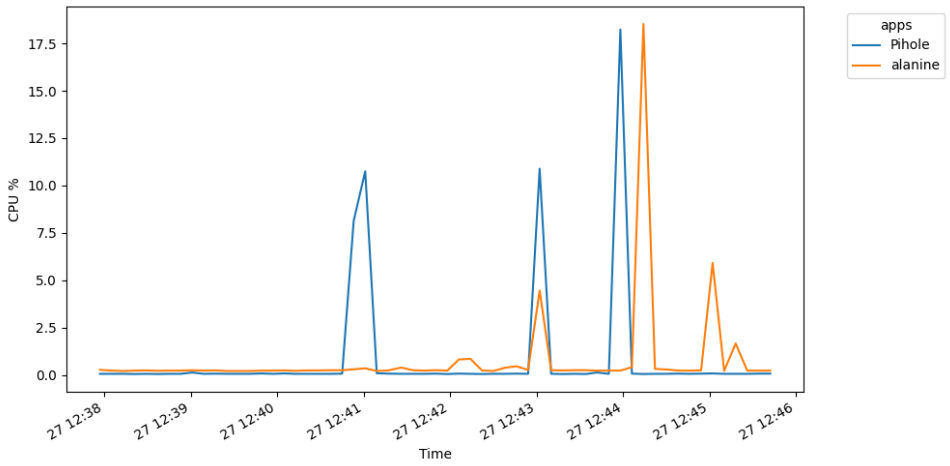
Druhý scénář nastává při stejných podmínkách, akorát na *master* větvi. Stejně jako první scénář kód otestuje, vytvoří a nahraje produkční Docker obrazy a vytvoří *vydání* (*release*) nové verze.

Integrační testy

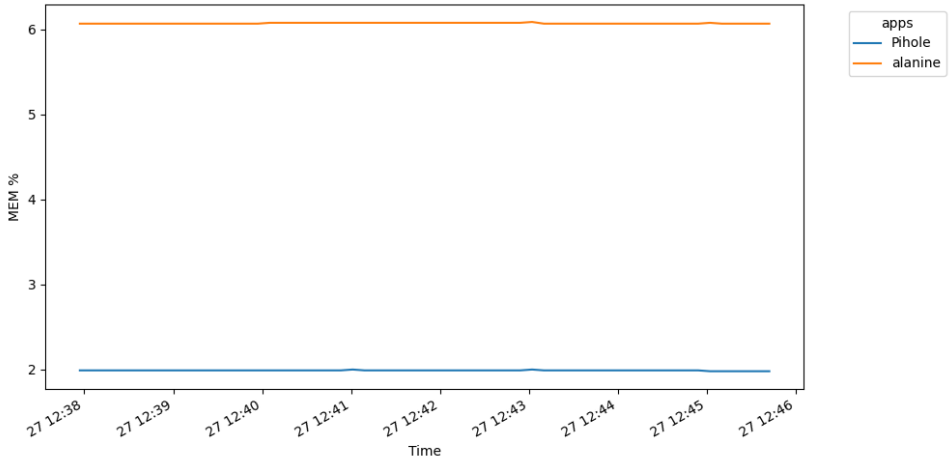
Je možné, že při nových verzích *Pi-hole* dojde ke změně *CLI* rozhraní Pi-hole. Aby byla tato změna odchycena, byla vytvořena sada integračních testů. Tyto testy vždy zavolají *REST* endpoint, a zkontrolují jestli server nevrací chybu a zda se správně změnil stav *Pi-hole*. Pokud by nějaký test neprošel, celý scénář se zastaví a nevznikne nekompatibilní obraz.



Obrázek 7.4: Graf využití paměti v procentech při uživatelských akcích



Obrázek 7.5: Graf využití CPU v procentech bez uživatelských akcí



Obrázek 7.6: Graf využití paměti v procentech bez uživatelských akcí

Kapitola 8

Uživatelské testování

Jelikož cílem práce bylo vytvořit alternativu k současným možnostem administrace *Pi-hole*, která by byla rychlejší a snadnější na použití, bylo potřeba provést uživatelské testování, které mělo za úkol odhalit, jak uživatelé s nástrojem pracují, a jestli práce splnila své cíle.

Proběhly dva oddělené způsoby uživatelského testování - testování pomocí uživatelského dotazníku a vzdálené testování. Každému druhu se věnuje jedna sekce.

V první sekci si představíme doplněk a jeho uživatelské rozhraní před uživatelským testováním. V následujících dvou sekcích si popíšeme oba způsoby testování a jejich výsledky. V poslední sekci následuje vyhodnocení a změny, které vyšly z uživatelského testování.

8.1 Počáteční stav

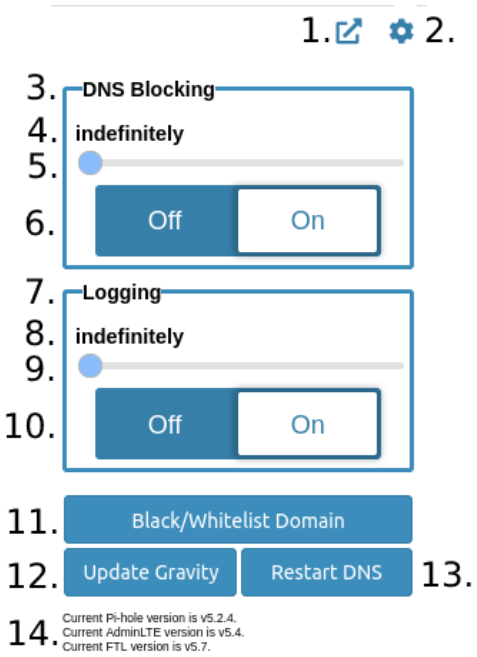
Počáteční stav UI (*User Interface*) doplňku s očíslovanými UI prvky je možno vidět na obrázku 8.1 a 8.2. Obrázek 8.3 obsahuje zprávu o výsledku uživatelské akce a vypnuté DNS blokování.

8.1.1 Významy UI prvků

1. Tlačítko na otevření Web Admin v nové záložce (*tab*)
2. Tlačítko pro otevření nastavení
3. Nadpis oddílu DNS blokování
4. Hodnota posuvníku
5. Posuvník (*slider*) pro nastavení času
6. Přepínač (*switch*) pro zapnutí / vypnutí DNS blokování
7. Nadpis oddílu Logování
8. Hodnota posuvníku
9. Posuvník (*slider*) pro nastavení času

10. Přepínač (*switch*) pro zapnutí / vypnutí logování
11. Přidání domény do blacklistu či whitelistu (otevře podmenu na přidávání domén do blacklistu či whitelistu)
12. Spuštění update gravity
13. Restart DNS serveru
14. Verze Pi-hole modulů
15. Tlačítko na návrat do hlavního kontrolního panelu
16. Označení textového pole
17. Textové pole pro doménu
18. Přepínač mezi tvary domény
19. Přidání domény do whitelistu
20. Přidání domény do blacklistu

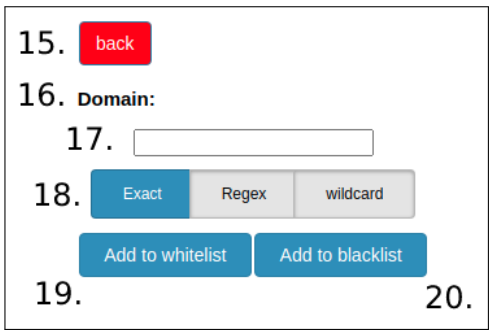
Tento vzhled a funkčnost byla testována v uživatelském testování.



Obrázek 8.1: Hlavní kontrolní panel s očíslovanými UI prvky.

8.2 Uživatelský dotazník

Jako první proběhlo testování pomocí dotazníku, který uživatelé Pi-hole měli vyplnit po instalaci a vyzkoušení doplňku.



Obrázek 8.2: Podmenu na přidávání domén do blacklistu či whitelistu s očíslovanými UI prvky.

8.2.1 Organizace

Spolu s vydáním doplňku byl vytvořen online dotazník pomocí nástroje *Google Forms*, který obsahoval sedmnáct otázek. Tento dotazník byl zmiňován v každém doprovodném textu, a byla zdůvodněna jeho důležitost. Také byl na něj umístěn odkaz v otevřeném popupu. Sběr odpovědí probíhal v období od 30.3.2021 až 23.4.2021.

Seznam otázek a odpovědí se nachází v příloze D.1.

8.2.2 Výsledky

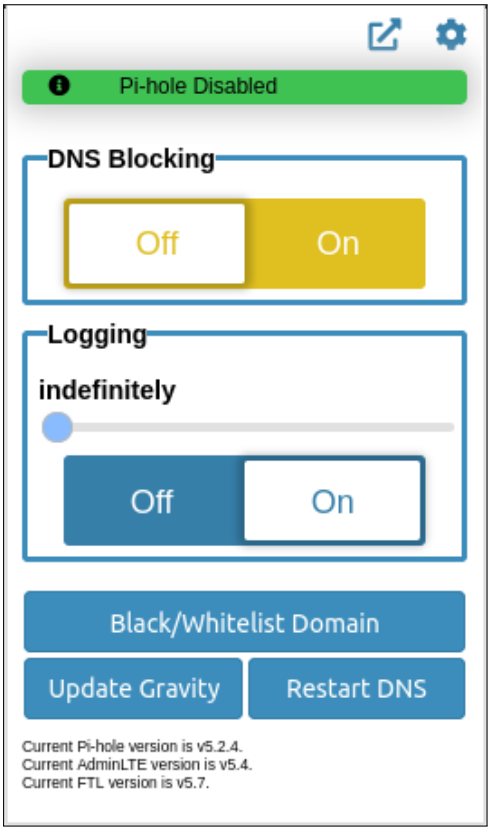
Vzhledem k nízkému počtu odpovědí (2) není možné získat statisticky významné výsledky. Obě dvě odpovědi se ale shodují, že ovládání přes doplněk je rychlejší a jednodušší než oba dva zbývající způsoby ovládání. Také udělili doplňku vysokou celkovou známku. Dobrou zprávou je absence chyb jak při používání, tak při instalaci.

8.3 Vzdálené testování

Po neúspěchu s uživatelským dotazníkem proběhlo vzdálené testování, které již relevantní výsledky dodalo.

8.3.1 Organizace

Původním plánem bylo provést osobní uživatelské testování, ale kvůli restrikcím způsobeným pandemií COVID-19 bylo provedeno vzdálené uživatelské testování. Testování probíhalo pomocí dvou programů - *Teamviewer* a *Microsoft Teams*, pomocí kterých testeři ovládali doplněk, který běžel na mém počítači. Byly také využity programy *Discord* a *Google Meet* pro hlasovou komunikaci.



Obrázek 8.3: Hlavní kontrolní panel se zprávou o výsledku akce uživatele a vypnutým DNS blokováním.

8.3.2 Průběh

Nejdříve proběhla krátká teoretická příprava, ve které byl vysvětlen princip *Pi-hole* a základní pojmy potřebné ke zvládnutí testovacích scénářů. Každý uživatel prošel třemi scénáři, které reflektovaly nejčastější způsoby použití doplňku. Každý scénář provedli jak pomocí doplňku, tak pomocí webového rozhraní (*Web Admin*). Následná diskuze po každém scénáři se zaměřovala na rychlost, srozumitelnost UI a porovnání obou způsobů. Předpokládaná délka testování byla kolem patnácti minut, ve skutečnosti testování trvala přes 30 minut.

8.3.3 Výběr testerů

Jelikož instalace a provoz *Pi-hole* vyžaduje konkrétní znalosti z IT a vlastní infrastrukturu, předpokládám, že mezi uživateli *Pi-hole* převažují osoby, které v IT pracují či se o něj zajímají. Proto tři z pěti vybraných testerů jsou studenti FEL ČVUT, kteří mají nadprůměrné znalosti o IT. Zbývající dva uživatelé jsou studenti netechnických vysokých škol, ti zastupují běžné uživatele.

8.3.4 Scénáře

Pro testování byly vytvořeny tři jednoduché scénáře.

1. Chceš se kouknout na svůj oblíbený seriál, který je dostupný online. Po načtení stránky ti přehrávač ukazuje hlášku *"Máte zapnutý adblock, pokud chcete pokračovat, vypněte jej."*. Vypni DNS blokování na 20 minut.
2. Druhý scénář měl dvě rozdílné varianty, které ale měly velmi podobný průběh:
   a. Neustále na tebe vyskakuje reklama z domény "reklama.cz". Přidej ji do blacklistu.
   b. Chceš navštívit internetové stránky "obchod.cz". po zadání jejich adresy do adresního řádku se ale nenačtou, místo toho na tebe vyskočí chybová stránka prohlížeče. Máš podezření, že je Pi-hole blokuje. Přidej doménu do whitelistu.
3. Chceš pro svou drahou polovičku koupit dárek online, ale nechceš po sobě zanechat žádné stopy. Víš, že *Pi-hole* ukládá všechny DNS dotazy, které by tě mohly prozradit. Vypni logování na neurčito.

8.3.5 Výsledky

Ze vzdáleného testování vyšlo mnoho důležitých poznatků. V následujících odstavcích jsou popsané zpracované výsledky, které vychází z odpovědí testerů[1].

Obecné připomínky

Hlavním nedostatkem,na kterém se shodli všichni testeři, byly přepínače (*prvky 6,10*). Tři odpověděli na otázku *"Je pro Vás na první pohled zřejmé jestli bylo DNS blokování zapnuté / vypnuté ?"* záporně. Dva sice odpověděli že ano, ale že si myslí, že většina uživatelů s tím bude mít problém. Testeři uvedli, že jak barevné schéma (modrá a oranžová), tak vzhled přepínače nejsou srozumitelné. Testery například mátlo to, že v zapnutém stavu je modře podbarvené *"Off"* a *"On"* je bíle podbarvené.

Dále se vyskytovaly jen ojedinělé připomínky. Například, že ikonka pro otevření *Web Admin* (*prvek 1*) je nepochopitelná. Jiný tester uvedl, že u tlačítek *11,12* a *13* je bílá na modré špatně čitelná.

První scénář

První scénář měl u všech testerů prakticky stejný průběh. Nejdříve vypnuli blokování na neurčito, a poté si až všimli posuvníku kterým lze nastavovat čas.

[1] Nezpracované výsledky je možné nalézt na přiloženém nosiči, či na tomto odkazu:`https://docs.google.com/spreadsheets/d/1AFFtk9NL__UnCuT7SwvbYQi_ZkfXMXG7jS6A4Q5sDio/edit?usp=sharing`

Toto bylo podle testerů způsobeno nejasnou příslušností časové hodnoty posuvníku, posuvníku a přepínače (*prvky 4,5,6*) k sobě, a nejasností, že tyto prvky mají něco společného s časem. Jako vyřešení druhého problému dva testeři navrhli umístění ikonky času k hodnotě posuvníku (*prvek 4*).

Dále testeři postrádali potvrzení, že blokování vypnuli na určitý čas. Ať již nějaký odpočet, či jen potvrzení ve zprávě od doplňku (ve zprávě není uváděn čas, na který bylo blokování vypnuto).

Dva testeři vznesli problém s pořadím kroků. V současné době se nejdříve na posuvníku nastaví čas, na který má být DNS blokování vypnuto a až poté se přepínačem vypne. Podle dvou testerů by bylo přirozenější DNS blokování vypnout, a až poté nastavit na jak dlouho.

V porovnání s *Web Admin* doplněk jasně vyhrál v rychlosti i jednoduchosti.

Druhý scénář

Druhý testovací scénář proběhl s méně výhradami než první. Jediná výtka u jednoho testera byla, že by tlačítko na přidání domény do blacklistu či whitelistu (*prvek 11*) rozdělil na dvě, aby poté v podmenu místo dvou tlačítek *19* a *20* bylo jen jedno. Vlastně jde o otočení pozice volby seznamu, kam má být doména přidána - nyní je seznam volen na konci procesu, a tester by si přál volbu na začátku procesu.

V rámci scénáře jsem pozoroval i reakci na přepínač *18* (a na stejné nastavení ve *Web Admin*), na kterém se nachází pojmy, které nebyly v teoretické přípravě vysvětleny. Většina testerů tento přepínač kvůli neznalosti pojmů přeskočila, což byl chtěný výsledek, přepínač byl ve výchozí podobě nastaven správně. Toto chování je totiž důležité, aby byl doplněk snadno srozumitelný i pro uživatele s horším porozuměním IT. Pokud totiž uživatel nezná pojmy jako *Regex*, je pro něj exaktní forma domény nejjednodušší.

V porovnání s *Web Admin* je podle testerů doplněk rychlejší, ale jen pokud se v něm uživatel nenachází. V otázce jednoduchosti jsou testeři nejednotní - dva uvádí že nastejno, a tři, že doplněk je jednodušší.

Třetí scénář

Třetí scénář proběhl u všech velmi rychle - i díky podobnosti ovládacích prvků s prvním scénářem. Jediná výtka u jednoho testera byla, že pochopil, že přepínač ovládá logování naimplementovaného serveru, ne Pi-hole.

V porovnání s *Web Admin* je podle testerů doplněk jasně jednodušší i rychlejší. I proto, že u *Web Admin* je zapnutí/ vypnutí logování schované v nastavení, kdežto v doplňku je toto nastavení na prominentním místě.

Porovnání s pihole command

Porovnání `pihole command` s ostatními způsoby ovládání proběhlo jen u jednoho scénáře u dvou testerů. U obou testerů bylo ověřeno, že ovládání pomocí `pihole command` je zásadně pomalejší, a i nebezpečnější, než u ostatních způ-

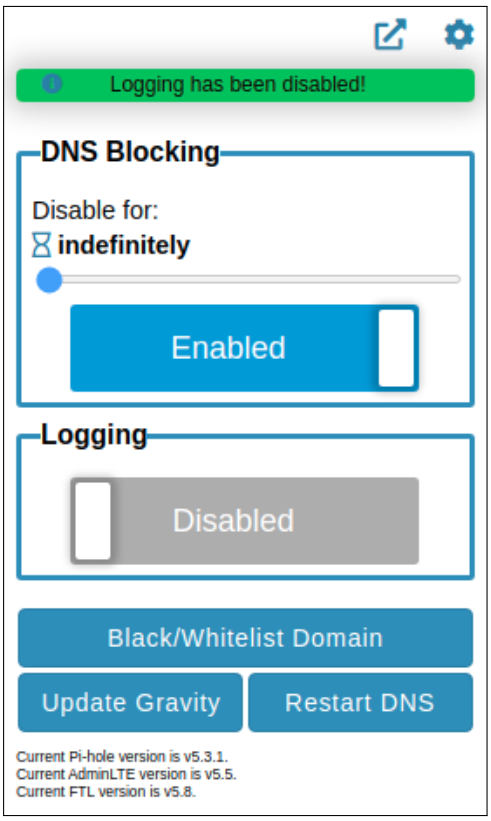
sobů - oba dva testeři při vypnutí logování smazali omylem všechny záznamy. Po těchto zkušenostech bylo u dalších testerů `pihole command` přeskočeno.

8.4 Zapracování

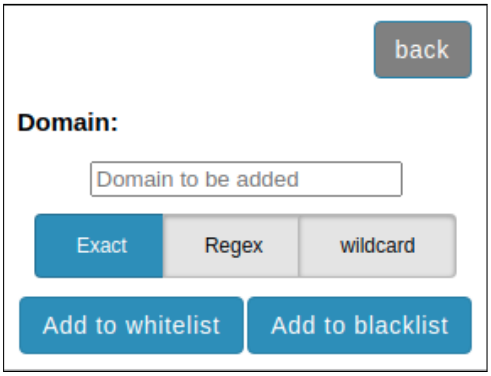
Důležitou součástí uživatelského testování je i zapracování výsledků. Změny byly zapracovány ihned po vypracování výsledků. Všechny změny, které se týkaly hodnoty posuvníku, posuvníku i přepínače pro DNS blokování (*4,5,6*), byly přeneseny i na totožné prvky v sekci zapnutí / vypnutí logování (*prvky 8,9,10*) kvůli jejich vizuální shodnosti. Novou podobu doplňku je možné vidět na obrázcích 8.4 a 8.5.

8.4.1 Seznam změn

- Kompletní vizuální změna přepínačů
- Přidání časového údaje do zprávy o vypnutí/ zapnutí DNS blokování / logování
- Přidání ikony času k časové hodnotě posuvníku
- Přidání označení *"Disable for:"* nad časové hodnoty posuvníku
- Zvětšení textu nadpisů
- Zvětšení mezer mezi písmeny v tlačítkách *11,12,13,19,20*
- Přesunutí tlačítka zpět (15) v podmenu na pravou stranu a upravení jeho barvy
- Přidání *placeholder* do textového vstupu (17)



Obrázek 8.4: Hlavní kontrolní panel se zapracovanými změnami.



Obrázek 8.5: Podmenu na přidávání domén do blacklistu či whitelistu se zapracovanými změnami.

Kapitola 9
Závěr

Cílem této práce byla implementace ovládání lokální instance *Pi-hole* pomocí doplňku do prohlížeče.

Nejprve byly popsány možnosti administrace Pi-hole pomocí CLI rozhraní a pomocí webového rozhraní. V této sekci byl kladen důraz na CLI rozhraní, na jeho možnosti a způsoby použití.

V následující sekci byly popsány principy tvorby doplňků do prohlížeče. Byly popsány jednotlivé moduly, ze kterých se doplněk skládá, byla vysvětleny možnosti komunikace mezi nimi. Velká část této kapitoly se zabývala možnostmi použití jednoho doplňku ve více prohlížečích.

V návrhu řešení se poprvé objevila potřeba nutnosti implementace vlastního serveru. Bylo navrženo základní schéma komunikace. Byly porovnány různé technologie na implementace serveru, a byla vybrána dvojice *Java* - *Quarkus*. Toto rozhodnutí značně ovlivnilo následující dvě kapitoly - Implementaci a Nasazení.

V implementační části byla popsána implementace obou částí. U obou částí byly popsány nástroje pro vývoj, architektura každé části, a poté se kapitola věnovala popisu zajímavějších částí implementace. Také byly porovnány frameworky *Quarkus* a *Spring Boot*.

Kapitola nasazení se věnovala rozšíření projektu mezi uživatele. Popisuje a srovnává průběh distribuce doplňku pomocí obchodů s doplňky. Kapitola se také věnuje distribuci serverové části, která byla poznamenána chybou v návrhu.

Důležitou součástí celé práce bylo uživatelské testování, jemuž je věnována jedna kapitola. Tato kapitola popisuje dvě na sobě nezávislé metody uživatelského testování. Popisuje jejich organizaci, jejich výsledky i změny, které byly implementovány do doplňku.

9.1 Zhodnocení cílů práce

Jak již bylo řečeno, cílem této práce bylo vytvořit doplněk, který by umožnil snadné a rychlé ovládání množiny základních příkazů. Důležitá byla i jednoduchost instalace. Podle výsledků uživatelského průzkumu byly splněny oba dva cíle. Všichni testeři při třech scénářích uvedli, že ovládání přes doplněk je rychlejší a jednodušší než ovládání pomocí webového rozhraní. Dalším důleži-

tým požadavkem byla snadnost instalace serverové části, což bylo splněno tím, že instalace přesně kopíruje instalaci *Pi-hole*. Neméně důležitým bodem byla i nenáročnost na výpočetní prostředky. Tohoto bylo také dosaženo, řešení je nenáročné na prostředky, při používání je dokonce méně náročné než webové rozhraní.

9.2 Budoucnost projektu

Budoucnost podobných projektů je vždy nejistá. Momentálně vše závisí na vůli uživatelů, pokud budou tento projekt využívat, bude pro mě mít cenu jej nadále vyvíjet a udržovat.

Určitě plánuji po následující půlrok projekt aktivně udržovat a přidávat nové funkce. Od komunity jsem zaznamenal poptávku po ovládání více instancí najednou či základní autorizaci a autentizaci.

Příloha A

Literatura a zdroje

1. NETMONITOR. *Internet během karantény dramaticky posílil. Lidé se tam naučili trávit o 50 % více času* [online]. Praha: Sdružení pro internetový rozvoj v České republice, z.s.p.o. (SPIR), 2020 [cit. 2020-11-23]. Dostupné z: `https://www.netmonitor.cz/internet-behem-karanteny-dramaticky-posilil-lide-se-tam-naucili-travit-o-50-vice-casu`.
2. CLARK, Douglas; WEIR, Corinne. *Google's US Ad Revenues to Drop for the First Time* [online]. New York: Insider Intelligence, 2020 [cit. 2020-11-23]. Dostupné z: `https://www.emarketer.com/newsroom/index.php/google-ad-revenues-to-drop-for-the-first-time/`.
3. DELANEY, John. *Implement host throttling for heavy ad intervention* [online]. 2019 [cit. 2020-11-20]. Dostupné z: `https://chromium-review.googlesource.com/c/chromium/src/+/1645142/11`.
4. SALMELA, Jacob. *Block Millions Of Ads Network-wide With A Raspberry Pi-hole 2.0* [online]. Jacob Salmela, 2015 [cit. 2020-11-23]. Dostupné z: `https://jacobsalmela.com/2015/06/16/block-millions-ads-network-wide-with-a-raspberry-pi-hole-2-0/`.
5. MONK, Simon. *Raspberry Pi Cookbook: Software and Hardware Problems and Solutions*. 3rd ed. Sebastopol: O'Reilly Media, 2019. ISBN 978-1492043225.
6. KRČMÁŘ, Petr. *Chrome vs. Chromium: kdo je kdo a který z nich chcete* [online]. Praha: Internet Info, 2020 [cit. 2020-11-27]. Dostupné z: `www.root.cz/clanky/chrome-vs-chromium-kdo-je-kdo-a-ktery-z-nich-chcete/`.
7. PONGE, Julien. *The GraalVM frenzy* [online]. A Medium Corporation, 2020 [cit. 2020-11-27]. Dostupné z: `https://jponge.medium.com/the-graalvm-frenzy-f54257f5932c`.
8. FONG, Jenny. *Are Containers Replacing Virtual Machines?* [Online]. Palo Alto, CA: Docker, Inc., 2018 [cit. 2021-04-20]. Dostupné z: `https://www.docker.com/blog/containers-replacing-virtual-machines/`.

9. DROUET, Jeremie. *Multi-arch build and images, the simple way* [online]. Palo Alto, CA: Docker, Inc., 2020 [cit. 2021-05-04]. Dostupné z: https://www.docker.com/blog/multi-arch-build-and-images-the-simple-way/.
10. DAVIS, Jennifer; DANIELS, Ryn; ANDERSON, Brian; DEMAREST, Rebecca. *Effective DevOps*. Third Release. Sebastopol, CA: O'Reilly Media, Inc., 2016. ISBN 9781491926307.
11. BOLMÉR, Percy. *GitHub Actions In Action* [online]. A Medium Corporation, 2021 [cit. 2021-04-26]. Dostupné z: https://betterprogramming.pub/github-actions-in-action-3b10083cc700.
12. VILLALOBOS, Jorge. *How to develop a Firefox extension* [online]. Mountain View: Mozilla Foundation, 2014 [cit. 2020-11-24]. Dostupné z: https://blog.mozilla.org/addons/2014/06/05/how-to-develop-firefox-extension/.
13. MOZILLA. *Sharing objects with page scripts* [online]. Mountain View, CA: Mozilla Foundation, 2018 [cit. 2020-11-24]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Sharing_objects_with_page_scripts.
14. W3COUNTER. *Browser & Platform Market Share: October 2020* [online]. Raleigh NC: Awio Web Services LLC, 2020 [cit. 2020-11-25]. Dostupné z: https://www.w3counter.com/globalstats.php?year=2020%5C&month=10.
15. DILLET, Romain. *Apple will let you port Google Chrome extensions to Safari* [online]. Verizon Media, 2020 [cit. 2020-11-25]. Dostupné z: https://techcrunch.com/2020/06/25/apple-will-let-you-port-google-chrome-extensions-to-safari/.
16. STRAUCH, Maximilian. *A real-world comparison of web frameworks with a focus on NodeJS* [online]. A Medium Corporation, 2020 [cit. 2020-11-26]. Dostupné z: https://medium.com/swlh/a-real-world-comparison-of-web-frameworks-with-a-focus-on-nodejs-c00efe1df7ca.
17. TECHEMPOWER. *Web Framework Benchmarks* [online]. El Segundo, CA: TechEmpower, Inc., 2020 [cit. 2020-11-26]. Dostupné z: https://www.techempower.com/benchmarks/.
18. W3TECHS. *W3Techs: World Wide Web Technology Surveys* [online]. Maria Enzersdorf: Q-Success, c2009–2020 [cit. 2020-11-27]. Dostupné z: w3techs.com.
19. HACKEREARTH. *Behind the code: The 2020 HackerEarth Developer Survey* [online]. San Francisco: HackerEarth, 2020 [cit. 2020-11-27]. Dostupné z: https://www.hackerearth.com/recruit/developer-survey/.

20. HOWARTH, Jesse. *Why Discord is switching from Go to Rust* [online]. Discord Inc., 2O2O [cit. 2020-11-27]. Dostupné z: `https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f`.

21. JAAIDI, Younes. *ReST APIs: Best Practices & Security* [online]. Lyon: Wishtack [cit. 2020-11-29]. Dostupné z: `https://blog.wishtack.com/rest-apis-best-practices-and-security/`.

Příloha B

Instalace

Samotná instalace nástroje *Pi-hole* je poměrně jednoduchá, ale jeho začlenění do síťového provozu je nutno provádět s dávkou opatrnosti.

B.1 Před samotnou instalací

Pi-hole potřebuje být spuštěna na vhodném zařízení, které bude v provozu pokaždé, když budeme potřebovat komunikovat se zařízeními mimo lokální síť. Pokud totiž bude síťový DNS server nefunkční, většina služeb na internetu může být omezena či přímo nefunkční.

Pro pohodlí uživatelů sítě je tedy vhodné, aby zařízení, na kterém je spuštěna Pi-hole, běželo prakticky nepřetržitě. Pro provozování Pi-hole je tedy vhodný domácí server, např. *Raspberry Pi* (2.2) (od kterého pochází název **Pi**-hole).

Pi-hole je možno nainstalovat na server mimo lokální síť, pro kterou bude poskytovat své služby (například na *Google cloud platform* spolu s *VPN*[1] (*Virtual private network*), toto je ale mimo rozsah tohoto projektu.

B.2 Instalace Pi-hole

Instalace *Pi-hole* je možná dvěma způsoby. Ovšem ne každý operační systém podporuje oba způsoby, viz B.1.

B.3 Docker install

Stačí spustit `docker run` s oficiálním Docker obrazem, [2](2.7.1) a s vhodně zvolenými parametry. Je možné také jednoduše nastavit automatické spuštění kontejneru po zapnutí systému.

[1] Podrobný návod na zprovoznění této instalace je například zde: `https://bit.ly/342Y3L4` [cit. 2020-12-11]

[2] K dispozici zde : `github.com/pi-hole/docker-pi-hole` [cit. 2020-11-7]

	Windows	Linux	MacOS
Docker install	Ano (Pro/Enteprise edice)	Ano	Ano
Bash install	Ne	Ano	Ne

Tabulka B.1: Možnosti instalace Pi-hole v závislosti na operačním systému

B.4 Bash install

Je možné instalovat *Pi-hole* jen pomocí jednoho *bash* příkazu `curl -sSL https://install.pi-hole.net | bash`. Snadný způsob, ale z bezpečnostního hlediska nevhodný - tzv. *curl bash piping*.[3] Vhodnějším způsobem je naklonování / stažení repozitáře z `https://github.com/pi-hole/pi-hole` [cit. 2020-12-13], zkontrolování a následné spuštění instalačního skriptu (`basic-install.sh`). Instalační skript nás poté provede základním nastavením *Pi-hole*.

B.5 Nastavení IP adresy

Pro správné fungování je potřeba systému, na kterém běží *Pi-hole*, nastavit pevnou IP adresu, ať již přímo v systému, či ji přiřadit *DHCP* (*Dynamic Host Configuration Protocol*) rezervaci v DHCP serveru, v domácnosti se DHCP server nejčastěji nachází v síťovém routeru.

Na obrázku B.1 je možné vidět nastavení statické adresy IP v *DHCP* serveru.

B.6 Přesměrování DNS serveru

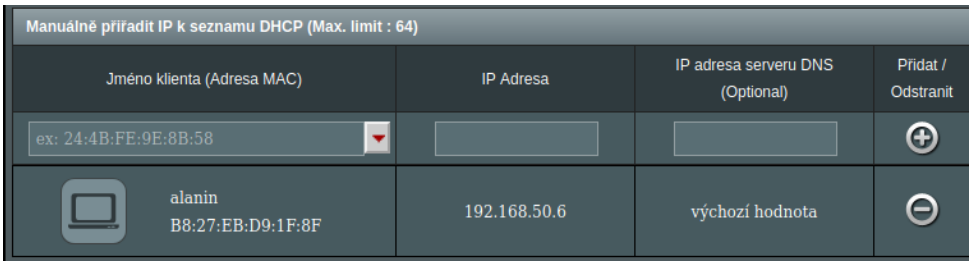
Dalším krokem je nastavení *Pi-hole* jako DNS serveru. Je možné každému zařízení v síti manuálně nastavit (či nenastavit) adresu systému, kde je spuštěna *Pi-hole*. Toto ale je potřeba opakovat pro každé nové zařízení v síti. Druhým způsobem je nastavení *Pi-hole* jako DNS serveru v routeru.

Pokud nechceme, aby nějaké zařízení využívalo Pi-Hole jako DNS serveru, stačí v síťovém nastavení zařízení nastavit jiný DNS server (či v případě prvního způsobu nenastavovat *Pi-hole* jako DNS server).

B.7 Přidání Adlist seznamů

Důležitou součástí *Pi-hole* jsou veřejně dostupné seznamy domén k blokaci - adlists, podrobněji jsou popsány v sekci 2.1.4. Můžeme je přidat přes webové rozhraní *Pi-hole* (či příkaz `pihole`).

[3] *Curl bash piping* je vykonávání *bash* příkazů, při kterém vykonáváme kód, který jsme předtím neměli šanci zkontrolovat kvůli škodlivému chování, více vysvětleno zde: `https://0x46.net/thoughts/2019/04/27/piping-curl-to-shell/`[cit. 2020-11-8]



Obrázek B.1: Nastavení statické adresy v DHCP serveru, router Asus RT-AC51

Příloha C

Ukázky kódu

C.0.1 manifest.json

Příklad manifestu doplňku do prohlížeče (4.3). [1]

```
{
  "browser_specific_settings": {
    "gecko": {
      "id": "addon@example.com",
      "strict_min_version": "42.0"
    }
  },

  "background": {
    "scripts": ["jquery.js", "my-background.js"],
  },

  "browser_action": {
    "default_icon": {
      "19": "button/geo-19.png",
      "38": "button/geo-38.png"
    },
    "default_title": "Whereami?",
    "default_popup": "popup/geo.html"
  },

  "commands": {
    "toggle-feature": {
      "suggested_key": {
        "default": "Ctrl+Shift+Y",
        "linux": "Ctrl+Shift+U"
      },
      "description": "Send a 'toggle-feature' event"
    }
  },

  "content_security_policy": "script-src 'self' https://example.com; object-src 'self
      '",

  "content_scripts": [
    {
      "exclude_matches": ["*://developer.mozilla.org/*"],
      "matches": ["*://*.mozilla.org/*"],
      "js": ["borderify.js"]
    }
```

[1] Autor Mozilla foundation, přebráno z : `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json` [cit. 2020-11-19]

```
  ],

  "default_locale": "en",

  "description": "...",

  "icons": {
    "48": "icon.png",
    "96": "icon@2x.png"
  },

  "manifest_version": 2,

  "name": "...",

  "page_action": {
    "default_icon": {
      "19": "button/geo-19.png",
      "38": "button/geo-38.png"
    },
    "default_title": "Whereami?",
    "default_popup": "popup/geo.html"
  },

  "permissions": ["webNavigation"],

  "version": "0.1",

  "user_scripts": {
    "api_script": "apiscript.js",
  },

  "web_accessible_resources": ["images/my-image.png"]
}
```

C.0.2 Docker manifest

Příklad platform-specific Docker manifestu (2.7.3).

```
{
       "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
       "schemaVersion": 2,
       "config": {
              "mediaType": "application/vnd.docker.container.image.v1+json",
              "digest": "sha256:62
                   d656868b3de256da60516fea2a932a8dc9fb888260901e7fec2b306a2e7dfe",
              "size": 8948
       },
       "layers": [
              {
                     "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
                          ",
                     "digest": "sha256:9159
                          b6bb943191d8630069e0441d3638c9b053fff75702992c69f9c341a26b35
                          ",
                     "size": 47362865
              },
              {
                     "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
                          ",
                     "digest": "sha256:010046
                          f0b40e08c144c63d54a65ddf1ed3dad0cac7fbc879fb1b37e1dd5e7efe
                          ",
                     "size": 1901
```

```
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
        "digest": "sha256:956067455
            ce880a5a14a7f42efdfa1ed49dc49f893bb760d5c8dcfb906fadc63",
        "size": 157
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
        "digest": "sha256:
            b26a0a9087a921eb09bbc4d243ae6f4ee7ed9ea24bc22f0aaca6436203bcf6fd
            ",
        "size": 122847171
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
        "digest": "sha256:
            bc2a1c51f34f751742ca87040f4b2ee2cd9635576c6211ca644546a4767b40b1
            ",
        "size": 1658
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
        "digest": "sha256:38
            fcc8c10af7aead0f0b1a93c5f5a87a645319001c704826b4708156e3eb93f7
            ",
        "size": 481
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
        "digest": "sha256:
            fea6f10554cefcea32180e31f8fc9a33f5e24e800ae78f9a0c31e2cafd370449
            ",
        "size": 2283
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
        "digest": "sha256:651
            d68f3083f16907ed431ff4c93814ae0a6776a33e497d60dcdeb4cd5e2686b
            ",
        "size": 4182
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
        "digest": "sha256:70188967755
            d66959a663bd40d0d6ecc3060f87568ad799b5d3ef8890cc19ed6",
        "size": 162
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
        "digest": "sha256:
            d7f129f6c8deba4e7081ec85295edd3fcbd49ebe00510676b5f00d943ece986c
            ",
        "size": 75794895
},
{
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
            ",
```

```
                "digest": "sha256:8
                    f4f92b3aaf51bc552ab904bf1098abd40c05828118147e42812d1c5b2b0111f
                    ",
                "size": 525
            },
            {
                "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
                    ",
                "digest": "sha256:6
                    c5731e204dd53022e0c8eb6b121ea1295bdb6cd90d90859b5584f43833d5039
                    ",
                "size": 15424427
            },
            {
                "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip
                    ",
                "digest": "sha256:54
                    cc44057e5ab879855707373f00842e89fb62d5b1ccebbdf45ce7512a5750e1
                    ",
                "size": 373
            }
        ]
}
```

C.0.3 Docker multiarch manifest

Příklad multiarch Docker manifestu (2.7.3).

```
{
   "schemaVersion": 2,
   "mediaType": "application/vnd.docker.distribution.manifest.list.v2+json",
   "manifests": [
      {
         "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
         "size": 3872,
         "digest": "sha256:
              ed47b818237d1fe562b70f63e117156dd65d8b2c784e6465e7db88bac8624d1f",
         "platform": {
            "architecture": "amd64",
            "os": "linux"
         }
      },
      {
         "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
         "size": 2826,
         "digest": "sha256:49
              d00854cc4197c71e08247084ed555bcd273f8b5506c077b4e720311ecf96bc",
         "platform": {
            "architecture": "arm64",
            "os": "linux"
         }
      },
      {
         "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
         "size": 3033,
         "digest": "sha256:66
              a16493fccac45af25a96eb38a2f9e35818dd0bbab869d8c70a083eb2fd0a99",
         "platform": {
            "architecture": "arm",
            "os": "linux",
            "variant": "v7"
         }
      }
   ]
}
```

C.0.4 Příklad Dockerfile

Příklad Dockerfile Java obrazu z kapitoly 7.

```
# image we extend
FROM pihole/pihole:latest


# install java
RUN mkdir -p /usr/share/man/man1 /usr/share/man/man2
RUN apt-get update && apt-get install -y openjdk-11-jre-headless && rm -rf /tmp/* /
    var/cache/apk/*

# set enviroment
RUN echo "export JAVA_HOME=/usr/lib/jvm/java-8-oracle" >> ~/.bashrc

# copy app files
COPY target/quarkus-app app-folder

# app lifecycle files
ADD s6-fastjar/ /etc/services.d


# tell docker we listen on port 8221
EXPOSE 8221
```

Příloha D

Uživatelský průzkum

D.1 Uživatelský dotazník

D.1.1 Otázky

Seznam otázek, které byly v online formuláři. Pokud není stanoveno jinak, formát odpovědí byl textový.

1. How difficult was the installation of Alanine server for you? (Hodnocení 1-5)
2. How would you rate the "friendliness"of UI? (Hodnocení 1-5)
3. How easy is it for you to use Alanine plugin? (Hodnocení 1-5)
4. How would you compare the speed and ease of administration while using Alanine plugin versus using Web admin ? (Hodnocení 1-5)
5. How would you compare the speed and ease of administration while using Alanine plugin versus using bash pihole command? (Hodnocení 1-5)
6. How would you grade Alanine as a simple and quick way to control your Pi-hole ? (Hodnocení 1-5)
7. If you want, you can write why you gave this grade here!
8. Can you easily say if DNS blocking is enabled or disabled ? Are the two main switches comprehensible ? (Yes/No)
9. Would you appreciate some kind of authorization ? So only a user that knows passwod is able to control Pi-hole via Alanine (Same as default Pi-hole password).
10. The Alanine server is a standalone REST server - you can control your Pi-hole from your own REST client. Would you consider using your own REST client instead of Alanine ? (Yes/No/Maybe)

11. We know that Pi-hole is usually running on not-so-powerfull hardware. Could you say how Alanine server affected your server ? And main specs of that system ?
12. Have you stumbled upon some difficulties or incorrect documentation while installing Alanine?
13. Have you stumbled upon any visual/functional bugs while using Alanine?
14. Do you see yourself using Alanine as your primary mean of administration ? Please summarize why yes/no in few sentences.
15. Last question - Do you have ANY comments on Alanine project or on this form?

D.1.2 Odpovědi

Odpovědi na uživatelský dotazník je možno najít v tabulce D.1.

Otázka/Odpověď		
1	1	1
2	1	1
3	1	2
4	1	2
5	1	1
6	1	1
7	It's so simple yet powerful and quick way to control pi-hole without constantly switching between tabs and enabling disabling etc..	
8	No	No
9	Probably the same password as pihole admin page ?	n
10	No	No
11	none, Raspberry Pi3B, 16GB Class10 Card	PI 4
12	Yes, my mistake not downloading from "raw.github"	no
13	No	no
14	Yes, for troubleshooting it is best option, would be great if it would show me for example last 10 blocked request or something similar.	yes
15	Maybe the ON/OFF switches in browser extension could be different color to clearly see if this is OFF or ON..	Awesome work !

Tabulka D.1: Tabulka odpovědí na uživatelský dotazník

Příloha E

Obsah přiloženého CD

Obsah CD přiloženého k práci má následující obsah:

```
/
├── source-code-plugin ..... Zdrojový kód doplňku
├── source-code-server ..... Zdrojový kód serveru
├── latex-source ..... Zdrojový Latex kód této práce
├── kolovjan-BP2021.pdf ..... Elektronická podoba této práce
└── user-testing
    └── user-testing-responses.xlsx ... Data z uživatelského testování
```