VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SLUŽBA PRO ARCHIVACI WEBOVÉHO OBSAHU

WEB CONTENT ARCHIVING SERVICE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ADAM MATUŠ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. LIBOR POLČÁK, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Matuš Adam**
Program: Informační technologie
Název: **Služba pro archivaci webového obsahu**
Web Content Archiving Service
Kategorie: Data mining

Zadání:

1. Nastudujte si problematiku analýzy a exportu webového provozu - protokol HTTP, HTTPS, internetové proxy a exportní formáty (zejména MHTML a MAFF).
2. Seznamte se s frameworky pro archivaci webu (např. Lemmiwinks).
3. Navrhněte službu a její API, která bude archivovat webový obsah s využitím frameworku Lemmiwinks.
4. Po konzultaci s vedoucím implementujte navrženou službu.
5. Otestujte a analyzujte výsledné řešení na reprezentativním vzorku stránek, se zaměřením na stránky z Deep webu.

Literatura:

- R. Fielding, "RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1", IETF, 1999.
- H. Lie, "The text/css Media Type", IETF, 1998.
- W3C, "Document Object Model (DOM)", online: http://www.w3.org/DOM/, 2011.
- Mozdev Community Organization Inc., "The MAFF Specification", online: http://maf.mozdev.org/maff-specification.html, 2014.
- L. Richardson, RESTful Web APIs, O'Reilly, 2013.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Polčák Libor, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. července 2019
Datum odevzdání: 31. července 2019
Datum schválení: 17. června 2019

Abstrakt

Tato bakalářská práce řeší návrh a implementaci webových služeb pro archivaci obsahu se zaměřením na stránky z deep webu. Zabývá se seznámením se s principy aplikačních protokolů HTTP a HTTPS a dále bude nastíněna architektura webových služeb. Čtenář bude seznámen s existujícím řešením frameworku pro archivaci a rekonstrukci webu Lemmiwinks. S využitím tohoto frameworku je navržen a implementován systém webových služeb, které lze využít pro archivaci zvoleného webového obsahu do formátu MAFF.

Abstract

This bachelor's thesis deals with the design of web microservices for web archiving, mainly focused on deep web pages. The HTTP and HTTPS protocols will be explained in detail as well as web services design and architecture. The reader will learn about web archiving frameworks, especially the Lemmiwinks framework. This framework will be used as a basis for web services designed for archiving into the MAFF archive.

Klíčová slova

web, služby, MAFF, MHTML, HTTP, REST, deep web

Keywords

web, services, MAFF, MHTML, HTTP, REST, deep web

Citace

MATUŠ, Adam. *Služba pro archivaci webového obsahu*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Libor Polčák, Ph.D.

Služba pro archivaci webového obsahu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Viliama Serečuna a Ing. Libora Polčáka, Ph.D.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Adam Matuš
29. července 2019

Poděkování

Chtěl bych poděkovat Ing. Viliamu Serečunovi a Ing. Liboru Polčákovi, Ph.D. za praktické rady.

Obsah

Kapitola 1

Úvod

Internet v současnosti dává možnost přistupovat k obrovskému množství informací, u kterých není zaručeno, že se v budoucnu nezmění nebo neztratí úplně. Proto se často využívají technologie pro zálohování a archivaci webového obsahu. Aplikací pro archivaci webových stránek existuje mnoho – ve formě softwarových frameworků či rozšíření do prohlížeče.

Trendem je implementovat aplikace jako webové služby, které jsou dostupné z Internetu a využívají běžné komunikační protokoly na webu. Výhodou webových služeb je snadná reprezentace dat pomocí serializace do textových formátů, jako je například JSON.

V této bakalářské práci jsem se zaměřil na transformaci archivačního frameworku Lemmiwinks, který vznikl v rámci diplomové práce Ing. Viliama Serečuna. Framework bude využit při implementaci dvou webových služeb - archivační a stahovací služby. Výhodou tohoto řešení je rozdělení zodpovědnosti, kdy složitější systém je rozdělen na menší části. K těmto službám je možnost přistupovat vzdáleně uživatelům pomocí webového prohlížeče nebo automatizovaným nástrojům a klientům pomocí REST aplikačního rozhraní. K popisu API bude použitá specifikace OpenAPI 3.0 a nástroj Swagger [19].

Na základě navrženého API budou služby implementované v jazyce Python 3.6 s využitím webového serveru Sanic. Stahovací služba bude implementovat přístup do sítě Tor.

Implementace bude následně vyzkoušena asynchronním testovacím klientem, který bude přistupovat k API archivační služby. Bude vyhodnocena rychlost archivace při zatížení a správnost fungování služeb. Dále bude testovaná archivace stránek z Deep webu s příklady snímků obrazovky archivovaných webů. Cílem bude ověřit kvalitu a přesnost vytvořených archivů.

Kapitola 2

Hypertextové protokoly

Tato kapitola se věnuje převážně protokolům HTTP a HTTPS. Je zde popsána jejich struktura a principy fungování v rámci Internetu. Dále budou nastíněny rozdíly mezi jejich verzemi. Čtenáři bude prezentováno několik příkladů komunikace a její zabezpečení.

2.1 Charakteristika protokolu HTTP

Hypertext Transfer Protocol, dále zkráceně HTTP, je bezstavový protokol aplikační vrstvy modelu ISO/OSI [15]. Byl navržen pro přenos dokumentů a hypertextových dat s možností rozšíření o další použití. Je postaven na komunikačním modelu klient-server, kde klient zahajuje komunikaci zasláním HTTP požadavku a server po přijetí žádosti pošle HTTP odpověď. HTTP je textový protokol, komunikace je tedy zobrazitelná v textově formě a většina přenášených informací je čitelná člověkem – v anglické literatuře existuje pojem *human-readable*.

Obvyklým textovým obsahem HTTP odpovědí jsou například dokumenty v jazyce HTML nebo skripty v jazyce JavaScript. Multimediální, binární nebo jiná netextová data je možné přenášet s použitím rozšíření MIME.

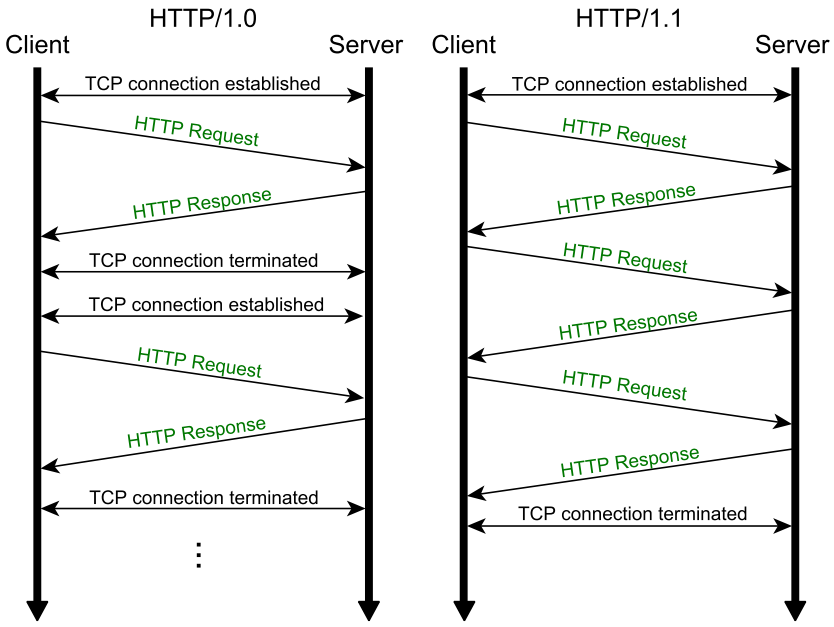
Protokol HTTP je bezstavový [15], takže požadavky jsou obsluhovány jednotlivě bez potřeby navazovat nebo si pamatovat historii požadavků. Každý požadavek musí plně specifikovat celou akci, aby mohl server korektně odpovědět. Existují stavové mechanismy, které ale nejsou součástí specifikace protokolu HTTP.

Princip komunikace

O navázání spojení se stará transportní vrstva s protokolem TCP. Standardně se HTTP využívá na výchozím portu 80, specifikace protokolu HTTP ale nebrání použití jiného transportního protokolu, který by měl podobné vlastnosti. Od transportní vrstvy se především předpokládá spolehlivý přenos, který protokol TCP zaručuje [14].

V rámci vytvořeného spojení se dále komunikuje na aplikační vrstvě formou HTTP zpráv. Zde je důležitý rozdíl mezi verzemi protokolů HTTP. Starší verze HTTP/1.0 dovoluje pouze jednu výměnu požadavku a odpovědi během spojení. Pro každou další konverzaci se spojení vytváří nové. Nevýhodami je velká režie a zátěž pro TCP congestion control. U novější verze HTTP/1.1 spojení existuje pro neomezený počet konverzací. Spojení se ukončuje až na žádost serveru nebo klienta. Obrázek 2.1 demonstruje rozdíl verzí.

Kromě HTTP/1.0 a HTTP/1.1 existuje novější standard HTTP/2, který řeší některé problémy zmíněných starších verzí protokolu. Pro zvýšení propustnosti HTTP/2 umožňuje



Obrázek 2.1: Sekvence požadavků HTTP 1.0 a HTTP 1.1

prokládání požadavků a odpovědí v rámci jednoho TCP spojení a používá efektivní kódování HTTP hlavičky. Další optimalizací je prioritizace požadavků, kde důležitější požadavky mají přednost ve zpracování [12]. HTTP/2 stále nemá většinovou podporu na webu[1].

Za zmínku stojí momentálně připravovaná verze HTTP/3 [1]. Tento navrhovaný standard bude využívat novější transportní protokol QUIC, který nahradí dosud používaný TCP.

2.2 Struktura protokolu

Základní datovou jednotkou protokolu HTTP je zpráva (message). Zpráva vytvořená klientem pro server se nazývá požadavek či dotaz (request) a zpráva vytvořená serverem pro klienta je odpověď (response). Každá zpráva začíná řádkem start-line. Dále následuje seznam hlaviček (headers) a tělo zprávy (body). Obsah částí protokolu se liší podle toho, zda se jedná o požadavek nebo o odpověď.

Jednotlivé elementy protokolu jsou oddělovány sekvencí CRLF, v textové reprezentaci jde o odřádkování. Ve znakové sadě ASCII této sekvenci náleží hodnoty `0x0d 0x0a`. Hlavičky a tělo zprávy jsou odděleny prázdným řádkem.

Identifikátor zdroje

Pro identifikaci zdroje na serveru se používá řetězec *Uniform Resource Identifier*, zkráceně URI. Zdrojem je obecně myšlen hypertextový dokument, multimediální obsah, služba nebo další podobné entity, které jsou serverem zpřístupněny pro dotazování. Obecný formát URI a příklad pro protokol HTTP je následující:

[1] HTTP/2 podporuje přibližně 31% webů – https://w3techs.com/technologies/details/ce-http2/all/all

<scheme>://<authority>/<path>?<query>#<fragment>
http://tools.ietf.org/html/rfc2616.txt

První povinná část je *schéma* (scheme) a je zde uveden komunikační protokol. Další části určují server, hierarchické umístění zdroje na serveru a informace dále upřesňující zdroj.

Formát požadavku

První řádek požadavku (request-line) se vždy skládá z názvu metody, identifikátoru zdroje a verze protokolu. Tyto položky jsou odděleny jednou mezerou a jsou očekávány v uvedeném pořadí. Verze protokolu je řetězec znaků, obvykle `HTTP/1.1` nebo starší standard `HTTP/1.0`.

Existuje celkem 9 typů metod požadavku. Název metody indikuje požadovanou akci a je povinně case-sensitive[2].

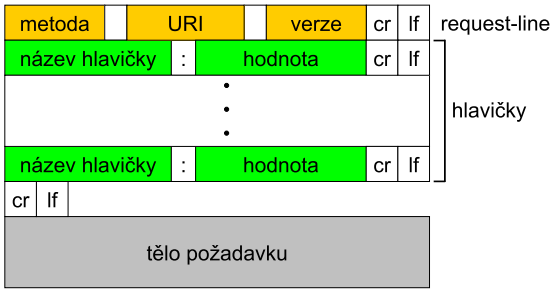
- GET metoda požaduje po serveru stáhnutí hypertextového zdroje, který se nalézá na dále uvedeném URI. Metoda byla navržena pro získávání dokumentů ze serveru, přesto ji lze nesprávně použít pro zasílání dat z formulářů od klienta. Metoda GET by správně měla být idempotentní. Stav zdroje se po zpracování požadavku nezmění.
- HEAD požaduje stejnou odpověď jako GET s tím rozdílem, že server vynechá tělo odpovědi. Metoda je vhodná pro zjištění dostupnosti zdroje a dalších informací obsažených v hlavičkách odpovědi, například čas vytvoření zdroje.
- POST slouží pro odeslání dat na server. Narozdíl od metody GET jsou data obsaženy v těle požadavku. Metoda POST není idempotentní.
- PUT odešle dokument zaslaný v těle zprávy na místo specifikované v URI. V případě existujícího dokumentu se nahradí novým.
- OPTIONS se dotáže serveru na možnosti komunikace pro konkrétní zdroj. Pro získání globálních možností serveru lze URI nahradit znakem hvězdičky.
- CONNECT metoda se pokusí zahájit obousměrné tunelované spojení. Požadavek se pošle HTTP Proxy serveru, který zprostředkuje komunikaci s cílovým serverem.
- DELETE požádá server o smazání zdroje na specifikovaném URI.
- TRACE je metoda používaná pro diagnostické účely a testování. Cílový server reflektuje přijatou zprávu zpět klientovi. Klient si může ověřit, zda požadavek dorazil s očekávaným obsahem.
- PATCH částečně modifikuje odkazovaný zdroj. Metoda umožňuje změnit část dokumentu bez nutnosti posílat celý dokument, jako tomu je u metody PUT.

Po prvním řádku následuje seznam hlaviček. Struktura hlavičky je vždy dvojce název a hodnota, tyto položky jsou oddělené dvojtečkou. Účelem hlaviček je blíže specifikovat požadavek. Existuje celá řada často využívaných hlaviček, níže uvedený seznam jsou pouze některé zajímavé hlavičky:

[2]Lexikální analýza je citlivá na velikost písmen. Řetězce "get"a "GET"jsou chápány jako rozdílné.

- **Cache-control** upravuje výchozí caching mechanismy pro HTTP konverzaci. Pomocí klíčových slov public, private a no-cache lze nastavit možnost kešování požadavku serverem, dále je možné modifikovat expirační a revalidační mechanismy.
- **Host** obsahuje doménové jméno dotazovaného serveru a číslo portu. Není-li port uveden, předpokládá se výchozí port 80. Tato hlavička je ve verzi HTTP/1.1 povinná.
- **Cookie** hlavička slouží k předávání a uchovávání informací o probíhající konverzaci. Cookies implementuje stavový mechanismus v jinak standardně bezstavovém protokolu HTTP [2].
- **User-agent** obsahuje informace o aplikaci, která komunikuje s druhou stranou. Taková aplikace může být například běžný internetový prohlížeč nebo webový crawler[3]. Webové servery tuto hlavičku často vyžadují a v případě nepřítomnosti generují chybovou odpověď.

Na obrázku 2.2 je znázorněna struktura požadavku.



Obrázek 2.2: Formát HTTP požadavku

Formát odpovědi

Odpověď začíná tzv. stavovým řádkem (status-line), který obsahuje verzi protokolu a stavový kód s textovou formou kódu. Stavový kód, anglicky *status code*, je tříčíselný kód indikující výsledek zpracovaného požadavku. Rozděluje se do pěti kategorií určených podle první číslice:

- **1xx** - Tato třída obsahuje informační a obvykle provizorní odpovědi. V případě kódu *100 Continue* server očekává pokračování požadavku, aby mohl zaslat finální odpověď. Kódem 101 Switching Protocols server informuje o změně komunikačního protokolu.
- **2xx** - Žádost od klienta byla úspěšně přijatá a zpracovaná. Odpověď s kódem 200 OK se liší podle požadavku, například pro požadavek GET je v těle odpovědi obsah zdroje. Kód *202 Accepted* indikuje kladné zpracování požadavku s tím, že server stále požadavek zpracovává a klient může očekávat finalní odpověď.
- **3xx** - Pokud byl požadovaný zdroj dočasně nebo permanentně přemístěn, generují se stavové kódy 301 Moved Permanently a 302 Found. Klientovi poskytují více informací o přesměrování na daný zdroj. V případě podmíněného požadavku kód 304 Not Modified dává najevo, že se zdroj od posledního získání nezměnil.

[3]Web crawler je software pro automatizovaný sběr dat z webových serverů.

- **4xx** – Odpovědi této třídy se posílají na chybný nebo nezpracovatelný klientský požadavek. Na syntakticky chybný požadavek se odpovídá kódem *400 Bad Request*. Pokud klient není autentizován pro přístup ke zdroji, generuje se *401 Unauthorized*. Pokud server odmítá zpřístupnit zdroj třeba z jiných důvodů, používá se kód *403 Forbidden*. Pro neexistující zdroj nebo zdroj, který server nechce zviditelnit, se posílá známý kód *404 Not Found*.

- **5xx** – Tyto stavové kódy značí chybu na straně serveru, příkladem může být generický *500 Internal Server Error*. Kód *502 Bad Gateway* je obvykle spojován s proxy serverem, který nemohl vyhovět klientovi z důvodu selhání některého z nadřazených (upstream) serverů.

Po stavovém řádku následují hlavičky odpovědi. Poskytují další informace o úspěšném i neúspěšném vyřízení požadavku:

- **Server** hlavička poskytuje informace o softwaru, který odpověď generoval. Obvykle se zde uvádí název, například `Apache`.

- **Content-Length** informuje o velikosti těla odpovědi v bytech.

- **Content-Type** obsahuje informaci o formátu dat a použité znakové sadě. Příkladem hodnoty pro HTML dokument je `text/html; charset=utf-8`.

- **Content-Encoding** informuje o použitém kódování těla odpovědi.

- **ETag** obsahuje verzi zdroje a používá se u cachovacích mechanismů.

- **Last-Modified** obsahuje čas poslední modifikace zdroje.

2.3 Zabezpečení protokolu HTTP

HTTP poskytuje mechanismy pro autentizaci uživatelů při posílání požadavků. Základní schéma autentizace používá hlavičku požadavku *Authorization* pro zaslání uživatelského jména a hesla. Tyto mechanismy ale neřeší šifrování komunikace. Zachycením probíhajícího přenosu lze zrekonstruovat celé bloky dat, a to i neautorizovaným útočníkem. Tato vlastnost je pro některé aplikace nebezpečná.

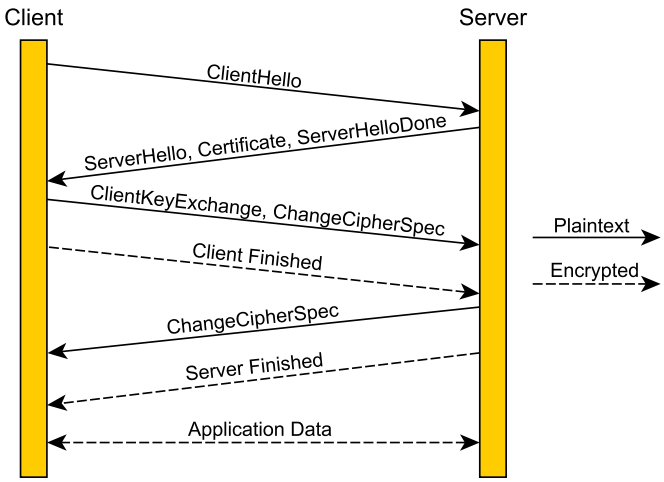
Hypertext Transfer Protokol over TLS, dále jen HTTPS [20], je protokol zajišťující šifrovanou zabezpečenou komunikaci. Před samotnou HTTP konverzací se vytvoří spojení s použitím technologie Transport Layer Security (TLS). V minulosti se využívala varianta HTTP over Secure Sockets Layer (SSL), která byla nahrazena TLS. Protokol HTTPS řeší zabezpečení autentizace a další komunikace ve většině moderních webových serverech.

Navázání TLS spojení

Před zahájením HTTP konverzace je potřeba ustavit TLS spojení pomocí *handshake* procedury. Předpokladem je podpora TLS na obou stranách. Stručně popsaný průběh je ukázán na obrázku 2.3.

- Klient pošle serveru zprávu 'Client hello' s kryptografickými informacemi,

- Server odpoví zprávou 'Server hello' a následuje zpráva obsahující certifikát serveru. Server také může požadovat certifikát po klientovi. Server dokončí inicializaci zprávou *Server hello done*
- Klient ověří pravost certifikátu přes certifikační autoritu. Tento krok zabrání falšování certifikátu při útoku *man-in-the-middle*[4].
- Klient vygeneruje a pošle Pre-Master Secret zašifrovaný veřejným klíčem, který získal z certifikátu serveru.
- Na základě Pre-Master Secret jsou obě strany schopné vygenerovat svoje symetrické klíče, kterými se bude obsah HTTP zpráv šifrovat a dešifrovat.
- Ukončení handshake proběhne výměnou nešifrovaných zpráv *Change cipher spec* a šifrovaných *Client finished* a *Server finished* pro ověření správného dokončení handshake. Další přenos aplikačních dat je šifrovaný.



Obrázek 2.3: TLS handshake

2.4 Internetová proxy

Při běžné komunikaci na protokolu HTTP se předpokládají dva účastníci, tedy klient přímo požaduje po serveru webový obsah a server přímo odpovídá klientovi. Procesu předávání HTTP zpráv se může zúčastnit i další server, který se nazývá *proxy server* [11]. Internetový proxy server zprostředkovává přeposílání zpráv mezi klientem a serverem. Stává se tak třetím síťovým prvkem, který zastupuje roli klienta nebo serveru v závislosti na směru zprávy. Mezi využití proxy serverů patří zapouzdření složitějšího systému distribuovaných serverů, kdy klient vnímá celý systém jako jeden server. Další využití je komunikace přes tunelovací proxy, který nemění obsah zpráv a pouze skrývá informace nižších protokolů (například IP adresu) původního klienta před serverem.

[4]Man-in-the-middle je typ síťového útoku, kdy útočník odposlouchává a zasahuje do komunikace mezi účastníky.

Pro využití klientských aplikací na Internetu se používá forward proxy. Spojení s tímto proxy serverem je obvykle v zájmu klienta. V případě veřejných open proxy je možné se setkat se službou anonymizace klienta před cílovým serverem. Požadavky jsou serveru přeposílány bez informací síťového protokolu, jako například IP adresa. V této oblasti je opakem transparentní proxy, která zahrnuje IP adresu původního klienta do zprávy cílovému serveru.

Ze strany distribuovaných serverů v privátní síti jsou zajímavé tzv. *reverzní proxy*. Klientovi se reverzní proxy jeví jako cílový server, který zajišťuje generování všech odpovědí. Ve skutečnosti tento server zastiňuje síť, která svoji odpověď na původní požadavek pošle klientovi přes proxy server. Reverzní proxy se stará o výběr cílového serveru a obvykle s nimi přímo spolupracuje na vyřízení složitějších požadavků, u kterých je nutné dešifrování, dekomprese a podobně. Při správném návrhu slouží reverzní proxy jako mechanismus vyrovnávání zátěže [6].

Cache proxy

Webový obsah je přenášen v HTTP odpovědích, které je možné uchovávat pro více klientů. Cache proxy servery uchovávají odpovědi vzdálených serverů po určitou dobu, čímž urychlují reakci na klientův dotaz. Zároveň se snižuje zatížení sítě na straně cílového serveru. V ideálním případě se cílové servery, které generují dynamický obsah, nemusí zabývat odpovídáním na stejné požadavky různých klientů na statické zdroje.

HTTP odpovědi jsou cachovatelné, pokud byly generované metodou GET nebo HEAD. Cachování odpovědí na POST požadavek je v některých případech možné, ale obvykle nebývá implementované. Odpovědi na další požadavky metod jako DELETE nebo PUT cachovatelné nejsou [7].

Kapitola 3

Webové archivační formáty

Archivační formát slouží pro uložení kolekce dat ve formě datových struktur nebo celých souborů a spolu s metadaty vytváří archivní soubor. Významnou vlastností webových archivačních formátů je kromě uložení textových dat také možnost znovu vizualizovat stránku v původní podobě. Výhodou archivních souborů je snadná přenositelnost – logický celek je sloučen do jednoho souboru. Archivy obsahují adresářovou strukturu včetně jednotlivých souborů a metadata pro detekci a opravu chyb. Archivační formáty obvykle využívají datové komprese pro zmenšení velikosti na disku. U některých typů archivačních formátu je možnost šifrování obsahu.

Archivace webu je proces sběru dat z webových stránek a ukládání do některého archivačního formátu, ať už za účelem uchovat stav webu pro pozdější analýzu, data mining nebo zveřejnění historického vzhledu webových stránek[1]. Problematiky archivace webu se dotýká pojem *web crawler*, což je software nasazován pro automatizované stahování a archivaci velkého množství dat z různých webových stránek nebo služeb.

Struktura webových dokumentů v archivu

Jádrem webových archivů je webová stránka, která je založena na stromové struktuře. Document Object Model, dále jen DOM, je objektově orientovaný model webových dokumentů standardizovaný organizací W3C. DOM tvoří rozhraní pro čtení a modifikaci obsahu a stylu částí dokumentu. Dnešní webové prohlížeče využívají DOM jako vnitřní reprezentaci HTML dokumentu, který je stěžejní pro vizualizaci webového obsahu.

3.1 Formát MHTML

MIME HTML, dále jen MTHML, je webový archivační formát pro uložení HTML dokumentů včetně odkazovaných zdrojů. Webová stránka je archivovaná v jediném souboru s příponou `.mht`. Obsah archívu je kódovaný pomocí rozšíření MIME. Formát MHTML je popsán jako internetový standard RFC 2557 [8]. Dnešní prohlížeče jsou schopné pracovat s formátem MHTML bez nutnosti instalování rozšíření.

Charakteristika rozšíření MIME

Multipurpose Internet Mail Extensions, dále jen MIME, je standard původně navržen pro posílání souborů přes email. V současnosti má uplatnění i obecně pro další hypermedi-

[1] Na stránkách https://archive.org/web/ můžete navštívit historii známých webů

ální obsah, proto bývá označován jako Media Extensions. Využívá se pro kódování souborů jako hypertextové dokumenty, netextové data, binární data a podobně. Obecně lze MIME uplatnit i mimo emailový protokol, například u přenosu netextových dat protokolu HTTP. Struktura MIME se dělí na hlavičku a tělo zprávy. Hlavička obsahuje položky, která pochází z původního návrhu pro email, MHTML tuto část využívá pro zaznamenání času archivace a další vhodné informace o souboru. Položka *Content-type* popisuje tělo zprávy. MHTML soubor, který se skládá z několika částí, je charakterizován MIME typem *multipart/related*. Tento typ umožňuje zřetězit části webových stránek. Mezi jednotlivé části se vkládá oddělovač – `multipart-boundary`.

Struktura MHTML

MHTML formát je strukturovaný jako MIME struktura typu `multipart/related`, jejíž části jsou potom popsány příslušným typem původního obsahu – *text/html* pro HTML dokument, *image/jpeg* pro obrázek a další. Celá struktura tvoří vázaný seznam, nebo-li sekvenci dokumentů ve formátu MIME.

Při vytváření MHTML souboru se prochází stromová struktura, přičemž navštívené uzly generují seznam MIME částí. Dalším průchodem seznamu se vytvoří příslušné vazby. Procházení stromu od kořene garantuje, že vazby na další části nikdy nebudou zpětné.

Protože je MHTML formát strukturovaný jako vázaný seznam MIME částí, je potřeba jej transformovat na stromovou strukturu webových stránek. Kořenový dokument je obvykle prvním prvkem seznamu, který odkazuje na další položky seznamu. Tyto položky jsou zpravidla uzly stromu o úroveň níže. Takto je možné projít lineární seznam a zpětně rekonstruovat stromovou strukturu webové stránky.

3.2 Mozilla Archive File Format

Mozilla Archive File Format [13], zkráceně MAFF, je formát pro archivaci webového obsahu. Umožňuje ukládání jedné nebo více webových stránek včetně multimediálních souborů do jednoho archivního souboru. MAFF je postaven na formátu ZIP souborů a využívá stejný princip komprese. MAFF není považován za internetový standard, formát vznikl jako rozšíření internetového prohlížeče Mozilla Firefox. Toto rozšíření již není v nových verzích podporované, přesto je specifikace MAFF udržovaná jako aktuální pro využití jinými aplikacemi [13].

Specifikace formátu

MAFF byl navržen pro archivaci výhradně webového obsahu – uložená metadata často určují URL webového zdroje, ze kterého byl archiv stránky vytvořen. MAFF není určen jako kompletní offline cache webového obsahu. Negarantuje, že archivované klientské skripty nebo reference na multimédia budou funkční, pokud využívají externí závislosti. Obsah webových stránek v MAFF souboru není vhodné částečně aktualizovat. Například pokud se obsah zdroje cílového serveru změní, je potřeba uložit celou stránku v aktuální podobě. Jednotlivé archivované stránky jsou proto chápány jako atomické prvky.

MAFF archiv používá stejnou metodu komprimace jako formát ZIP[2]. ZIP je běžně používaným formátem pro bezeztrátovou kompresi dat. Z tohoto pohledu se chová MAFF stejně a lze jej komprimovat a rozbalovat pomocí nástrojů pro ZIP formát.

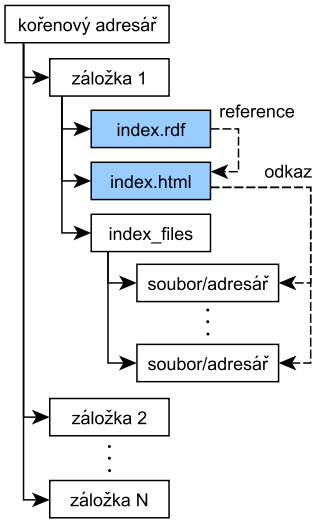
[2]ZIP je všeobecný souborový formát pro kompresi a archivaci dat

Adresářová struktura

MAFF je založen na adresářové struktuře a vychází ze struktury souborového systému, viz. obrázek 3.1. Kořenový adresář obsahuje pouze adrešáře pro uložené stránky a nesmí obsahovat žádné soubory. Jednotlivé adresáře potom obsahují metadata archivované stránky a stromovou strukturu založenou na uspořádání cílového webu.

Soubor pro uložení metadat jedné stránky se jmenuje `index.rdf`. Zde se ukládají informace o archívu jako čas uložení, znaková sada a podobně. Tento soubor může chybět v případě, že stránka obsahuje hlavní dokument `index.html` nebo index soubor jiného typu. Soubor RDF využívá značkovací jazyk XML[3]. Příklad RDF souboru je na obrázku 3.2.

Adresář pro webovou stránku dále obsahuje adresář `index_files`, ve kterém se vyskytují další dokumenty dostupné z indexového dokumentu. Při rozbalení se tento adresář zobrazí jako jedna záložka (bookmark) ve webovém prohlížeči.



Obrázek 3.1: Struktura MAF formátu

```
<RDF:RDF>
  <RDF:Description RDF:about="urn:root">
    <MAF:title RDF:resource="mozdev.org - maf: index"/>
    <MAF:originalurl RDF:resource="http://maf.mozdev.org/"/>
    <MAF:indexfilename RDF:resource="index.htm"/>
    <MAF:archivetime RDF:resource="Sat, 24 Nov 2018 16:11:28 +0100"/>
    <MAF:charset RDF:resource=""/>
  </RDF:Description>
</RDF:RDF>
```

Obrázek 3.2: Příklad souboru index.rdf

[3]XML – Extensible Markup Language

3.3 Frameworky pro archivaci webu

Archivace webu je proces sběru dat z WWW za účelem zachování obsahu webových stránek v archivu pro pozdější analýzu či zobrazení. Vzhledem k tomu, že Internet je velmi rozsáhlá síť obsahující přibližně 1.5 miliard webových stránek[4], je výhodné využít sofistikovaný software pro archivaci webu. Snahou je vytvořit takový software, který bude automatizovat činnost stahování, analyzování a archivování stažených dat.

Aplikace pro archivaci webu často musí řešit problémy stejného typu. Základem pro takové aplikace může být framework – programová struktura, která usnadní vývoj archivačního software pro specifický účel. Cílem frameworku pro archivaci je řešení typických problémů, například zpracování HTTP odpovědí, analýza stažených dokumentů a algoritmy pro vytváření archivů. Programátoři pracující s frameworkem se potom mohou soustředit na řešení problémů specifických pro své zadání.

3.4 Framework Lemmiwinks

Lemmiwinks je framework pro automatizované stahování a archivaci webového obsahu [17]. Je implementovaný v jazyce Python 3.6, který umožňuje využití asynchronních úloh pomocí standardní knihovny *asyncio*. Dále využívá několik externích knihoven, například *beautifulsoup4* pro analýzu HTML dokumentů nebo *tinycss* pro zpracování CSS.

Framework Lemmiwinks je rozdělen do čtyř samostatných modulů, které řeší specifické problémy samostatně

- *httplib* se stará o klientskou část aplikace. Modul obsahuje jednoduchého klienta a asynchroního klienta.
- *parslib* řeší analýzu stažených dokumentů, zejména HTML a CSS a zajišťuje stažení všech potřebných zdrojů, které jsou v dokumentech mnohdy odkazované rekurzivně.
- *archive* je modul pro archivaci staženého obsahu do formátu MAFF
- *extractor* specifikuje rozhraní pro extrahování dat pomocí regulárních výrazů.

V implementační části této práce se bude stavět na tomto frameworku. Pro získávání zdrojů k archivaci bude využit upravený modul *httplib*. Modul bude upraven tak, aby komunikoval se stahovací službou a získaval obsah skrze ni.

[4] http://www.internetlivestats.com/total-number-of-websites/

Kapitola 4

Návrh webových služeb

V softwarovém inženýrství je *servisně orientovaná architektura*, zkráceně SOA z anglického *Service-oriented architecture* [3], sada principů pro nasazení nezávislých komponent poskytující služby. Architektura je navržena pro fungování v počítačové síti. Jednotlivé komponenty implementují elementární činnosti a komunikují s okolím přes společný protokol. Tímto způsobem lze sestavit celý informační systém, jehož komponenty nejsou k tomuto systému pevně vázané.

4.1 Aplikační rozhraní

SOA je nejčastěji implementovaná pomocí webových služeb komunikujících pomocí protokolu HTTP. Jedním ze standardních přístupů k návrhu využívající HTTP je *Representational state transfer*, známé pod zkratkou REST [4]. REST je datově orientovaná technika pro distribuované hypermediální systémy a je použitelné pro jednotný přístup ke zdrojům. Zdrojem jsou data nebo stav aplikace odvoditelný z dat. Zdroj je klientovi předán jako jeho reprezentace, například ve formě JSON objektu s pevně specifikovaným schematem. Každý zdroj je identifikovány svým URL, který vychází ze specifikace protokolu HTTP.

Webové službě, která splňuje a správně implementuje architekturu REST, se říká RESTful [5]. Taková aplikace se řídí těmito omezeními:

- Oddělení zájmů klienta od zájmů serveru zajišťuje, že se komponenty systému mohou vyvíjet nezávisle.
- Server musí být bezstavový, což je přetrvávající princip z protokolu HTTP, kdy si server neudržuje kontext. Řídit stav komunikace je v možnostech klienta.
- Server je povinen klientovi sdělit, jak a která data odpovědí je možné uchovávat (vztahující se anglické pojmy *cache*, *cacheable requests*). Klient je oprávněn využít data odpovědi po určitou dobu, pokud by klient chtěl poslat ekvivalentní požadavek.
- Rozhraní mezi klienty a servery je uniformní. Všechny požadavky klienta jsou orientovány na zdroj, který odpoví reprezentací zdroje. Klient je potom schopný využít tuto reprezentaci k modifikaci zdroje na serveru. Každá zpráva musí obsahovat informaci, jak tuto zprávu zpracovat. Zprávy serveru mohou specifikovat možnosti kešování požadavků.

Službu lze chápat jako komponentu, která se chová jako server i klient zároveň. Pro splnění požadavku, což je serverová část služby, často aplikace komunikuje s jiným serverem či službou, čímž se stane klientem jiné služby.

4.2 Návrh služeb

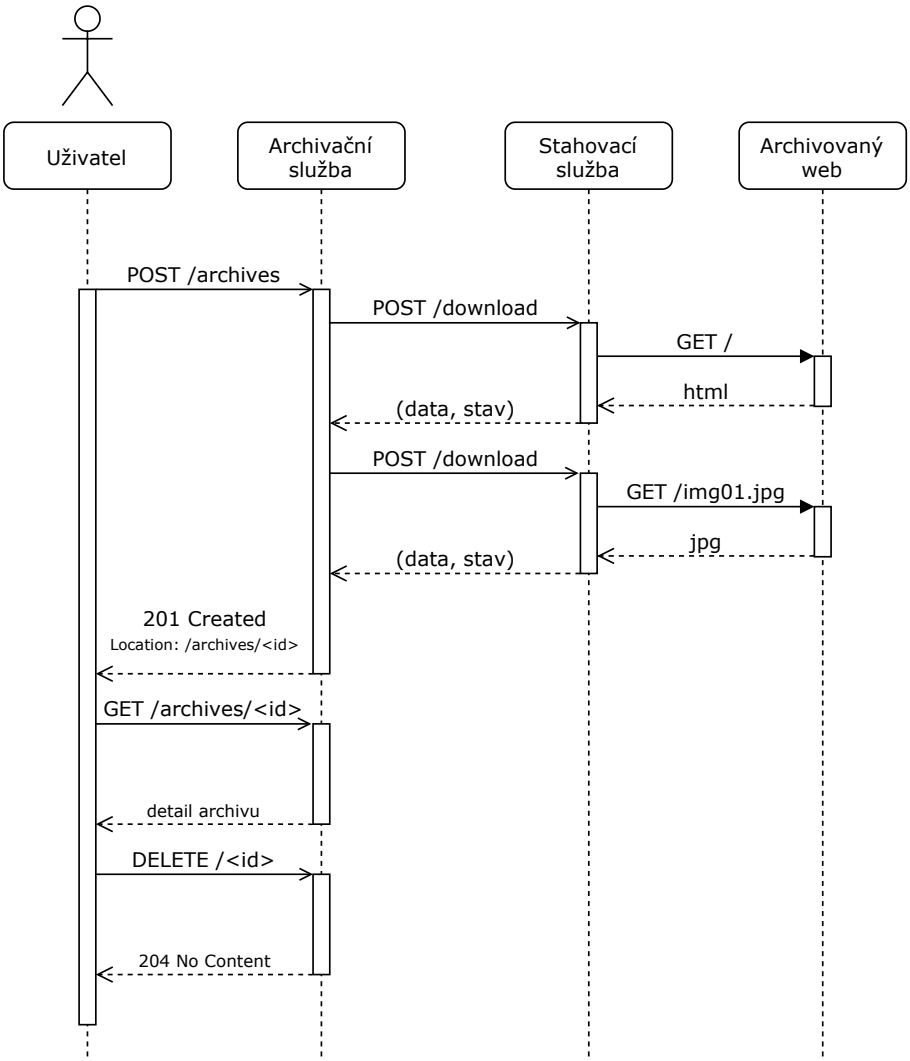
Archivace webu se dá rozdělit na 2 hlavní podúkoly, které lze implementovat jako samostatné služby:

- Stahovací služba – typickou úlohou je stažení zdroje na požadovaném URL, vyhodnotit úspěšnost stažení, poskytnout informace o přesměrování a podobně. Služba umožňuje stahovat obsah přes síť Tor.
- Archivační služba – účelem služby je uchovávat existující archivy, poskytnout o nich informace a umožnit jejich stažení. Služba především umožňuje vytvořit nový archiv z poskytnutých URL adres a nastavení. Proces archivace řídí framework Lemmiwinks – pro získávání dat k archivaci využívá výše zmíněnou stahovací službu. V případě úspěchu stahování ze zdrojů se ze získaných dat vytvoří archiv ve formátu MAFF.

Pro přístup k systému z vnějšku se často vytváří ještě služba typu *API gateway* (brána), která funguje jako řídící jednotka při vyhodnocování požadavků u složitějších systémů. V případě této architektury bude bránou archivační služba – uživatelům bude poskytovat webové rozhraní pro prohlížeč a API založené na reprezentaci v JSON formátu.

K návrhu obou služeb a jejich API byl použit nástroj Swagger [19]. Nástroj využívá OpenAPI specifikaci pro návrh a dokumentaci rozhraní a validační schemata.

Na obrázku 4.1 je zobrazen sekvenční diagram příkladu komunikace navržených služeb. Uživatel zašle archivační službě požadavek obsahující seznam URL adres. Archivační služba zprávu zpracuje a zahájí komunikaci se stahovací službou. Zdroj `/` může představovat např. soubor `index.html`, který obsahuje element `img`, jehož zdrojem je soubor `img01.jpg`, což bude například obrázek v HTML dokumentu. Po získání všech potřebných souborů je vytvořen MAFF archiv na disku archivační služby. Při úspěšné archivaci uživatel dostane odpověď s hlavičkou `Location`, ve které je reference na nově vytvořený archiv. Archiv lze později stáhnout nebo smazat.



Obrázek 4.1: Sekvenční diagram komunikace služeb

4.3 Rozhraní služby pro archivaci

Tato služba bude jádrem systému a bude sloužit jako výchozí bod pro využití aplikace. Součástí je rozhraní pro webový prohlížeč a JSON API. Toto rozhraní je také popsané dokumentem OpenAPI, který lze nalézt v příloze A.

Rozhraní pro webový prohlížeč reprezentuje zdroje formou HTML dokumentů. Toto rozhraní je následující:

- **GET /** – Hlavní stránka ve formě HTML dokumentu. Obsahuje jednoduchou navigaci webu, přihlašovací formulář a základní informace o službě.
- **GET** `/login` – HTML stránka s přihlašovacím formulářem. Uživatel zadá jméno a heslo a následně odešle metodou POST.

- **POST** `/login` – Zašle přihlašovací údaje k ověření v těle požadavku. Tělo obsahuje data typu *application/x-www-form-urlencoded*:
  - `username`
  - `password`

  Úspěšné přihlášení generuje odpověď s přesměrováním na hlavní stránku.

Další zdroje jsou dostupné pouze autorizovaným uživatelům, kteří jsou přihlášení:

- **GET** `/logout` – Odhlásí uživatele a přesměruje jej na hlavní stránku.
- **GET** `/archives` – Stránka s archivy. V HTML tabulce je seznam všech existujících souborů MAFF uložených na serveru s odkazy na jejich detail. Dále je zde formulář pro zaslání POST požadavku, kterým se vytváří nový archiv.
- **POST** `/archives` – Požadavek na vytvoření archivu. Tělo požadavku jsou data typu *application/x-www-form-urlencoded* získané z HTML formuláře:
  - `url` – URL adresa archivovaného obsahu
  - `name` – název vytvořeného archivu. Název může obsahovat pouze velká a malá písmena, číslice a pomlčku
  - `forceTor` – pokud je nastaven na `true`, potom pro všechny požadavky na stahování bude vyžádané použití Toru bez ohledu na to, jestli je URL dostupná obyčejným klientem. V opačném případě služba požádá o stažení přes Tor u rozpoznaných `.onion` adres.

  Odpověď s kódem `201 Created` obsahuje hlavičku `Location` s referencí na nově vytvořený archiv.
- **GET** `/archives/{id}` – Stránka s detaily konkretního archivu. Obsahuje obecné informací o archivu jako čas vytvoření a velikost. Je zde odkaz na stažení archivu.
- **GET** `/archives/{id}/{filename}` – stažení archivu `.maff`. Data odpovědi jsou typu *application/x-maff*.

Službu může využívat i klientská aplikace, která komunikuje s API ve formě JSON objektů v těle zpráv. Přístup k API vyžaduje zaslání hlavičky `Authorization` se správnými přihlašovacími údaji. Rozhraní je následující:

- **GET** `/api/archives` – JSON kolekce všech existujících archivů MAFF uložených na serveru. Pro vyhledávání a omezení rozsahu kolekce lze použít `query string`, jehož argumenty jsou:
  - `name` – vyhledání podle názvu archivu nebo podřetězec obsažený v názvu.
  - `skip` – počet přeskočených položek od začátku kolekce.
  - `limit` – počet požadovaných položek kolekce.

- **POST** `/api/archives` – Požadavek na vytvoření nového archivu. Tělo zprávy obsahuje data typu *application/json*:
  - `urls` – seznam řetězců URL pro archivaci.
  - `name` – název vytvořeného archivu.
  - `forceTor` – Pro všechny požadavky na stahování bude vyžádané použití Toru. Parametr má stejnou sémantiku jako u **POST** `/archives` popsaného výše
  - `headers` – obsahuje dvojce název HTTP hlavičky a hodnotu. Tyto hlavičky budou předané stahovací službě.
- **GET** `/api/archives/{id}` – Reprezentace detailu jednoho archivu daného parametrem `id`.
- **DELETE** `/api/archives/{id}` – Smazání jednoho archivu daného parametrem `id`.
- **GET** `/api/archives/{id}/{filename}` – Stažení souboru .maff archivu s příslušným `id`.

Požadavky na chráněné zdroje od neautorizovaného uživatele vedou k odpovědi s kódem `401 Unauthorized`. Neexistující zdroj, kolekce, identifikátor a podobně vedou k odpovědi `404 Not Found`. Použití nepovolené metody na existující zdroj směřuje ke odpovědi `405 Method Not Allowed`.

4.4 Rozhraní služby pro stahování

Základem této služby je zajistit stažení požadovaného zdroje a informovat o případném neúspěchu stahování. Rozhraní je taktéž popsané OpenAPI dokumentem v příloze A. Stahovací služba má následující rozhraní:

- **GET** */* – Hlavní HTML stránka, která obsahuje základní informace o využití API služby.
- **POST** */api/download* – Požadavek na stažení zdroje. Tělo zprávy obsahuje *application/json* data:
  - `resourceURL` – řetězec URL, který identifikuje zdroj ke stažení.
  - `headers` – obsahuje HTTP hlavičky, které budou vloženy do GET požadavku. Lze nastavit například `User-Agent` string nebo `Accept-Language` pro vyžádání specifické jazykové mutace obsahu.
  - `useTor` – v případě nastavení na `true` služba použije Tor pro stahování, jinak použije obyčejného klienta.

  Při úspěšném stažení nebo alespoň nějaké odpovědi od vzdáleného webu bude odpověď ve formátu *application/json* následující:
  - `content-type` – MIME typ staženého obsahu, například *application/png*.
  - `url_and_status` – pole párů url a status odpovědi. Pokud nedojde k přesměrování, pole bude obsahovat 1 pár.
  - `data` – binární nebo textový stažený obsah v kódování *base64*.

  Pokud se nestáhnou žádné data, odpověď je typu `204 No Content`.

Kapitola 5

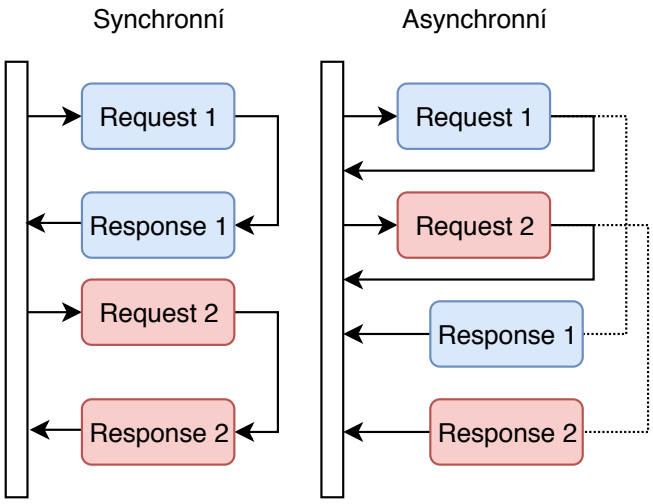
Implementace

Tato kapitola popisuje techniky asynchronního programování a implementační detaily obou služeb v jazyku Python 3.6. Bude přiblížen způsob využití frameworku Sanic v kombinaci s asynchronním klientem. V části týkající se archivačního frameworku budou zdokumentovány modifikace frameworku Lemmiwinks a motivace těchto modifikací.

5.1 Asynchronní programování

Asynchronní programování je druh programování obecně jakékoliv aplikace, kde je kladen důraz na souběžný běh jednotek. Stále je zde k nalezení primární vlákno, kterým obvykle aplikace začíná a stará se o asynchronně běžící úlohy. Po dokončení úlohy je hlavní vlákno informované o stavu proběhlé operace.

Na obrázku 5.1 je zobrazen rozdíl mezi synchronním a asynchronním vyřízením 2 požadavků. Tečkovaná čára značí vytvoření korutiny pro zpracování požadavku, která se váže ke konkretní později generované odpovědi. Časový interval mezi odpověďmi může být daleko větší, protože vyřízení různých požadavků může trvat značně delší dobu.



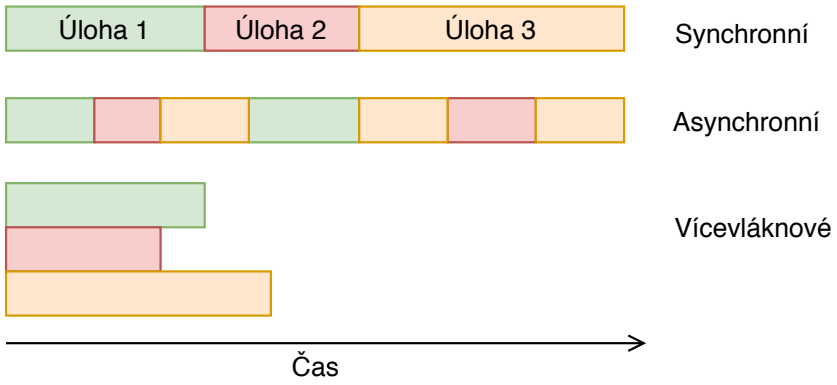
Obrázek 5.1: Rozdíl mezi synchronním a asynchronním modelem požadavek-odpověď

Paralelní běh úloh je výhodně využitelný při implementaci serverové logiky, například v rámci webových služeb, kde je důležitá rychlá reakce na požadavky, a to zejména na požadavky, které jsou snadno zpracovatelné a není důvod generovat a posílat odpověď se zpožděním. Vyřizování požadavků často obsahuje operace, kdy se čeká na stažení dat po síti a podobně. Asynchronně napsaná aplikace potom nebude v těchto případech blokující.

Podobně lze asynchronně naprogramovat klientskou logiku. Například pokud je potřeba získat N zdrojů nezávisle na sobě, je možné paralelně poslat všechny požadavky a postupně přijímat odpovědi od serveru.

Důležitým rozdílem oproti vícevláknové aplikaci je pouze 1 běžící vlákno. V asynchronním kódu Pythonu je paralelismus v režii interpretu a modulů pro řešení asynchronních operací. Využití vícevláknových aplikací v Pythonu je problematické z důvodu samotné implementace interpretu Pythonu, jenž využívá mutex *Global Interpreter Lock* pro řízení přístupu k objektům a bytekódu Pythonu [22], asynchronní kód ale tento problém nemá.

Na obrázku 5.2 je ilustrace výkonávání úloh v čase pro různé varianty plánování.



Obrázek 5.2: Příklad zpracování 3 úloh synchronně, asynchronně a více vlákny

Asyncio

Od verze Python 3.4 je k dispozici modul *asyncio* a *async/await* syntaxe [18]. Modul asyncio implementuje tyto speciální konstrukce:

- *smyčka událostí* – anglicky také *event loop* – je smyčka která řídí a distribuuje vykonávání úloh. Řeší tok řízení mezi novými a existujícími úlohami. Kromě základní implementace smyčky také existuje alternativní implementace *uvloop*.
- *korutiny* jsou speciální funkce deklarované syntaxí *async*. Korutina před spuštěním musí být naplánovaná ve smyčce událostí. Korutina může předat řízení zpět pomocí *await* syntaxe.
- *futures* jsou objekty, které reprezentují výsledek asynchronní operace.

5.2 Struktura projektu

Kořenový adresář projektu se jmenuje `Lemmiwinks-services`. Tento adresář dále obsahuje podadresáře jednotlivých služeb a testovacího klienta:

- `archive_service`

- `download_service`
- `test_client`

Adresáře obsahují vlastní zdrojové kódy a závislosti. Tato struktura dovoluje systém mikroslužeb dále dělit a rozšiřovat o nezávisle vyvíjitelné služby, například službu pro uživatelské webové rozhraní či jiné implementace služeb v odlišných technologiích.

Kořenový adresář dále obsahuje soubor `README.md` se základními informacemi o projektu a návod k instalaci. Soubor `requirements.txt` obsahuje všechny externí balíčky a moduly pro Python 3.6 použité v projektu. Pro jejich nainstalování je potřeba vytvořit virtuální prostředí v Pythonu a následně příkazem `pip install -r requirements.txt` spustit instalaci.

5.3 Archivační služba

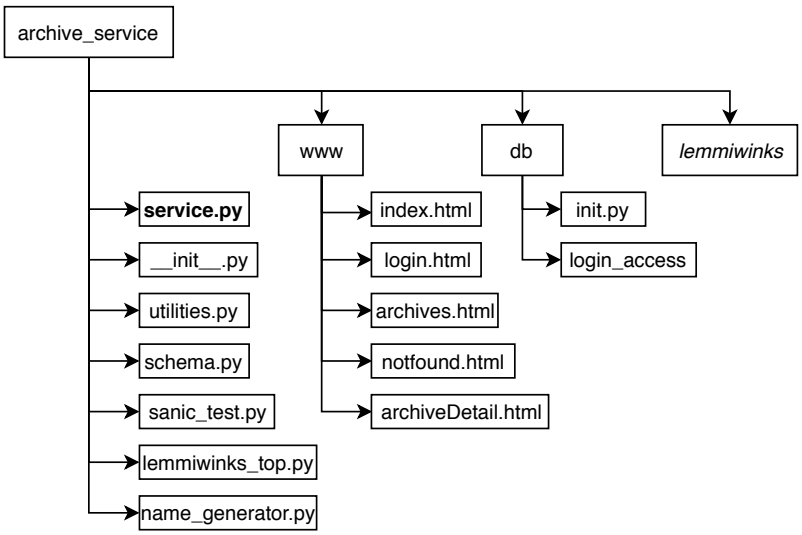
Hlavní úlohou této služby je poskytovat uživateli reprezentaci archivů, umožnit jejich stažení a řídit proces vytváření archivu. Implementace se vyskytuje v adresáři `archive_service` s top-level souborem `service.py`. Archivační služba využívá modifikovanou verzi frameworku Lemmiwinks. Úpravy se týkají především HTTP klienta.

Služba běží v rámci jediného procesu a lze ji lokálně spustit příkazem:

```
python3.6 service.py --download_service_url <URL>
```

Argument `--download_service_url` specifikuje URL adresu stahovací služby, kterou tato služba využívá. Při vynechání argumentu se použije výchozí URL `http://0.0.0.0:8081`

Stromová struktura zdrojových souborů je zobrazena na obrázku 5.3. Struktura modulu Lemmiwinks je popsaná v kapitole 3.4.



Obrázek 5.3: Soubory archivační služby

Implementace API

Pro implementaci serverové funkcionality je nasazen web server framework Sanic verze 19.03 [16]. Při spuštění se automaticky inicializuje objekt Sanic, který reprezentuje instanci

serveru a běží navždy do doby, než je proces interpretu ukončen. Pro testovací účely je server spouštěn na adrese `0.0.0.0` na portu `8080`, kde očekává příchozí HTTP požadavky.

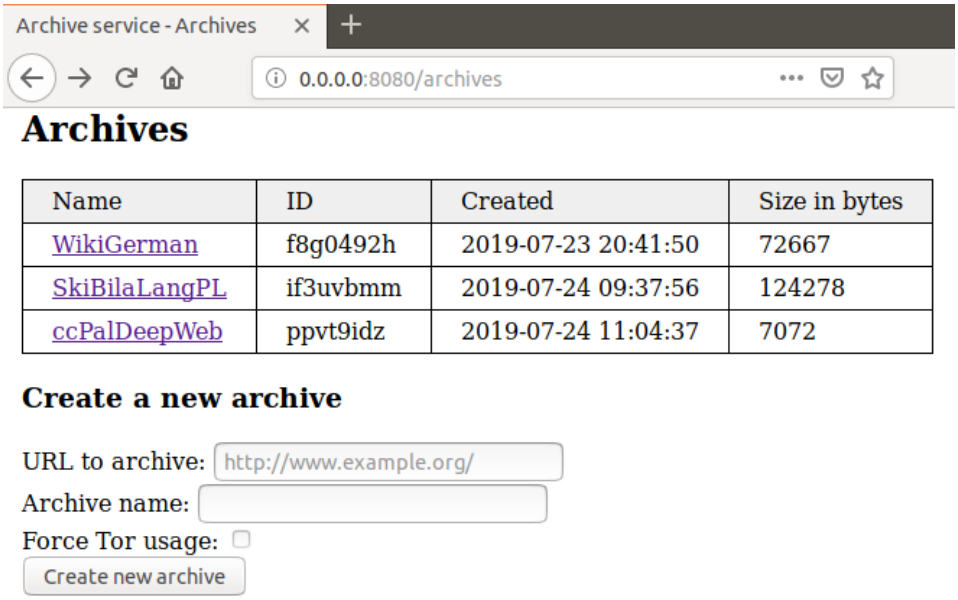
Služba implementuje REST API popsané v kapitole 4.3. Pro přístup ke zdrojům služby jsou implementované *handler* funkce.

Rozhraní pro webový prohlížeč využívá HTML dokumenty jako šablonu pro generování odpovědí. HTML a CSS soubory jsou uložené v adresáři *www*. Rozhraní pro webový prohlížeč je implementované těmito obslužnými funkcemi:

- `get_index` – `GET /` – vygeneruje HTML dokument ze souboru index.html, který obsahuje úvodní informace. Přihlášenému uživateli zobrazí tlačítko k pokračování k archivům, nepřihlášenému uživatel bude vyzván k přihlášení odkazem na `login`.
- `login` – `GET, POST /login` – zašle přihlašovací formulář ze souboru login.html. Potvrzovací tlačítko formuláře vytvoří POST požadavek. Zadané přihlašovací údaje jsou ověřované v souboru login_access. Po úspěšném přihlášení je uživatel přesměrován na hlavní stránku, jinak dojde k zobrazení upozornění pod formulářem.
- `logout` – `GET /logout` – odhlásí uživatele, pokud byl přihlášen. Jako odpověď odešle přesměrování na hlavní stránku.
- `get_archives` – `GET /archives` – vygeneruje odpověď ze souboru archives.html. Dokument obsahuje tabulku, která obsahuje informace a odkazy na uložené archivy získané z třídy Archives. Příklad stránky je na snímku obrazovky 5.4.
- `post_archives` – `POST /archives` – získané data z formuláře ověří pomocí schema validace. Následně inicializuje framework Lemmiwinks s požadovanými URL a spustí archivaci. Po vytvoření archivu je uživatel upozorněn zprávou pod formulářem.
- `get_archive_item(id)` – `GET /archives/{id}` – podle parametru *id* se vyhledá archiv a vygeneruje odpověď ze souboru archiveDetail.html. Stránka obsahuje informace o konkrétním archivu s odkazem na stažení. Pro neexistující archiv je generována stránka notfound.html.
- `get_archive_file(id, filename)` – `GET /archives/{id}/{filename}` – podle parametru id se vyhledá archiv a jeho soubor *.maff* je zaslán jako file stream typu *application/x-maff*.

Rozhraní API poskytuje reprezentace ve formě JSON objektů. Významově jsou zdroje podobné. Rozhraní je implementované těmito funkcemi:

- `api_get_archives` – `GET /api/archives` – vrátí seznam detailů archivů. Navíc oproti webovému rozhraní je možné omezit množinu výsledků podle zadaných parametrů *name*, *limit* a *skip* v řetězci dotazu (*query string*).
- `api_post_archives` – `POST /api/archives` – data z požadavku ověří pomocí validačního schema. Inicializuje parametry archivace a spustí archivaci.
- `api_get_archive_item(id)` – `GET /api/archives/{id}` – vyhledá archiv a vytvoří JSON reprezentaci jeho detailu.
- `api_delete_archive_item(id)` – `DELETE /api/archives/{id}` – vyhledá a smaže archiv. Odpověď je vždy s kódem 204, i když archiv neexistuje.



Obrázek 5.4: Stránka s archivy

- `api_get_archive_file(id, filename)` – `GET /api/archives/{id}/{filename}` – stáhne soubor *filename* MAFF archivu *id*.

Pro každý požadavek se aktivuje právě 1 obslužná funkce, která vždy vrací `response` objekt definovaný v Sanicu. Provázání obslužné funkce a zdroje se provádí dekorátorem `@app.route`, kde `app` je instance třídy Sanic. Cesta ke zdroji se specifikuje v prvním parametru a seznam povolených metod pro přístup k tomuto zdroji. Handler musí být vždy deklarován jako asynchronní pomocí syntaxe `async`.

Validace požadavků

Příchozí POST požadavky obsahují tělo, které obsahuje uživatelem zadané data. Tyto data je nutné validovat a na nevalidní data generovat chybovou odpověď. Základní kontroly syntaxe HTTP požadavků provádí již Sanic.

Tělo požadavku je ověřeno pomocí schema. Pro validaci požadavků na API je použitý externí modul *jsonschema*. Tento modul implementuje specifikaci schemat OpenAPI pro Python. V souboru schema.py je definovaná třída Schema s položkami:

- `instance` – příchozí data v těle požadavku
- `schema` – předpis pro validaci dat.
- `is_valid` – ověří, že data v `instance` odpovídají schematu v `schema`. Vrací boolean.

Třída ArchiveDetailSchema dědí od třídy Schema a implementuje předpis pro data, která modelují detaily archivu. Dále třída ArchivePostSchema je předpis pro příchozí POST požadavky na zdroj `/api/archives`.

V případě nevalidních dat je vyvolána výjimka ValidationError. Tato výjimka je zachycena v metodě `is_valid`.

Třída Archives

Tato třída modeluje existující archivy uložené na serveru. Třída je definovaná v souboru utilities.py. Při inicializaci se vytvoří seznam všech souborů .maff. Třída má 4 metody:

- `details` – vrátí seznam s informacemi o všech archivech.
- `detail` – parametr file je název souboru. Vrátí informace o konkretním archivu:
  - *file* – název souboru s příponou .maff
  - *name* – název archivu
  - *aid* – identifikátor archivu
  - *ctime* – čas vytvoření archivu
  - *size* – velikost v bytech
  - *href_download* – hypertextový odkaz na stažení
  - *href_download_api* – reference na stažení pro API
  - *href_detail* – hypertextový odkaz na stránku s detailem archivu
  - *href_detail_api* – reference na detail archivu
- `searchById` – vrátí detail archivu s odpovídajícím ID. Vrací None, pokud takové ID neexistuje.
- `searchByName` – vrátí detaily archivů, kde se shoduje část názvu. Vrací prázdný list, pokud nedojde k žádné shodě.

Název archivu

ArchiveName je třída, která implementuje skládání názvů archivů. Před uložením nového archivu se vygeneruje název bez přípony .maff, který se skládá ze 2 řetězců spojených podtržítkem:

- Název archivu, který uživatel zaslal v požadavku na archivaci. Pokud uživatel žádný název nezadal, generuje se jako výchozí název `unnamed`.
- Generovaný náhodný identifikátor složený z 8 malých písmen a-z nebo číslic 0-9, například `tz54aj0o`.

Generování náhodného identifikátoru zabrání kolizi, která by mohla nastat v případě vytvoření archivů se stejným jménem. Při použití 26 písmen anglické abecedy a 10 číslic je pravděpodobnost kolize identifikátorů prakticky nulová, a to i při existenci tisíců různých archivů. Zároveň z identifikátoru nejde odvodit posloupnost vytvoření a identifikátory mají vždy stejnou délku.

Autentizace

Uživatel, který chce použít všechny možnosti služby, se musí autentizovat pomocí uživatelského jména a hesla. API je dále dostupné při použití HTTP hlavičky Authorization se správnými přihlašovacími údaji ve formě Basic Authentication.

V rozhraní pro prohlížeč je autentizace implementovaná jako HTML login formulář, jehož obsah se posílá metodou POST na `/login`. Správnosti zadaných údajů se ověřuje

přes souborovou databázi, kterou implementuje Python modul TinyDB. V adresáři db se nachází skript init.py, který generuje soubor `login_access` s uživateli a jejich hesly. Pro demonstrační účely existuje uživatel `admin` s heslem `admin`. Přístup k chráněným endpointům služby řeší modul `Sanic-auth`. Přihlášený uživatel, resp. jeho session je uložena v objektu `auth`. Handlery jsou potom dekorované pomocí `@auth.login_required`.

Přístup k API funguje na principu Basic authentication. Požadavky na API musí obsahovat hlavičku Authorization. Obsah hlavička se skládá z klíčového slova `Basic` a kódovaného uživatelského jména a hesla spojené znakem dvojtečky (`:`). Celá hlavička pro uživatele *admin:admin* bude vypadat takto:

```
Authorization: Basic YWRtaW46YWRtaW4=
```

Testy

Framework Sanic poskytuje testovací metodu `test_client` pro instanci `app` třídy Sanic. Pro testování archivační služby musí běžet stahovací služba na `0.0.0.0:8081`. V souboru `sanic_test.py` jsou implementované funkce na otestování základních vstupů a výstupů API:

- `test_index` – základní test, ověří přístup k index.html dokumentu a jeho získání s typem *text/html*
- `test_api_unauth` – testuje přístup bez autorizace k chráněným zdrojům. Očekávané kódy jsou 401 Unauthorized.
- `test_api_auth` – správné výstupy pro autorizovaného uživatele. Očekávané kódy jsou z kategorie 2xx nebo 3xx.
- `test_api_bad_requests` – testuje špatné vstupní data. Očekávaný kód je 400 Bad Request.

Klient

Abstraktní třída asynchronního klienta je součástí frameworku Lemmiwinks v modulu `httplib`.

Asynchronní klient je implementovaný třídou ServiceClient abstraktní třídy AsyncClient. Konkretní implementace ServiceClient je podobná již existující třídě AIOClient s těmito hlavními rozdíly:

- Třída implementuje metodu `post_request`. Pro stahování zdrojů je využíváno aplikační rozhraní stahovací služby, které zasílá požadavky metodou POST.
- Přijatá data jsou formátu JSON, který nese metadata obsahu a samotný obsah v kódování Base64. Tento obsah je potřeba dekódovat a uložit jako dočasný soubor pro potřeby frameworku.

Pro klienta ServiceClient je přidána položka *service_factory* ve třídě frameworku ClientFactoryProvider pro zaregistrování nové implementace asynchronního klienta. Třída *HTTPClientDownloader* frameworku byla přepsána tak, aby komunikovala se stahovací službou přímo.

Vstupním bodem frameworku je vytvořená třída ArchiveServiceLemmiwinks, která dědí od kostry třídy Lemmiwinks. Je zde deklarace veřejné asynchronní metody `task_executor`. Účelem je inicializovat seznam úloh, které budou dále naplánovány pro asynchronní vykonávání.

Archivace je implementovaná v privátní metodě `__archive_task`. Metoda má 2 povinné argumenty:

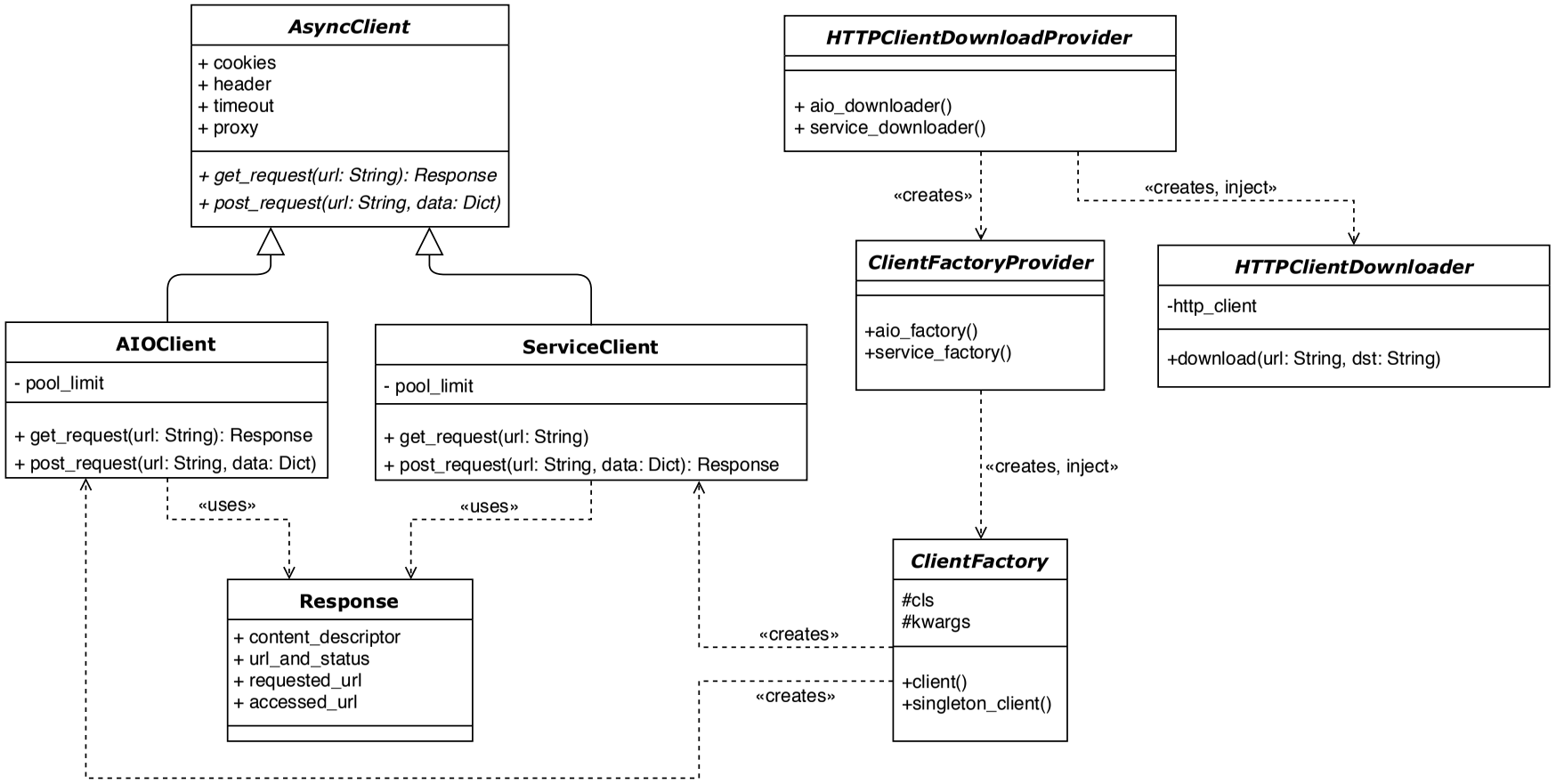
- `urls` – seznam URL řetězců, jejichž obsah se bude archivovat
- `archive_name` – řetězec reprezentující název archivu bez přípony

Pro každou URL se následně vytvoří úloha s klientem pro zaslání požadavků na stažení. Tyto úlohy jsou spuštěny paralelně. Naplánování a synchronizaci těchto úloh řeší metoda asyncio.gather, která vyčká na dokončení všech úloh stahování.

Odpověď od stahovací služby s sebou nese data v kódování *Base64* [9]. Používá se pro přenos binárního obsahu v textovém protokolu HTTP nebo JSON řetězců, které mají reprezentovat netextovou hodnotu. Pro dekódování dat je použitý standardní modul Pythonu *base64*.

Stažená data a její deskriptory na soubory jsou dále předány úloze *___archive_responses*. Framework Lemmiwinks definuje seznam tabů v budoucím MAFF archívu třídou Envelop a jednotlivé taby jako objekt *letter*. V úloze se iterativně naplní objekt envelop a vytvoří se odpovídající MAFF archív.

Na obrázku 5.5 je znázorněn diagram tříd upraveného modulu *httplib*. Abstraktní třída *AsyncJSClient* a třída *SeleniumClient* z původního Lemmiwinks nejsou využity a proto nejsou součástí diagramu.



Obrázek 5.5: Diagram tříd klienta archivační služby

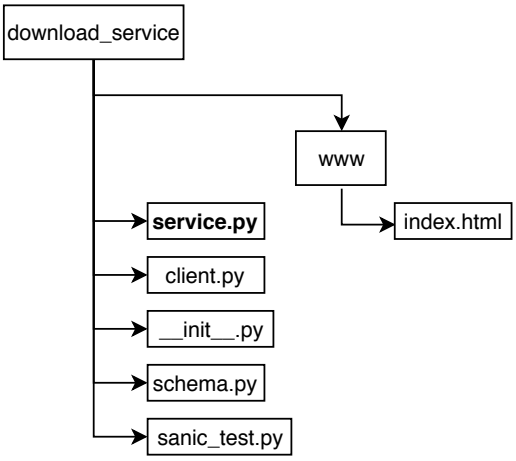
5.4 Stahovací služba

Účelem této služby je řešit komunikaci se vzdálenými servery, na kterých se nalézají požadované zdroje k archivaci. Podobně jako u archivační služby je jádrem aplikace server a příslušná implementace HTTP klienta. Implementace se nalézá v adresáři `download_service` s top-level skriptem `service.py`. Tato služba očekává požadavky primárně od archivační služby.

Stahovací služba obsahuje tyto závislosti:

- Sanic,
- Tor client for Ubuntu [21],
- pysocks,
- aiohttp_socks,
- jsonschema.

Struktura zdrojových kódů je zobrazena na obrázku 5.6.



Obrázek 5.6: Struktura zdrojových souborů stahovací služby

Implementace API

Serverová část využívá modul Sanic verze 19.03, implementace a využití je podobné archivační službě. Server je spouštěn lokálně na adrese `0.0.0.0` a poslouchá na portu `8081`.

Požadavky jsou obsluhovány těmito funkcemi:

- `get_index` – generuje stránku ze souboru index.html s informacemi o službě.
- `api_post_download` – validuje JSON data v požadavku. Následně inicializuje normálního klienta nebo klienta pro Tor v závislosti na položce *useTor* v požadavku. Klient stáhne zdroj z *resourceURL* a vygeneruje odpověď.

Validace požadavků

Třída `DownloaderJsonSchema` v souboru `schema.py` implementuje validaci JSON struktury v požadavku POST na `/api/download`. Validace probíhá inicializací objektu DownloaderJsonSchema s parametrem instance, kterým je JSON objekt k validaci. V případě nevalidní JSON instance je vyvolána výjimka ValidationError. Schéma JSON zprávy je znázorněno v tabulce 5.1.

Vlastnost	**Typ**	**Povinný**
resourceURL	string	ano
useTor	boolean	ano
headers	object	ano

Tabulka 5.1: JSON schema pro POST požadavky na stahovací službu

Testy

V souboru `sanic_test.py` jsou implementované funkce na testování základních vstupů a výstupů služby. Vyhodnocují se správné HTTP stavové kódy a typ odpovědi.

- `test_index` – základní test, ověří přístup k index.html dokumentu a jeho získání s typem *text/html*
- `test_api` – testuje správné chování API. Očekává stavové kódy z kategorie 2xx.
- `test_api_bad_requests` – testuje kombinace špatných vstupních dat, očekává odpovědi s kódem 400 Bad Request.

Klient

Klient stahovací služby využívá knihovnu aiohttp. Komunikuje se vzdáleným serverem, jehož zdroj je požadován k archivaci. Klient je implementovaný třídou DownloaderClient, v konstruktoru se nastavuje:

- **headers** – HTTP hlavičky, které budou vloženy do GET požadavku.
- **useTor** – pokud je nastaven na *true*, interně využije SOCKS5 [10] a běžící Tor-socks proxy na portu 9050. Umožňuje přistupovat na adresy .onion, které jsou využívané na deep webu.
- **timeout** – doba čekání na odpověď

Požadavek se posílá metodou `get_request(url)`.

Kódování staženého obsahu

Stažený netextový obsah, jako jsou například obrázky, je převeden do textového kódování Base64. Tento formát lze reprezentovat řetězcem a je tedy možné jej přenést v JSON formátu. Base64 implementace v Pythonu je v modulu `base64`. Výsledný řetězec bajtů je potom převeden do UTF-8 textového kódování:

```
base64.b64encode(bytes(content)).decode('utf-8')
```

Kapitola 6

Testování

Tato kapitola se zaměřuje na testování implementovaných služeb. V prvním sekci proběhnou testy se zaměřením na měření doby zpracování při zatížení větším počtem požadavků na archivační službu. K zasílání požadavků bude využitý jednoduchý asynchronní klient.

Další část testování se zaměřuje na testování archivace stránek z *deep webu*. Účelem těchto testů bude porovnání kvality a vizuální přesnosti vytvořených archivů s realným obsahem.

6.1 Rychlost zpracování požadavků

K účelům testování služeb byl implementován asynchronní klient ve třídě `AsyncTestClient` v souboru `test_client.py`. Testovací klient využívá knihovny *aiohttp* a *asyncio* pro zasílání HTTP požadavků na archivační službu. Třída obsahuje metodu `post_requests` pro vytváření POST požadavků s argumenty:

- `data` – JSON data v požadavku na API archivační služby
- `count` – počet asynchronních požadavků poslaných najednou.

Aby rychlost testů archivace nebyla zkreslená z důvodů čekání na síťové operace, je využit testovací server v adresáři `test_server`. Server obsahuje ve svém kořenovém adresáři jednoduchý HTML dokument s CSS a obrázkem.

Server lze spustit příkazem `python3.6 service.py` a poběží na `http://0.0.0.0:8000`.

Pro vytváření asynchronních GET požadavků slouží metoda `get_requests` s argumenty:

- `params` – parametry v query string v URL požadavku. Testované parametry jsou `name`, `limit` a `skip`.
- `count` – počet asynchronních požadavků poslaných najednou.

Pro časování byl použit Python modul `time`. Realný čas byl zachycen před a po vykonání asynchronní smyčky. Rozdílem těchto časových hodnot je uplynulá doba.

Vyhodnocení

Testování služeb proběhlo na virtuálním stroji s operačním systémem Ubuntu 18.04.1 LTS s těmito prostředky:

- CPU: Intel Core i7-4700HQ CPU @ 2.40GHz
- RAM: 4 GB
- Síťová karta: Realtek PCIe GBE (1000 Mbit/s)
- Rychlost připojení: 20 Mbit/s download, 3 Mbit/s upload

Při testování běžel testovací klient, obě služby a archivovaný web na lokálním počítači. Rychlost archivace byla vyzkoušena generováním různého počtu POST požadavků na `/api/archives`:

- 100 požadavků – 3,73 sekundy
- 200 požadavků – 8,49 sekund
- 1000 požadavků – 40,44 sekund

Pruměrný čas zpracování požadavku na vytvoření archivu je přibližně 40 milisekund při relativně velkém zatížení.

Další test rychlosti vyhodnocuje GET požadavky na `/api/archives`. Tento požadavek získává seznam informací o archivech dostupných na službe. Je možné nastavovat různé parametry v query string URL požadavku na API, čímž bude ovlivněna rychlost zpracování. U těchto testů bylo na serveru uložených **100 archivů**. Pro všechny testy v této kategorii bylo asynchronně posláno 10000 požadavků:

- bez parametrů - všechny výsledky – 44 sekund
- parametr *name* s neexistujícím archivem – žádné výsledky – 16 sekund
- parametr *limit 10* – 38 sekund
- parametr *skip 50* – polovina výsledků – 42 sekund

Průměrná doba zpracování 1 požadavku je přibližně 4 milisekundy, pokud budou požadavky vyžadovat celou kolekci archivů.

Doba zpracování bude také záležet na počtu archivů na serveru. Výsledky testování při **1000 archivech** na serveru pro 10000 současných požadavků:

- bez parametrů - všechny výsledky – 195 sekund
- parametr *name* s neexistujícím archivem – žádné výsledky – 39 sekund
- parametr *limit 10* – 200 sekund
- parametr *skip 500* – polovina výsledků – 195 sekund

Průměrná doba zpracování se zvýšila na zhruba 20 milisekund pro query bez parametrů. Pro desetinásobný počet prohledávaných archivů se doba zpracování zvýšila pětkrát.

Kromě zatížení serveru a počtu archivů bude rychlost aplikace také ovlivněna počtem vláken přiřazených službě (parametr Sanicu *workers*) a použitý hardware. Výsledky jsou proto pouze orientační.

6.2 Archivace stránek z Deep webu

Tato sekce se zaměřuje na testování služeb na stránkách z *Deep webu* s využitím anonymní sítě Tor a onion routing.

Deep web je označení pro obsah na webu, který je skrytý a vyhledávače ho neindexují. Někdy se také označuje jako skrytý web (*Hidden web*). Obsah je možné získat znalostí přesné URL adresy. S pojmem Deep web dále souvisí tzv. *Dark web*, nebo-li temný web. Přístup do této sítě je podmíněn použitím Toru, který zajišťuje velkou úroveň anonymity díky onion směrování[1]. Dark web je proto lákadlo na provozování nelegálních činností.

Vybrané adresy k testování jsou součástí pseudodomény .onion a jsou dostupné pouze přes Tor. Bylo vybráno 50 adres z webů zaměřených na kolekci odkazů a wiki stránek, jako například *thehiddenwiki*[2]. Ačkoliv byla snaha vybrat stránky relativně slušné a prezentovatelné, stránky stále mohou obsahovat nelegální a citlivý obsah.

Testované stránky z deep webu jsou rozděleny do souborů podle tématiky:

- Vyhledávače stránek deep webu a seznamy s odkazy – `test_link_sites.json` – 10 adres
- Tržní, obchodní, finanční a kryptoměnové weby – `test_finance.json` – 10 adres
- Služby – `test_commercial_services.json` - 10 adres
- Blogy a rádia – `test_blogs.json` – 10 adres
- Politika a konspirační weby – `test_politics.json` – 10 adres

Některé z testovaných adres jsou již zastaralé nebo jsou nedostupné. Servery mohou také odmítnout požadavky od web crawleru. Archivace těchto webů většinou dopadne vypršením času na zpracování nebo vzdálený server přímo přeruší spojení. Ve vyhodnocení testů budou prezentované pouze adresy, na kterých byl server dosažitelný alespoň pro stažení hlavního dokumentu.

Testování využívá třídu TestClient v soboru `test_client.py`. Klient posílá POST požadavky na archivační službu s požadovanými URL k archivaci.

Při testování na několika desítkách stránek může běh skriptu trvat až několik minut. Především archivace stránek z Deep webu trvá poměrně dlouho z důvodu velké režie síťových operací.

Vyhodnocení

Z vybraných adres k testování se podařilo spojit s 23 servery, z toho bylo možné úplně nebo částečně archivovat jejich obsah. Testování proběhlo třikrát a až na drobné odchylky v době zpracování se výsledky nelišily.

Tabulka 6.1 obsahuje stránky a jejich adresy z první skupiny testů. Jedná se o vyhledávače, wiki a seznamy, tedy stránky s poměrně jednoduchou dokumentovou strukturou.

[1] Webové stránky projektu Tor – https://www.torproject.org/
[2] https://thehiddenwiki.org/

Název stránky	URL
DuckDuckGo Search Engine	http://3g2upl4pq6kufc4m.onion/
TORCH – Tor Search Engine	http://xmh57jrzrnw6insl.onion/
Uncensored Hidden Wiki	http://zqktlwi4fecvo6ri.onion/
Seeks Search	http://5plvrsgydwy2sgce.onion/
Onion Wiki	http://wiki5kauuihowqi5.onion/
The Hidden Wiki	http://kpvz7ki2v5agwt35.onion/
TorLinks	http://torlinkbgs6aabns.onion/
Hidden Wiki .Onion Urls	http://jh32yv5zgayyyts3.onion/

Tabulka 6.1: Archivované stránky z testové sady test_link_sites

Tabulka 6.2 obsahuje stránky a jejich adresy z druhé skupiny testů. V této skupině jsou stránky zaměřené na finance a kryptoměny.

Název stránky	URL
EasyCoin	http://easycoinsayj7p5l.onion/
WeBuyBitcoins	http://jzn5w5pac26sqef4.onion/
OnionWallet	http://ow24et3tetp6tvmk.onion/
ccPal Store	http://3dbr5t4pygahedms.onion/
HQER	http://y3fpieiezy2sin4a.onion/
Counterfeit USD	http://qkj4drtgvpm7eecl.onion/

Tabulka 6.2: Archivované stránky z testové sady test_finance

Tabulka 6.3 obsahuje stránky a jejich adresy z třetí skupiny testů. V této skupině jsou stránky zaměřené na obchod a služby.

Název stránky	URL
Tor Market Board	http://6w6vcynl6dumn67c.onion/
UK Guns and Ammo	http://tuu66yxvrnn3of7l.onion/
Hidden BetCoin	http://hbetshipq5yhhrsd.onion/

Tabulka 6.3: Archivované stránky z testové sady test_commercial_services

Tabulka 6.4 obsahuje stránky a jejich adresy ze čtvrté skupiny testů. V této skupině jsou blogy, rádia a news weby.

Název stránky	URL
Beneath VT	http://74ypjqjwf6oejmax.onion/
Deep Web Radio	http://76qugh5bey5gum7l.onion/
All the latest news for tor	http://newsiiwanaduqpre.onion/

Tabulka 6.4: Archivované stránky z testové sady test_blogs

Tabulka 6.5 obsahuje stránky a jejich adresy z páté skupiny testů. Tato sada testů obsahuje stránky zaměřené na politiku a informační/dezinformační weby.

Název stránky	URL
Bugged Planet	http://6sgjmi53igmg7fm7.onion/
Example rendezvous points page	http://duskgytldkxiuqc6.onion/
Common Sense by Thomas Paine	http://duskgytldkxiuqc6.onion/comsense.html

Tabulka 6.5: Archivované stránky z testové sady test_politics

Příklad archivované stránky `Hidden Wiki .Onion Urls` je zachycen na snímku obrazovky 6.1. Obsah stránky byl extrahovaný z archivu a zobrazen v prohlížeči Firefox. V porovnání se stránkou staženou a zobrazenou přímo v prohlížeči Tor Browser vypadá vizuálně stejně.

Obrázek 6.1: Snímek obrazovky stránky OnionList

Další výsledné snímky z testování lze nalézt v příloze B. I pro tyto archivy platí, že jejich obsah přesně reprezentuje reálné stránky. Pro stránku `ccPal Store` je výsledná rekonstrukce na snímku obrazovky B.1. Na snímku obrazovky B.2 je zrekonstruován online obchod se zbraněmi `UK Guns and Ammo`. Dále na snímku obrazovky B.3 je rekonstrukce stránky rádiostanic Deep webu `Deep Web Radio`. Z testů páté skupiny je na snímku obrazovky B.4 rekonstrukce blogového webu `Common Sense` autora Thomas Paine.

Kapitola 7

Závěr

V rámci této bakalářské práce jsem si nastudoval problematiku hypertextových protokolů a exportní a archivační formáty MHTML a MAFF. Dále jsem představil principy frameworků pro archivaci webu. Pro řešení archivace jsem zvolil archivační framework Lemmiwinks.

Protože framework řeší dílčí problémy modulárně, byla aplikace rozdělena do 2 samostatných služeb - archivace a stahování. Obě tyto webové služby jsou navrženy tak, aby mohly být vyvíjeny samostatně. Dohromady tvoří systém, který zajišťuje archivaci a rekonstrukci webu podobně jako konzolové řešení využívající Lemmiwinks. Služby je možné využít pomocí aplikačního rozhraní vycházejícího z principů REST API. K návrhu rozhraní byl použit online nástroj Swagger a OpenAPI 3.0. Bylo navrženo rozhraní pro webové prohlížeče, které umožňuje prohlížet a stahovat existující archivy ve formě HTML stránek. Dále bylo navrženo JSON API, které umožňuje prohledávat, přidávat, stahovat a mazat archivy. Přístup k API je chráněn autentizací pomocí jména a hesla nebo token autorizace.

Služby byly implementované v jazyce Python 3.6 s využitím archivačního frameworku Lemmiwinks. Klientská část frameworku byla částečně upravena tak, aby vyhovovala návrhu služeb. Serverová implementace obou služeb využívá asynchronní web server Sanic, který dosahuje vyššího výkonu než běžné synchronní implementace.

Implementace byla otestovaná asynchronním klientem s důrazem na měření rychlosti zpracování požadavků při větším zatížení. Dále byla archivace testovaná na vzorku stránek z deep webu, přičemž u úspěšných archivací byly uvedeny příklady snímků obrazovky výsledné rekonstrukce ve webovém prohlížeči.

Literatura

[1] Bishop, M.: *Hypertext Transfer Protocol Version 3 (HTTP/3)*. 2019, [Online; navštíveno 29.7.2019].
URL https://quicwg.org/base-drafts/draft-ietf-quic-http.html

[2] D. Kristol, L. M.: *RFC 2109 HTTP State Management Mechanism*. 1997, [Online; navštíveno 18.10.2018].
URL https://tools.ietf.org/html/rfc2109

[3] Erl, T.: *SOA: Principles of Service design*. 2008, [Online; navštíveno 26.10.2018].
URL http://serviceorientation.com/index.php/serviceorientation/index

[4] Fielding, R.: *Architectural Styles and the Design of Network-based Software Architectures*. Dizertační práce, University of California, 2000.
URL https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[5] Fielding, R. T.: *Representational State Transfer (REST)*. 2000, [Online; navštíveno 27.10.2018].
URL https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[6] Graham Leggett: *Apache Module mod_proxy*. [Online; navštíveno 29.1.2019].
URL http://httpd.apache.org/docs/2.0/mod/mod_proxy.html

[7] *Cacheable responses*. 2019, [Online; navštíveno 29.7.2019].
URL https://developer.mozilla.org/en-US/docs/Glossary/cacheable

[8] J. Palme, A. H.: *MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)*. 1999, [Online; navštíveno 2.11.2018].
URL https://tools.ietf.org/html/rfc2557

[9] Josefsson, S.: *The Base16, Base32, and Base64 Data Encodings*. 2006, [Online; navštíveno 4.4.2019].
URL https://tools.ietf.org/html/rfc4648

[10] Leech, M.: *SOCKS Protocol Version 5*. 1996, [Online; navštíveno 11.5.2019].
URL https://tools.ietf.org/html/rfc1928

[11] Luotonen, K., Ari a Altis: World-Wide Web proxies. *Computer Networks and ISDN Systems*, ročník 27, 11 1994: s. 147–154, doi:10.1016/0169-7552(94)90128-7.

[12] M. Belshe, R. P.: *RFC 7540 Hypertext Transfer Protocol Version 2*. 2015, [Online; navštíveno 29.7.2019].
URL https://tools.ietf.org/html/rfc7540

[13] *The MAFF specification*. 2018, [Online; navštíveno 11.5.2019].
URL http://maf.mozdev.org/maff-specification.html

[14] Postel, J.: Transmission Control Protocol. RFC 793, RFC Editor, September 1981.
URL https://www.rfc-editor.org/rfc/rfc793.txt

[15] R. Fielding, J. G.: *RFC 2616 Hypertext Transfer Protocol – HTTP/1.1*. 1999, [Online; navštíveno 14.10.2018].
URL https://tools.ietf.org/html/rfc2616

[16] Sanic 19.03.1 documentation. [Online; navštíveno 11.4.2019].
URL https://sanic.readthedocs.io/en/latest/

[17] Serečun, V.: *Automatic web page reconstruction*. Diplomová práce, Brno University of Technology, 2018.
URL http://www.fit.vutbr.cz/study/DP/DP.php.cs?id=21138&file=t

[18] Solomon, B.: *Async IO in Python: A Complete Walkthrough*. 2019, [Online; navštíveno 4.4.2019].
URL https://realpython.com/async-io-python/

[19] *Swagger*. 2019, [Online; navštíveno 11.7.2019].
URL https://swagger.io/

[20] T. Dierks, E. R.: *RFC 5246 The Transport Layer Security (TLS) Protocol*. 2008, [Online; navštíveno 14.10.2018].
URL https://tools.ietf.org/html/rfc5246

[21] *Installing Tor on Debian/Ubuntu*. 2019, [Online; navštíveno 11.5.2019].
URL https://2019.www.torproject.org/docs/debian.html.en

[22] Wouters, T.: *Global Interpreter Lock*. 2017, [Online; navštíveno 10.4.2019].
URL https://wiki.python.org/moin/GlobalInterpreterLock

Přílohy

Příloha A

OpenAPI archivační a stahovací služby

```yaml
openapi: 3.0.0
# Added by API Auto Mocking Plugin
info:
  description: Lemmiwinks archive service API
  version: "1.0.0"
  title: Lemmiwinks archive service API
  contact:
    email: xmatus31@stud.fit.vutbr.cz
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
tags:
  - name: user
    description: |
      Operations available to authorized users
  - name: guest
    description: |
      Operations available to guests
paths:
  /:
    get:
      tags:
        - user
        - guest
      operationId: get_index
      description: |
        Main page containing basic information and navigation.
      responses:
        '200':
          description: |
            HTML page with web navigation. Contains a login form for guests
            and introduction text for logged in users.
          content:
            text/html:
              schema:
                type: string
  /login:
    get:
      tags:
        - user
```

```
      - guest
    operationId: get_login
    description: |
      Login page containing login form.
    responses:
      '200':
        description: |
          HTML page with login form. Contains a login form.
        content:
          text/html:
            schema:
              type: string
  post:
    tags:
      - user
      - guest
    operationId: post_login
    description: Posts login credentials.
    responses:
      '200':
        description: |
          Login form with error message on incorrect credentials.
        content:
          text/html:
            schema:
              type: string
      '302':
        description: |
          Successful login, redirects
        headers:
          Location:
            schema:
              type: string
/logout:
  get:
    tags:
      - user
      - guest
    operationId: get_logout
    description: |
      Logout the current user.
    responses:
      '302':
        description: |
          Successful logout, redirects
        headers:
          Location:
            schema:
              type: string
/archives:
  get:
    tags:
      - user
    summary: Page with archives
    operationId: get_archives
    description: |
      Gets collection of archives. Archives are sorted by creation time.
    parameters:
      - in: query
```

```
        name: page
        description: |
          Specify archive apge. Page contains 10 archives. If no page is
          queried, returns first page.
        required: false
        schema:
          type: integer
    responses:
      '200':
        description: |
          HTML page containing list of archives and a POST form for adding
          new archive.
        content:
          text/html:
            schema:
              type: string
      '400':
        description: |
          HTML page with bad request error info.
      '401':
        description: Unauthorized
  post:
    tags:
      - user
    summary: |
      Creates new archive from HTML form POST request
    operationId: post_archives
    description: |
      Creates new archive from given URL in HTML form. You can specify
      additional archiving options.
    responses:
      '201':
        description: item created
      '400':
        description: invalid input, object invalid
      '401':
        description: Unauthorized
    requestBody:
      content:
        application/x-www-form-urlencoded:
          schema:
            $ref: '#/components/schemas/FormPostArchive'
      description: URL to archive with options.
/archives/{id}:
  get:
    tags:
      - user
    summary: Gets an archive
    operationId: get_archive_item
    description: Gets archive item
    parameters:
      - in: path
        name: id
        required: true
        schema:
          type: integer
          format: int32
    responses:
      '200':
```

```
          description: |
            HTML page with archive details and download link.
          content:
            text/html:
              schema:
                type: string
        '404':
          description: |
            HTML page with not found message.
          content:
            text/html:
              schema:
                type: string
        '401':
          description: Unauthorized

  /archives/{id}/{filename}:
    get:
      tags:
        - user
      summary: Downloads archive MAF file
      operationId: get_archive_file
      description: |
        downloads the file
      parameters:
        - in: path
          name: id
          required: true
          schema:
            type: integer
            format: int32
        - in: path
          name: filename
          required: true
          schema:
            type: string
            example: 'RedditArchive_a1b2c3d4.maff'
      responses:
        '200':
          description: Archive details with a link to file download
          content:
            application/x-maff:
              schema:
                type: string
                format: binary
        '404':
          description: Archive not found
        '401':
          description: Unauthorized

  /api/archives:
    get:
      tags:
        - user
      summary: Searches and gets existing archives
      operationId: api_get_archives
      description: |
        Search archives using query parameters.
      parameters:
```

```
      - in: query
        name: name
        description: pass an optional search string for looking up archive name
        required: false
        schema:
          type: string
      - in: query
        name: skip
        description: number of records to skip for pagination
        schema:
          type: integer
          format: int32
          minimum: 0
      - in: query
        name: limit
        description: maximum number of records to return
        schema:
          type: integer
          format: int32
          minimum: 0
          maximum: 50
    responses:
      '200':
        description: search results matching criteria
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/ArchiveDetail'
      '400':
        description: bad input parameter
      '401':
        description: Unauthorized
  post:
    tags:
      - user
    summary: Creates new archive
    operationId: api_post_archives
    description: |
      Creates new archive from given URLs. You can specify
      additional archiving options
    responses:
      '201':
        description: item created
        headers:
          Location:
            schema:
              type: string
              format: url
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/APIArchiveCreated'
      '400':
        description: invalid input, object invalid
      '401':
        description: Unauthorized
    requestBody:
```

```
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/APIPostArchive'
      description: Inventory item to add
  /api/archives/{id}:
    get:
      tags:
        - user
      summary: Gets archive details
      operationId: api_get_archive_item
      description: |
        gets an archive file with information and link to archive
        download
      parameters:
        - in: path
          name: id
          required: true
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: Archive details with a link to file download
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ArchiveDetail'
        '404':
          description: Archive not found
        '401':
          description: Unauthorized
    delete:
      tags:
        - user
      summary: Deletes an archive
      operationId: api_delete_archive_item
      description: Deletes an archive given by id
      parameters:
        - in: path
          name: id
          required: true
          schema:
            type: integer
            format: int32
      responses:
        '204':
          description: Archive deleted sucessfully
        '401':
          description: Unauthorized

  /api/archives/{id}/{filename}:
    get:
      tags:
        - user
      summary: Downloads archive MAF file
      operationId: api_get_archive_file
      description: |
        downloads the file
```

```yaml
      parameters:
        - in: path
          name: id
          required: true
          schema:
            type: integer
            format: int32
        - in: path
          name: filename
          required: true
          schema:
            type: string
            example: 'RedditArchive_a1b2c3d4.maff'
      responses:
        '200':
          description: Archive details with a link to file download
          content:
            application/x-maff:
              schema:
                type: string
                format: binary
        '404':
          description: Archive not found
        '401':
          description: Unauthorized

components:
  securitySchemes:
    basicAuth:
      type: http
      scheme: basic
  schemas:
    ArchiveDetail:
      type: object
      required:
        - file
        - name
        - id
        - ctime
        - size
        - href_download
        - href_detail
        - href_download_api
        - href_detail_api
      properties:
        file:
          type: string
          example: 'RedditAll_p41bdo14.maff'
        name:
          type: string
          default: 'unnamed'
          example: 'RedditAll'
        id:
          type: string
          example: 'p41bdo14'
        ctime:
          type: integer
          format: int32
          example: 1560541146
```

```
        size:
          type: integer
          format: int32
          example: 458101
        href_download:
          type: string
          format: url
          example: '/archives/p41bdo14/RedditAll_p41bdo14.maff'
        href_detail:
          type: string
          format: url
          example: '/archives/p41bdo14'
        href_download_api:
          type: string
          format: url
          example: '/api/archives/p41bdo14/RedditAll_p41bdo14.maff'
        href_detail_api:
          type: string
          format: url
          example: '/api/archives/p41bdo14'
    FormPostArchive:
      type: object
      required:
        - url
      properties:
        url:
          type: string
          format: url
        name:
          type: string
          default: 'Unnamed archive'
          example: 'Reddit archive'
          pattern: ^[a-zA-z0-9-]+$
        forceTor:
          type: string
    APIPostArchive:
      type: object
      required:
        - urls
        - name
        - forceTor
      properties:
        urls:
          type: array
          items:
            type: string
            format: url
        name:
          type: string
          default: 'Unnamed archive'
          example: 'Reddit archive'
          pattern: ^[a-zA-z0-9-]+$
        forceTor:
          type: boolean
          default: False
        headers:
          type: object
          properties:
            User-Agent:
```

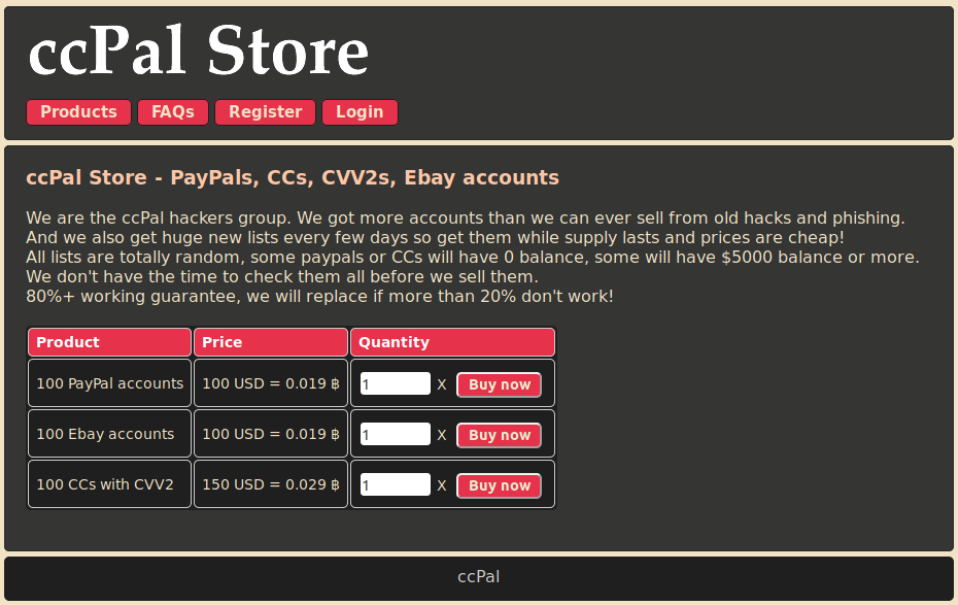
```
              type: string
            Accept-Language:
              type: string
# Added by API Auto Mocking Plugin
servers:
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/matusadam/lemmiwinks_archiver_service/1.0.0
```

```
openapi: 3.0.0
# Added by API Auto Mocking Plugin
info:
  description: Downloader service API
  version: "1.0.0"
  title: Downloader service API
  contact:
    email: xmatus31@stud.fit.vutbr.cz
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
tags:
  - name: users
    description: Auth users calls
  - name: guests
    description: All users calls
paths:
  /:
    get:
      tags:
        - guests
        - users
      summary: information page
      operationId: getIndex
      description: |
        HTML page with basic information about this service.
      responses:
        '200':
          description: 'HTML index page'
  /api/download:
    post:
      tags:
        - users
      summary: downloads a resource
      operationId: download
      description: |
        A request to download a resource given by its URL. You can specify more
        options in the request body.
      responses:
        '200':
          description: downloaded resource
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/DownloadedResource'
        '400':
          description: bad input parameter
        '401':
          description: requires authorization header with correct auth token
      requestBody:
        content:
```

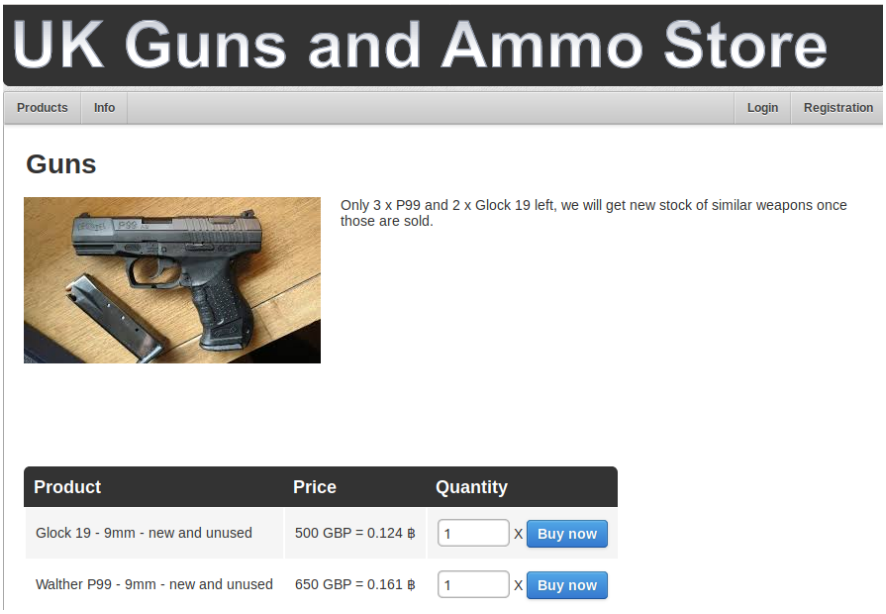
```
          application/json:
            schema:
              $ref: '#/components/schemas/DownloadReq'
        description: Inventory item to add
components:
  schemas:
    DownloadReq:
      type: object
      required:
        - resourceURL
        - useTor
        - headers
      properties:
        resourceURL:
          type: string
          format: url
          example: 'http://www.example.com/pic.png'
        useTor:
          type: boolean
        headers:
          type: object
          properties:
            User-Agent:
              type: string
            Accept-Language:
              type: string
    DownloadedResource:
      type: object
      required:
        - content-type
        - url_and_status
        - data
      properties:
        content-type:
          type: string
          example: 'application/jpg'
        url_and_status:
          type: array
          items:
            type: array
            items:
              type: string
        data:
          type: string
          format: base64
# Added by API Auto Mocking Plugin
servers:
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/matusadam/lemmiwinks_downloader_service/1.0.0
```

Příloha B

Snímky obrazovky archivů z Deep webu

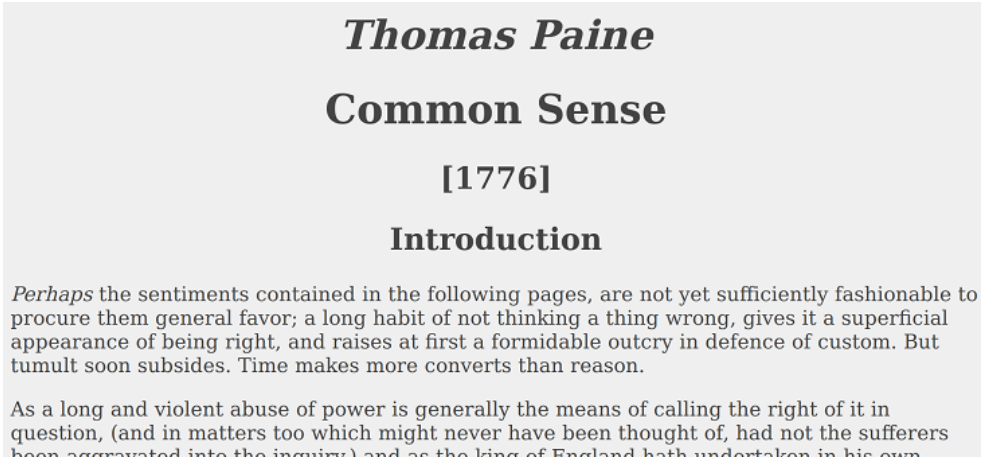


Obrázek B.1: Snímek obrazovky stránky ccPal Store



Obrázek B.2: Snímek obrazovky stránky UK Guns and Ammo

Obrázek B.3: Snímek obrazovky stránky Deep Web Radio



Thomas Paine

Common Sense

[1776]

Introduction

Perhaps the sentiments contained in the following pages, are not yet sufficiently fashionable to procure them general favor; a long habit of not thinking a thing wrong, gives it a superficial appearance of being right, and raises at first a formidable outcry in defence of custom. But tumult soon subsides. Time makes more converts than reason.

As a long and violent abuse of power is generally the means of calling the right of it in question, (and in matters too which might never have been thought of, had not the sufferers

Obrázek B.4: Snímek obrazovky stránky Common Sense