České vysoké
učení technické
v Praze

Fakulta
Strojní

Diplomová
práce

Rozšíření aplikace pro monitorování dostupností zařízení na síti

2024

Vypracoval: Jan Veselý
Vedoucí: Ing. Matouš Cejnek Ph.d.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Veselý** Jméno: **Jan** Osobní číslo: **483223**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídicí techniky**

Studijní program: **Automatizační a přístrojová technika**

Specializace: **Automatizace a průmyslová informatika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Rozšíření aplikace pro monitorování dostupností zařízení na síti

Název diplomové práce anglicky:

Extension of the application for monitoring the availability of devices on the network

Pokyny pro vypracování:

- Vytvořte aplikaci, která bude testovat síťové zařízení pomocí ICMP paketů a zjišťovat jejich dostupnost. Data se budou ukládat do databáze. Využijte přitom již vytvořenou logiku částečně funkční aplikace.
- Připravte aplikaci tak, aby byla jednoduše instalovatelná a spustitelná na různých platformách - Windows, Unix
- Nasaďte a otestujte aplikaci v kontejnerizovaném prostředí
- Proveďte teoretickou analýzu aplikace a jejího nasazení včetně zabezpečení komunikace.

Seznam doporučené literatury:

Docker for Beginners: The Ultimate Step-by-Step Guide to Docker (ISBN: 9781801490498)

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Matouš Cejnek, Ph.D. U12110.3

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

Datum zadání diplomové práce: **27.10.2023** Termín odevzdání diplomové práce: **19.01.2024**

Platnost zadání diplomové práce: _____

Ing. Matouš Cejnek, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Tomáš Vyhlídal, Ph.D. podpis vedoucí(ho) ústavu/katedry	doc. Ing. Miroslav Španiel, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

Datum převzetí zadání Podpis studenta

Prohlášení:

Prohlašuji, že jsem diplomovou práci na téma “Rozšíření aplikace pro monitorování dostupností zařízení na síti“ vypracoval samostatně pod vedením Ing. Matouše Cejnka, Ph.D. a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

V Praze dne: 8. ledna 2024 Podpis:

Obsah

Abstrakt

Tato diplomová práce se zaměřuje na rozvoj a implementaci rozšíření aplikace pro monitorování dostupnosti zařízení na síti. Práce začíná teoretickým základem, pokrývajícím základy síťové komunikace a bezpečnosti, a následně se věnuje praktické implementaci navrženého řešení. Klíčové aspekty práce zahrnují rozvoj metod autentizace, zabezpečení datového přenosu a integrace těchto prvků do stávajícího systému. Výsledkem je robustnější a bezpečnější systém pro monitorování sítí, který je schopen identifikovat a zaznamenávat potenciální bezpečnostní hrozby.

Klíčová slova: Monitorování sítí, Zabezpečení sítě, Autentizace, Rozšíření aplikace, Bezpečnostní protokoly, Síťová komunikace, Databázový design, Implementace softwaru, Testování softwaru, Síťová infrastruktura.

Abstract

This thesis focuses on the development and implementation of an extension for a network device monitoring application. The work begins with a theoretical foundation covering the basics of network communication and security, followed by the practical implementation of the proposed solution. Key aspects of the thesis include the development of authentication methods, securing data transmission, and integrating these elements into the existing system. The result is a more robust and secure network monitoring system capable of identifying and recording to potential security threats.

Keywords: Network Monitoring, Network Security, Authentication, Application Extension, Security Protocols, Network Communication, Database Design, Software Implementation, Software Testing, Network Infrastructure.

1. Úvod

V současné době, kdy je síťová komunikace a monitorování zařízení na síti klíčové pro udržení bezpečnosti a efektivity v podnikových i osobních sítích, se moje diplomová práce zaměřuje na rozšíření aplikace pro monitorování dostupnosti zařízení na síti. Tato práce se zabývá nejen technickými aspekty vývoje a implementace takového systému, ale také klade důraz na zabezpečení a autentizaci, které jsou v dnešní digitálně propojené době nezbytné.

Hlavním cílem mé práce je navrhnout a implementovat rozšíření stávající aplikace tak, aby byla schopna monitorovat a reportovat stav síťových zařízení. Toto rozšíření zahrnuje zavedení metod autentizace, zabezpečení přenosu dat. Kromě technické realizace se práce věnuje také potenciálním rizikům a bezpečnostním hrozbám, které mohou vzniknout při používání aplikace, která by nebyla zabezpečená.

Dalším klíčovým aspektem mé práce je praktická implementace navrhovaného řešení. Zde se soustředím na detailní popis vývoje aplikace, včetně použitých technologií, návrhu databáze, a postupů, které zajistí hladký chod a snadnou správu systému. Praktická část také ilustruje, jak lze teoretické aspekty zabezpečení a síťové komunikace aplikovat v reálném prostředí.

Tato diplomová práce je strukturována tak, aby postupně vedla čtenáře od základních konceptů až po konkrétní implementaci a nasazení systému. V první části se zaměřuji na teoretický základ, včetně zabezpečení komunikace, autentizace a protokolů používaných pro bezpečný přenos dat. Následně se věnuji praktické části, kde detailně rozebírám konstrukci a funkčnost aplikace, a to jak z hlediska backendu, tak i frontendu. Závěrečná část práce se pak soustředí na nasazení a možný scénář použití aplikace v reálných podmínkách.

Cílem této práce je tedy nejen rozšíření funkcionality stávající aplikace, ale také představení komplexního pohledu na výzvy a řešení spojené s monitorováním a zabezpečením síťových zařízení.

2. Zabezpečení komunikace

Zabezpečení a šifrování je v dnešní době naprosto nepostradatelnou součástí webové komunikace. Zatímco v roce 2017 bylo využíváno přesměrování na zabezpečenou komunikaci pouze u 20% [1] webových stránek, tak v roce 2018 už to bylo cca 50% [2]. Celkový trend je zřejmý – uživatelé, provozovatelé ale i vývojáři webových prohlížečů si uvědomují důležitost ochrany osobních údajů.

Vzhledem k tomu, že běžní uživatelé nepoužívají internet pouze s cílem informace získávat, ale internet slouží i pro výměnu informací, tak získává kryptografie na své důležitosti. Do dob šifrování mohl kdokoliv přenášené informace zaznamenat a následně přečíst. Bohužel i nyní se najdou webové stránky, které nepoužívají zabezpečení pomocí šifrování. Z mé zkušenosti to ale nechává koncové uživatele vcelku v klidu a neuvědomují si možné důsledky. Je tedy potřeba, aby se provozovatelé webových aplikací chovali zodpovědně a své uživatele ochránili.

Naše aplikace přímo nesdílí žádná citlivá data a v případě, že by se systém nacházel v interní síti je pravděpodobné, že pokud by nebyla infikována tato část, tak by nehrozilo zneužití dat. Využíváme ale možnosti autentizace uživatelů a přiřazování rolí. Nechceme, aby kdokoliv mohl do systému zasahovat, měnit data a nastavení. V tomto případě přichází nejslabší článek – uživatel. Pokud bychom provedli obecnou analýzu rizik, tak únik přihlašovacích údajů do systému by mohl zapříčinit například ztrátu zaznamenaných dat. Pro většinu uživatelů ale bude mít nástroj využití v možnosti monitoringu sítě a dlouhodobá historie dat nemá příliš velkou hodnotu. Musíme ale vzít v potaz, že uživatelé běžně používají jedno uživatelské jméno a heslo do více systému. A pak bychom mohli napomoci útočníkům v síti posunout svá oprávnění na vyšší úroveň díky krádeži těchto přihlašovacích údajů.

2.1. Definice pojmů

- Klíč je skupina znaků (obvykle náhodných). Šifrovací algoritmy využívají tohoto klíče k transformaci dat – zakódování zprávy. Tento a nebo další klíč je pak využíván i k dešifrování.
- Veřejný klíč – jak už název napovídá, se jedná o veřejně dostupný klíč, využívají ho asymetrické šifry, může sloužit pro případ šifrování i dešifrovaní.
- Soukromý klíč – u asymetrického šifrování slouží pro dešifrování soukromé zprávy, nebo k šifrování u podpisů
- Symetrické šifrování – tento systém používá k šifrování i dešifrování stejný klíč
- Asymetrické šifrování – systém používá odlišné klíče pro šifrování a dešifrování [3].

3. Autentizace

Jedna stránka zabezpečení aplikace se zabývá šifrováním přenášených informací, druhá ověřování uživatelů. Jakmile uživatel využije API, ať už si zobrazí status webové aplikace, zadá jakékoliv data a pošle je na server, nebo se přihlásí, tak přenášená data mezi serverem a uživatelem jsou (a nebo by měla být) šifrována. Ale je potřeba zajistit i část autentizace uživatelů, aby v systému mohl uživatel provádět pouze změny k němu příslušejících.

Neautentizují se pouze uživatelé, ale i dílčí části systému (Workery – bude zmíněno dále). Subsystém se musí autorizovat, že má přístup do hlavního systému. Teoreticky je možné využívat stejné metody jak pro ověřování uživatelů, tak subsystémů. V běžné praxi to ale není vhodné. Uživatelé jsou z běžné praxe zvyklí na užívání přihlašovacího jméno a hesla. Zatímco pro subsystém se využívá autentizace přes token. Tyto dvě metody se ale lehce překrývají.

3.1. OAuth2

OAuth je zkratka pro Open Authorization. Jedná se o průmyslový standard pro online autorizaci. Byl definován v roce 2012 a nyní je využíván firmami jako Google, Facebook, Microsoft a mnoha dalšími. [4]

3.1.1. Pojmy:

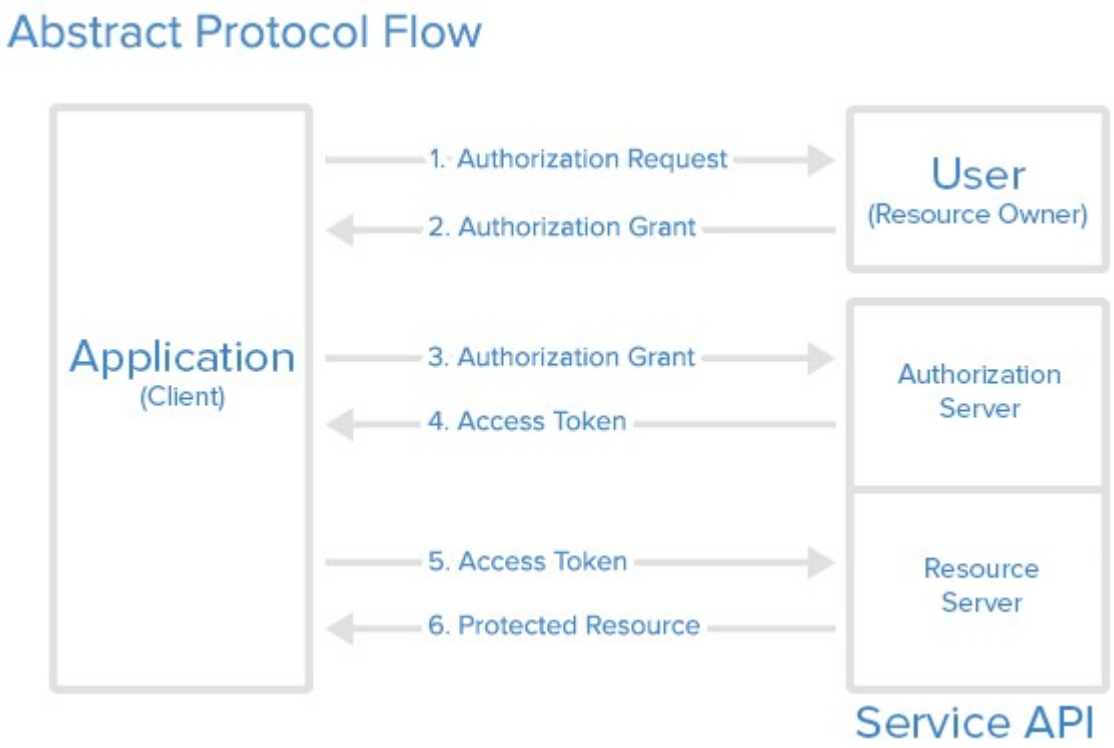
- **Resource owner** – uživatel nebo systém
- **Resource server** – jedná se o server, na kterém jsou uložena data uživatele včetně jejich oprávnění. Tyto oprávnění uživatele se nazývají **scope**. Přijímá požadavky na přístup a vrací mu jeho prostředky (data)
- **Client** – aplikace, která přistupuje k uživatelským datům – v našem případě se jedná o GUI
- **Authorization** server – server, který autentizuje jednotlivé uživatele, odpovědí je **access token** [4].

Na nejjednodušší úrovni funguje OAuth autentizace na principu uživatelských přihlašovacích údajů – client ID a client secret.
Client ID je unikátní uživatelský identifikátor, může se jednat o číselný identifikátor, uživatelské jméno nebo emailovou adresu. Client secret je potom přihlašovací heslo, opět se může jednat o jakýkoliv datový typ.
Pokud autorizační server ověří client ID a client secret, pak je výsledkem Access Token, který slouží pro ověřování jednotlivých requestů (internetových žádostí) [4].

3.1.2. Scopes

Scopes je velmi důležitá součást OAuth. Při ověřování se uživatelům přiřazují jednotlivá oprávnění, která mají mít. Neznamená to, že se jedná pouze o jejich roli – například administrátor. Ale můžeme definovat přímo jejich možnosti k přistupování prostředků. Díky tomu můžeme nastavit jednotlivé kategorie oprávnění individuálně každému uživateli – například čtení a editace. Dalšímu uživateli mohu nastavit čtení, editace, mazání apod [4].

3.1.3. Princip



Obrázek 1 - Popis OAuth2 autentizace [4]

Princip autorizace si můžeme prohlédnout na (Obrázek 1), dle zmíněného schématu si ho můžeme rozdělit do několika rovin. První rovinou je uživatel – předpokládejme, že uživatel žádá o přístup k soukromým datům skrze aplikaci. Aplikace vyžaduje oprávnění, a tak vyšle Authorization Request (žádost o autorizaci) uživateli. Uživatel se autorizuje svými přihlašovacími údaji.

Druhou rovinou je Authorization Server. Aplikace potřebuje Access Token, kterým následně autorizuje své požadavky vůči Resource Server. Aplikace již zná od uživatele přístupové údaje a pošle je Autorizačnímu serveru, který je ověří. Pokud jsou validní, pak vrátí Access Token, který mimo jiné obsahuje příslušné scopes uživatele.
Jakmile aplikace zná Access Token může zažádat o data Resource Server a ten vrátí soukromá data.

První čtyři kroky se zabývají získáním přístupového tokenu. Do této doby jsem zmiňoval, že aplikace potřebuje přístupové údaje uživatele, aby ho mohla autorizovat. OAuth2 protokol definuje tyto tři typy ověření:

- Authorization Code – používají systémy, které přímo komunikují se serverem. Ověření probíhá pomocí identifikátoru. Pokud má uživatel autorizační kód, pak ho vymění za Access Token.
- Client Credentials – používají se aplikacemi, které mají přístup k API. Běžně se jedná o přihlašovací jméno a heslo.
- Device Code – používají se pro zařízení, která nemají prohlížeč nebo mají omezený možnosti získat přístupový token [4].

3.1.4. Refresh token

Po celou dobu jsme mluvili o Access Token, který umožňuje přístup k prostředkům uživatele na Resource Serveru. Tento token zpravidla má svojí omezenou platnost (mnohdy pouze v řádech minut). Po expiraci tohoto tokenu se při žádosti na API vyskytne chyba Invalid Token Error. Tato platnost tokenu má především bezpečnostní charakter. Pokud by nám někdo token dokázal ukrást, pak bychom neměli žádný prostředek na jeho zneaktivnění.

V praxi se tedy při ověření uživatelských údajů vrací dva tokeny – Access Token a Refresh Token. Po expiraci Access Tokenu si můžeme pomocí Refresh Tokenu vyžádat nový Access Token [4]. Následně dostaneme oba dva nové tokeny. Refresh Token má také svojí expirační lhůtu. Ale nyní se jedná o značně delší časové období – například 7 dní. Pokud by nám někdo ukradl Refresh Token, sice by díky němu mohl obnovit Access Token, ale následně by obnovení neprošlo u uživatele. Respektive vždy funguje pouze prvnímu, který o nový token zažádá. Následně se vygenerují oba tokeny nové.

3.2. Alternativy

Při autentizaci můžeme využít ještě dalších metod namísto OAuth2. Nejjednodušší variantu shledávám u HTTP Basic Authentication. Jedná se o nejvíce přímočarou metodu ověřování. Odesílatel posílá v HTTP requestu v hlavičce zároveň uživatelské jméno a heslo [5]. Výhodou tohoto řešení je, že není potřeba žádných komplexních systémů. Vzhledem k tomu, že přenos informace o autentizaci je v hlavičce, tak není třeba žádných handshaků a a dalších navazujících komunikacích.

V našem systému ale má několik nevýhod, se kterými bychom si museli poradit: rozšiřitelnost aplikaci a udělování oprávnění. V našem projektu využíváme FastApi, které má již předpřipravené knihovny pro OAuth2 autentizaci. Pokud bychom se vydali cestou autentizace skrze HTTP hlavičku, pak bychom museli napsat vlastní autentizátor, který by kontroloval, zda k danému endpointu má uživatel přístup, nebo nikoliv. [6]
Vzhledem k tomu, že OAuth2 je běžný standard a FastApi již implementovalo tento protokol do svého frameworku, nedává příliš smysl této metody využít.

3.2.1. API klíče

API klíče však již v našem projektu najdou uplatnění. Jedná se o široce využívanou metodu v průmyslu, přestože obecně není považována z bezpečnostního hlediska za nejlepší. V této metodě se využívá unikátně předem vygenerovaného klíče, který je pevně přiřazen k uživateli (a nebo jakémukoliv jinému klientovi). Následně se v HTTP hlavičce používá jako autentizační „token" [6].

Nevýhodou tohoto řešení je, že se token zpravidla příliš často nemění. Tudíž zvyšujeme šanci jeho zneužití. Musíme zajistit, aby byla cesta od klienta k serveru dobře zabezpečená a nemohlo dojít k prozrazení klíče.

Jeho výhodou je lehká implementace a v některých případech se jedná o vhodné řešení, pokud například potřebujeme identifikovat jednotlivé systémy. V případě využívání uživatelského jména a hesla bychom pro jednotlivé identifikování soustavy potřebovali vygenerovat tuto kombinaci pro každý systém.

3.3. Zabezpečení dat

Abychom měli kapitolu autentizace kompletní ještě je potřeba vyřešit zabezpečení dat na straně serveru. Authorization server musí mít k dispozici databázi uživatelských jmen a hesel. Je nevhodné tyto data ukládat v plain textu, protože pak každý, kdo má přístup k databázi, má zároveň přístup i k uživatelským heslům. Jak jsem již zmínil na začátku kapitoly, tak uživatele je potřeba chránit. A to i před námi samotnými správci dat.
K tomu využíváme hashe. Hašovací funkce je algoritmus, který převádí vstupní data do výstupních dat pevné délky. Jedná se o jednosměrnou

transformaci. Vzhledem k tomu, že hash má pevnou délku a délka vstupních dat není nijak omezena, je jednoznačné, že k více vstupním datům může být přiřazen stejný hash (otisk) [7].

Hlavní funkce hashe jsou:

- Nezávisle na délce vstupních dat vždy dojde k vytvoření stejně dlouhého otisku.
- Malou změnou vstupních dat musí dojít k velké změně otisku (při změně jednoho znaku ve vstupních datech, je výsledný otisk na první pohled zcela odlišný).
- Mělo by být zajištěno, aby byla nízká pravděpodobnost, aby dvěma vstupním vzorkům byl přiřazen stejný hash [7].

Pro uchování hesel využíváme těchto otisků. Při ukládání hesla, vytvoříme hash a uložíme do databáze. Pokud přijde požadavek na ověření hesla, pak z něj znovu vytvoříme hash a porovnáme s hashem v databázi.

Jednou funkcí hashe je, že nelze (jednoduše) rekonstruovat zpětně původní obraz vstupních dat. Při ověření uživatele však nezáleží, zda se jedná o konkrétní heslo uživatele, pokud bude výsledkem stejný hash. Zároveň jednou z hlavních funkcí, kterou jsem zmínil výše, je, že dvěma vstupním vzorkům by neměl být přiřazen stejný hash. Pokud tedy známe hash a dokážeme tento hash zpětně „dešifrovat“, pak budeme pravděpodobně znát i původní heslo.

Musíme si ale uvědomit, že uživatelé nepoužívají příliš složitá hesla. A pokud známe použitý hashovací algoritmus, pak můžeme pomocí brute force útoku vyzkoušet všechna potenciální hesla, než najdeme konkrétní hash. Aby to útočníci měli ještě jednodušší, tuto práci provedl někdo již za ně. Jsou dostupné databáze různých hashů pro různé varianty hesel. A teoreticky (i prakticky) stačí ukradený hash projet touto databází [7].

Z tohoto důvodu se při ukládání hesel vytváří zároveň tzv. sůl (salt). Jedná se o prefix k heslu, který se k němu přidá a je uložen v databázi v plain textu zároveň s heslem. Postup ověření je potom následovný:

- Na server přijde požadavek na ověření uživatele (vstupní data jsou uživatelské jméno a heslo)

- Server si najde uživatele a zjistí uloženou sůl k němu přiřazenou
- Server vytvoří sloučeninu hesla a soli, následně vytvoří hash a ten porovná s hodnotou uloženou v databázi.

Výhodou tohoto řešení je, že každý uživatel má jinou sůl. A pokud by došlo k ukradení databáze s uživatelskými jmény a hesly. Pak by bylo potřeba vyřešit všechny varianty hashů pro hesla s konkrétní solí. Je jen velmi malá pravděpodobnost, že by byla k nalezení databáze již vyřešených hashů pro heslo s touto solí.

Teoreticky je pro slabé heslo stále možné získat původní obraz hashe a tím i původního hesla (a nebo jiné heslo se stejným hashem), ale výpočetní výkon není zdarma a útočníkovi zabere nějaké úsilí a prostředky na jeho získání.

3.3.1. Bcrypt

Bcrypt je kryptografická hashovací funkce, které obsahuje poměrně komplikované algoritmy. Pomocí nich se redukuje šance útočníka na prolomení uložených hesel v databázi. Zaměřuje se hlavně na omezení prolomení hesla pomocí dictionary password útoku. Jedná se útok popsaný výše. Bcrypt automaticky přidává k heslu sůl, a díky tomu nemusíme při implementaci zabezpečení na tuto stránku věci myslet [8]. Zároveň jedním hlavním předpokladem je použití silného hesla. Bcrypt umí pomocí funkce nazvané protahování klíčů v případě krátkého hesla natáhnout původní heslo do delšího řetězce a tím sílu hesla posílit [8].

3.4. HTTPS

HTTPS protokol pouze rozšiřuje protokol HTTP zajištující komunikaci mezi serverem a klientem o šifrování. Snižuje se tak riziko odposlouchávání, změny obsahu nebo zneužití osobních údajů. Vzhledem k tomu, že v dnešní době se považuje zabezpečení webových stránek za naprostý standard, tak Google zařadil použití protokolu HTTPS do ranking faktorů – faktory hodnotící webové stránky a tím i indexaci při vyhledávání [9].
V našem případě, protože naše API je čistě webová aplikace, se zaměříme právě na zabezpečení komunikace pomocí HTTPS.

3.4.1. Výhody HTTPS

- Šifrovaná komunikace – nejde pouze o osobní údaje, které zadáváme například při autorizaci, ale i o přenášená data, která by potenciálně mohl útočník zneužít (mzdové údaje, finanční toky apod.)
- Zajištění integrity – útoky man in the middle fungují na principu prostředníka. Útočník se naboural mezi komunikaci a informace předkládá druhé straně. V tomto kroku může útočník pozměnit přenášená data, případně do nich něco přidat – malware, reklamní banner. Samotné šifrování tomuto zcela nepředchází, je nutné zajistit důvěryhodnost certifikátu.
- Ověření identity – díky důvěryhodnému certifikátu můžeme ověřit, že web opravdu patří patřičnému majiteli [9].

3.4.2. Princip

Hlavní rozdíl v jednotlivých protokolech HTTP a HTTPS je přidání vrstvy šifrování pomocí SSL (Secure Sockets Layer) a nebo nověji TLS (Transport Layer Security). Při rozšiřování webových aplikací se začalo čím dál tím více zabývat bezpečností na internetu. Samotný SSL protokol byl představen v roce 1994 [10]. První verze SSL ale obsahovala značené nedostatky. Proto se za oficiální vznik SSL považuje rok 1995, kdy se představila verze SSL 2.0 [10].

Od roku 2011 IETF (The Internet Engineering Task Force) označila verzi SSL 2.0 za zastaralou [10]. IETF doporučila SSLv2 kompletně opustit především díky dalším bezpečnostním nedostatkům, které se v čase objevily (slabé hashování MD5 pro ověřování zpráv, použití stejných klíčů jak pro šifrování zpráv integrity, tak šifrování zprávy). Zpráva integrity se využívá pro ověření, že nebylo s touto zprávou nijak manipulováno [11]. Pokud využívám stejné klíče pro šifrování zprávy i pro ověření, že se zprávou nebylo manipulováno, pak nám stačí zjistit pouze tento jeden klíč.

TLS verze přišlo jako přímý nástupce SSL, vznikl v roce 1999 a přímo tak nahradil SSL verze 3.

3.5. TLS protokol

Vzhledem k tomu, že SSL se v praxi téměř už nepoužívá u webových aplikacích, budu se nadále věnovat pouze protokolu TLS. Ten je navržen, aby zajišťoval tři hlavní aspekty:

- Šifrování: zajištění bezpečného přenosu dat mezi serverem a klientem
- Autentizaci: mechanismus pro ověření platnosti dat
- Integrita: mechanismus pro detekci manipulaci dat

3.5.1. TLS handshake

Předtím než můžeme komunikaci prohlásit za bezpečnou je potřeba vytvořit tzv. šifrovaný tunel (encrypted tunnel). Nejdříve se server a klient musí shodnout na verzi TLS, algoritmu šifrování a případně ještě ověření certifikátu.

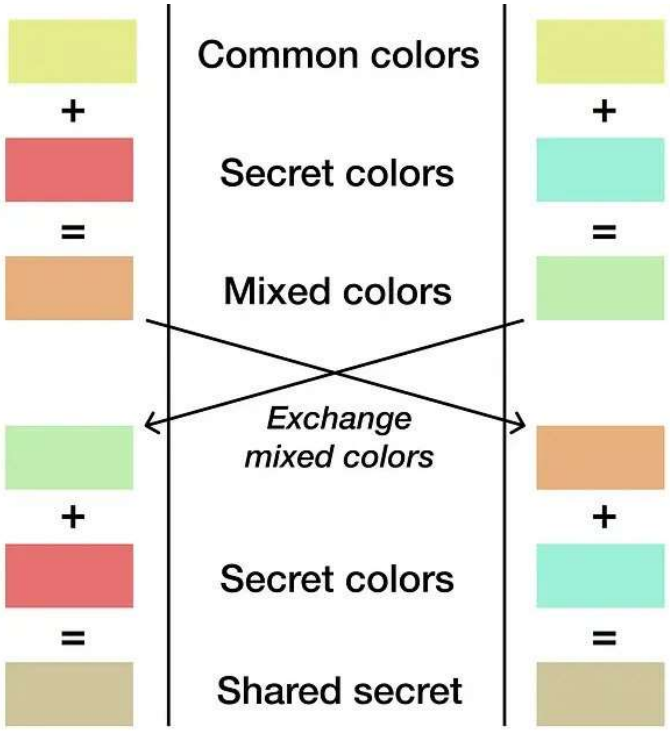
1. Prvním krokem je navázání spolehlivého TCP spojení, to zůstává stejné jak pro zabezpečenou komunikaci, tak nezabezpečenou. Zabezpečení je pouze přidaná vrstva k původnímu HTTP protokolu. Z tohoto důvodu se nejprve provede tří krokový handshake pro navázaní TCP komunikace.
2. Poté klient může serveru poslat informace o TLS připojení – verze TLS protokolu, list podporovaných algoritmů a případně další volitelné parametry.
3. Server obdrží tyto informace: Vybere si svoji verzi protokolu, zvolí si šifrovací algoritmus a k celé komunikaci připojí svůj certifikát. Klientovi následně odešle odpověď.
4. Předpokládáme, že klient i server je schopen najít společné parametry, aby komunikaci zabezpečil. Pokud například server má nastavené jiné verzi protokolu než klient, pak komunikace skončí s chybou. V případě nalezení společných parametrů a klient je spokojen s certifikátem, který mu poskytne server, klient zahájí výměnu klíčů pro vytvoření symetrického klíče pro následující relace.
5. Server zkontroluje parametry pro výměnu klíčů klientem a zkontroluje integritu zprávy. Následně vrátí klientovi zašifrovanou zprávu: Finished
6. Klient dešifruje zprávu, a pokud je všechno v pořádku, je inicializován zabezpečný tunel a aplikace si mohou vyměňovat zabezpečeně data [12].

Aniž bych to přímo zmiňoval, tak z textu lze vydedukovat jednu zásadní funkcionalitu. V případě navázání zabezpečeného tunelu se využívá jak asymetrického, tak symetrického šifrování. Symetrické šifrování má tu výhodu, že je výpočetně jednodušší než pracovat s asymetrickou šifrou. Celý proces funguje tak, že díky veřejnému klíči poskytnutým serverem můžeme jakoukoliv zprávu šifrovat tak, aby ji server dokázal zpátky dešifrovat. Klient si tedy vytváří svůj klíč určený pro symetrické šifrování. Server tento klíč ale nezná, proto ho klient zašifruje veřejným klíčem a sdělí ho serveru. Ten ho dešifruje a následně komunikace může pokračovat v symetrickém šifrování.

Tento princip má ale jednu nevýhodu. Stejný pár veřejného a soukromého klíče, který poskytuje server, se používá jak pro ověření serveru, tak k šifrování symetrického klíče vytvořeným klientem. To znamená, že pokud útočník dokáže odcizit tento privátní klíč, tak může dešifrovat celou komunikaci (v případě, že naslouchá výměně klíčů). Další slabinou je, že útočník může celou komunikaci pouze zaznamenávat a komunikaci dešifrovat až v případě, jakmile se mu podaří privátní klíč získat.

Tomu se dá zabránit tzv. dopředným utajením. Používá se Diffieho-Hellmanova výměna klíčů. Jedná se o protokol, který umožňuje komunikaci mezi stranami, které do té doby o sobě nemají dohodnuty žádné šifrovací klíče. Často se používá analogie s barvami namísto složitých a dlouhých čísel [13].

Předpokládejme dvě strany: klient a server. Obecným předpokladem Diffieho-Hellmanova výměny klíčů je, že obě strany si vyberou svojí společnou veřejnou barvu a zároveň svoji rozdílnou tajnou barvu (ve skutečnosti se jedná o složité prvočíslo). Poté si tyto dvě barvy (svojí sdílenou a soukromou) namíchají a vymění. Následně znovu vyměněné barvy si namíchají se svou tajnou barvou. Tím dosáhnou toho, že jak klient,



Obrázek 2 - Popis Diffieho-Hellmanovy výměny klíčů [13]

tak server zná svoje sdílené tajemství. Vizualizace je na (Obrázek 2).

Diffieho-Hellmanova výměna klíčů probíhá bez explicitního sdělení jedné a nebo druhé straně. Soukromý klíč serveru se používá jenom pro podepsání a ověření handshaku, ale symetrický klíč se mezi serverem a klientem nijak nepřenáší. Díky tomu není možné útočníkem zaznamenat tuto komunikaci a následně sdělení dešifrovat. Kombinované použití klíčů momentálně preferují každé moderní prohlížeče [13].

3.5.2.Man in the middle (MITM) útok

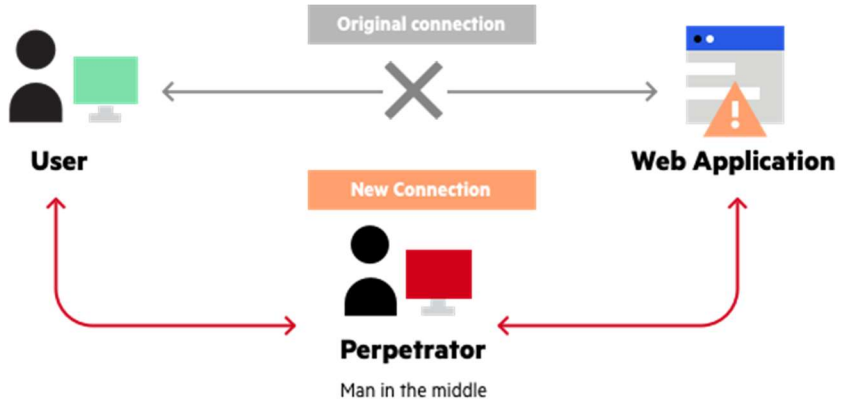
V úvodu kapitoly HTTPS jsem se již zmínil, že samotné šifrování a používání certifikátů, nemusí nutně zaručovat integritu dat. Zde popíšu

možný útok man in the middle, který může vést k úniku dat, přestože na obou stranách používáme zabezpečenou komunikaci.

Představme si běžný scénář navázání spojení klienta se serverem. Klient kontaktuje server, dojde k ověření integrity pomocí soukromého klíče, vytvoření symetrického klíče a navazující komunikace dle předchozích odstavců. Princip útoku man in the middle je jednoduchý v tom, že útočník může vlézt do naší komunikace. V případě klienta se tak klient napřímo nedotazuje serveru, ale útočníka.

Jednotlivé kroky man in the middle attack:

- Klient se dotáže zdánlivého serveru (ve skutečnosti se dotazuje útočníka).
- Útočník naváže zabezpečenou komunikaci s klientem dle běžných standardů.
- Útočník se zároveň dotáže cílového serveru, na který se původně dotazoval klient.
- Útočník klientovi poskytne data se serveru [14].



Obrázek 3 - Útok Man in the Middle [14]

Z tohoto principu je zřejmé, že vzhledem k tomu, že se cílového serveru dotazuje útočník a klientovi pouze zprostředkovává data, tak má informace o obou symetrických klíčích (klíč pro šifrování komunikace mezi klient - útočník a dále útočník – cílový server).

Útočník takto může pak krást a manipulovat s daty, jak je mu příjemné. Vizualizaci lze nalézt na (Obrázek 3)

3.5.2.1. Jak se bránit

Bránit se MITM útoky je na jednu stranu jednoduché a na druhou poměrně složité. Vzhledem k tomu, že integritu dat kontrolujeme pomocí certifikátu poskytnutým druhou stranou, pak není způsob, jak zajistit, že do této komunikace nevleze žádná třetí strana. Útočník může vždycky certifikát podvrhnout a pozměnit tak, abychom si mysleli, že poskytovatelem je skutečná webová aplikace. Tuto kapitolu zde uvádím právě z důvodu, že webová aplikace a klient si při zjišťování důvěry nikdy nepostačí samostatně. V navazujícím textu se budu věnovat vytváření certifikátů, které jsou podepsány certifikační autoritou.

Autorizační autorita je právě tou nutnou třetí stranou, která zajišťuje důvěryhodnost stran. A v případě, že aplikace na straně klienta vyhodnotí, že tato důvěra je poškozena, tak komunikaci buď nenaváže, a nebo alespoň dá uživateli zřejmým způsobem vědět, že zabezpečení nemusí být důvěryhodné.

Certifikační autorita má svůj kořenový certifikát, pomocí kterého podepisuje další certifikáty, které sama vydává. Pomocí veřejných klíčů můžeme pak ověřit, že vydaný certifikát je opravdu ten, který očekáváme.

Vlastnosti CA:

- Zajištění ochrany svých kořenových certifikátů. Pokud by byl privátní klíč odcizen, pak by nikomu nebránilo vytvořit nový certifikát a sám si ho podepsat.
- Zajistit vydání certifikátů pouze důvěryhodným uživatelům. Častým způsobem je vydání certifikátu pro danou doménu. Pak certifikační autorita ověří, že doména patří žadateli certifikátu [15].

Posledním článkem řetězu je odpověď na otázku, jak nám certifikační autorita pomůže v získání důvěryhodnosti. Z principu by nemělo záležet na tom, zda certifikát podepíše certifikační autorita, nebo kdokoliv jiný. A odpovědí je, že prohlížeče a další klienti, kteří kontrolují, zda je certifikát platný, mají v sobě již předinstalovány veřejné klíče těchto certifikačních autorit. Pokud tedy máme zkontrolovat důvěryhodnost certifikátu, tak se

využívá právě těchto veřejných klíčů, které se ale nepřenáší mezi žádnými ze stran.

V případě podmínky, že bychom nechtěli využít žádných certifikačních autorit, tak bychom museli zajistit spolehlivý přenos kořenového certifikátu na klienty.

4. Praktická část – Aplikace

Cílem praktické části je navrhnout aplikační část, která se bude zabývat monitoringem síťových zařízení. Naše aplikace bude umět nejen vyhodnocovat, zda je zařízení online, nebo offline. Ale zároveň budeme moci sledovat jednotlivý vývoj v čase. V této práci jsem již částečně vycházel z připravené funkcionality, která se týká zjišťování stavu zařízení pomocí ICMP paketů. Logika systému, která zůstala v mé práci zachována, je taková, že považujeme zařízení za offline, pokud zařízení v časovém limitu neodpoví na ICMP paket. Jedná se o obecně jednoduše implementovatelnou možnost, jak sledovat online zařízení. Většina z nich (pokud to není explicitně na zařízení zakázané), odpovídá na tyto pakety. Aplikace je rozšířena o mnoho parametrů, aby byla možnost definovat vlastní parametry pro každého hosta jednotlivě.

Obecně na trhu existují již řešení, která se zabývají podobnou tématikou. Není možné v rozsahu diplomové práce konkurovat těmto systémům. Přesto ale tato aplikace přináší pár funkcionalit, které bychom si v podobných systémech museli řešit vlastními skripty nebo doplňky. Jednotlivé funkcionality budou popsány v dalších kapitolách, ale mohu zmínit například funkce jako testování zařízení z více míst a dlouhodobý monitoring odezvy v čase. Aplikace se nezaobírá pouze stavem online – offline, jako většina těchto aplikací.

Zároveň je možné chápat tuto práci jako jakousi úvodní kuchařku pro zájemce, kteří se chtějí porozhlédnout po nových možnostech vývoje aplikace. Budou zde probrány i kapitoly jako nasazení v kontejnerizovaném prostředí a tyto výhody oproti běžně používaném způsobu spouštění aplikace ve virtualizovaném prostředí. Již v úvodu jsem se zmínil o několika základních tématech zabezpečení aplikace, které jsou v dnešní době nepostradatelnou součástí vývoje každé aplikace. A zanedbání těchto principu může v budoucnu vést ke kompromitaci aplikační části. V této a případně další části, která bude navazovat a zabývat se čistě kontejnerizovaném prostředím zmíním, jak nasazovat zabezpečené aplikace. Aplikace bude běžet na vlastní doméně, na kterou bude vázán

certifikát a pomocí certifikační autority bude zajišťovat důvěryhodný přenos dat. Tyto certifikáty je třeba pravidelně obnovovat a vzhledem k jejich nízké době platnosti (mnohdy i méně než půl roku) způsobují IT administrátorům nemálo vrásek na čele. Naše aplikace pomocí dalších služeb bude tyto certifikáty obnovovat s dostatečným předstihem automaticky a v případě problémů budeme informování emailem.

4.1. Princip aplikace

Aplikační část jsem rozdělil do dvou úrovní. O každou úroveň se stará oddělená logická část aplikace, které jsou ale úzce provázané. Tyto části rozděluji na Worker a Datastore. Worker je aplikační část, která se stará o jednotlivé testování síťových zařízení. Budeme-li mít například server, tak Worker se v pravidelných interval dotazuje serveru, zda je dostupný. V předepsaný čas mu pošeleme ICMP paket a čekáme na odpověď. Těchto Workerů může být nasazen nespočet, fungují téměř v autonomním režimu. Mají svou databází, svou logickou část a běží na oddělených hostitelích.

Na druhou stranu Datastore obsahuje centrální správu dat. Byl kladen důraz na to, aby bylo možné aplikaci rozšiřovat na další místa, ale zároveň aby byla lehce spravovatelná. V Datastoru se tak budou centrálně shromažďovat data z Workerů, definovat uživatelé, konfigurace Workerů apod. Jedná se o centrální bod aplikace. Na této části také poběží API server, která umožňuje získávat data z databáze a poskytovat je dále frontendu pro zobrazování uživateli. Budeme-li chtít rozšířit úkol pro všechny Workery, stačí udělat záznam do databáze Datastoru a úkol se automaticky rozšíří na všechny Workery.

Obě části aplikace programuji v programovacím jazyce Python a používám přitom několik dalších knihoven, včetně frameworku pro API – FastApi.

Poznámka: Vzhledem k tomu, že jsem některé základní funkční bloky převzal, tak pokud bude v této diplomové práci kód zmíněn, pak uvedu, že nejsem jeho autorem.

```python
import ping3

from modules.common import ms_time

def get_response_ping(address, timeout_value):
    """Get ping value in millisecons.
    Convert None values to -1.
    """

    result = ping3.ping(address, timeout=timeout_value)

    if isinstance(result, float):
        return ms_time(result)
    else:
        return -1
```

Zdrojový kód 1 - Testování adres

4.2. Worker

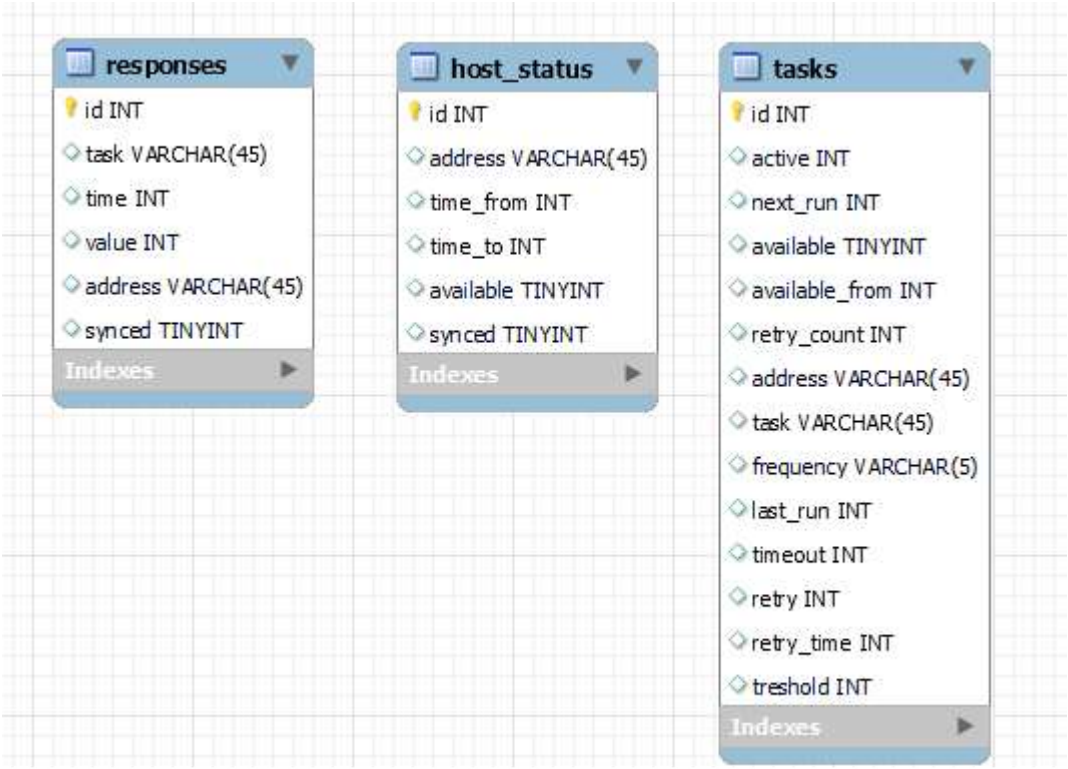
Středobodem Workeru je testovat cílové stanice a zjišťovat jejich odezvu. Základní logiku za tímto řešení jsem převzal z původní aplikace. Jedná se odesílání ICMP paketů, přičemž měříme latenci odpovědi. K tomuto používáme Python knihovnu ping3.

Jedná se o jednoduchou funkci, která má dva vstupní parametry: adresa a timeout_value. Kód si lze prohlédnout na (Zdrojový kód 1). Pokud síťové zařízení neodpoví v předepsaném timeout hodnotě, pak funkce vrátí -1 a považuje se zařízení za nedostupné. Tato funkci byla převzata a pouze jsem ji rozšířil o timeout_value.

Dále bylo potřeba vyřešit dva zásadní problémy:

a) Jak vyvolat testování cílového stroje
b) Jak zpracovávat data

Nejprve se budu věnovat tomu, jak zpracovávat data. S tím nesouvisí pouze ukládat záznamy o latenci stroje v nějakém čase. Potřebujeme znát i cílové stranice určené k testování, jejich frekvenci, případně další volitelné parametry – délka okna pro timeout, název zařízení apod.



Obrázek 4 - Návrh databáze Worker

Databáze se skládá ze tří hlavních tabulek: reponses, host_status a tasks. Grafickou vizualiazaci si můžete prohlédnout na (Obrázek 4).

4.2.1.Tabulka tasks

Tasks obsahuje tyto sloupce:

a) Active – časové razítko pro synchronizaci z DataStoru do Workeru
b) Next_run – jedná se o čas, kdy bude adresa znova otestována, princip vysvětlím dále
c) Available – pokud je cílová stanice nedostupná, uloží se hodnota 0, pokud je cílová stranice dostupná, uloží se 1
d) Available_from – pokud dojde ke změně stavu available, uloží se sem časové razítko změny
e) Retry_count – označuje, kolikrát se má cílová stanice otestovat v případě změny stavu, aby se stanice považovala za nedostupnou / dostupnou.
f) Address – IPv4 adresa cílové stanice
g) Task – typ úkolu, momentálně máme pouze PING, navrženo pro rozšíření
h) Frequency – časová perioda, se kterou se adresa testuje
i) Last_run – časové razítko posledního provedení úkolu

j) Timeout – nastavení časového limitu pro odpověď na ping
k) Retry – aktuální počet zopakování tasku, postupně se zvedá, jak se znovu testuje task. Jakmile dosáhne požadovaného počtu, pak logika změní stav klienta.
l) Retry_time – můžeme nastavit jinou frekvenci pro zopakování tasku v případě, že cílová stranice změní stav available
m) Treshold – timeout nastavuje časové okno, po které se přestane čekat na odpověď. Treshold je hodnota latence, která když bude přesáhnuta, tak ji stanici můžeme považovat za neaktivní.

Podrobné vysvětlení programu bude následovat.

4.2.2.Tabulka host_status

Obsahuje sloupce:

a) Address – adresa, které se týká záznam
b) Time_from – časový údaj: **od** kterého okamžiku byl stanice dostupná / nedostupná
c) Time_to – časový údaj: **do** kterého okamžiku byla stanice dostupná / nedostupná
d) Available – dostupnoust
e) Synced – přiznak, zda řádek byl synchronizován s datastore

Do této tabulky se zapisuje log dostupnosti cílové stanice. Můžeme jednoduše vypsat, kdy byl cílový stroj dostupný a kdy nikoliv.

4.2.3.Tabulka reponses

Obsahuje sloupce:

a) Task – typ tasku
b) Time – čas, kdy byl paket ICMP odeslán
c) Value – hodnota latence
d) Address – adresa, které se týká záznam
e) Synced - přiznak, zda řádek byl synchronizován s datastorem

4.2.4.Chod programu

Program běží v několika základních krocích:

a) Spuštění a inicializace

Při spuštění programu se nejprve dotážeme datastoru a autentizujeme se. Následně si vyžádáme jednotlivé úkoly (dojde k synchronizace tasku). Obdržená data si uložíme do tabulky tasks.

b) Provedeme první otestování cílových adres

Při synchronizaci obdržíme od datastoru data s úkoly. Všechny adresy jednotlivě otestujeme a zapíšeme si jejich dostupnost a podle frekvence vypočítáme, kdy se má adresa znovu otestovat.

c) Opakování úkolů

Po úvodním otestování adres máme již podle frekvence zjištěno, kdy bude třeba adresu znovu otestovat. Periodicky zjišťujeme, který úkol je na řadě a ve správný čas odešleme ICMP požadavek.

d) Synchronizace

Data o odpovědích a dostupnosti cílové stanice synchronizujeme s datastorem.

4.2.5. SQLAlchemy

V aplikaci je potřeba pracovat s daty, která jsou ukládána v databázi. Používám přitom knihovnu SQLAlchemy, která ulehčuje práci a má značné výhody v případě například změny používané databáze. Jedním z požadavků na aplikaci byla jednoduchá databáze – zvolil jsem SQLlite. V budoucnu, pokud by ale byl požadavek na databázi většího rozsahu, bylo by v případě psaní SQL dotazů třeba požadavky do databáze všechny přepisovat. SQLAlchemy nám umožní pouze změnit connector, kterým se připojujeme k databázi a knihovna vše vyřeší za nás.

SQLAlchemy poskytuje ORM (Object-relation mapping). Jedná se o techniku, která vpodstatě zprostředkovává spojení mezi databází a objektově orientovaném programem.

Pomocí nástroje ORM můžeme přistupovat k datům z databáze jako k objektům. Předpokládejme, že chceme získat uživatele z databáze. Pak pomocí čistého SQL requestu bychom museli napsat příkaz podobný

```
"SELECT id, name, email FROM users WHERE id = 1"
```

tomuto:
Při použití ORM nástroje ale stačí napsat pouze toto:
Na první pohled to nevypadá jako velké zjednodušení. Celý princip, ale spočívá v tom, že zpátky dostáváme objekt, pokud chceme například získat

```
session.query(Users).filter(Users.id == 1).first()
```

pouze name, můžeme jednoduše napsat:

```
result = session.query(Users).filter(Users.id == 1).first()
result.name
```

Podobným způsobem můžeme data přidávat, upravovat a mazat. Práce s daty pomocí ORM bude vidět na ukázkách kódu.

4.2.6. Výhody ORM

a) S daty můžeme pracovat jako s objekty. To zjednodušuje práci s daty a zároveň zlehčuje následné porozumění kódu (jednodušší syntax)
b) Flexibilita – Pokud změníme typ databáze, v inicializaci připojení k databázi pouze změníme nastavení připojení. O všechno ostatní (úpravu SQL příkazů, relace) se postará knihovna.
c) Transakce – obecně modely ORM zajišťují konzistentnost dat [16]

Využívání ORM může mít ale i jisté stinné stránky. Hlavní nevýhodou, která se uvádí, je potřebný výkon databáze. Vzhledem k tomu, že často voláme a žádáme data, která vlastně třeba ani nepotřebujeme, tak tím zatěžujeme databázi. Pokud voláme složitý dotaz, tak ORM model může využívat dotazy, které nejsou plně optimalizované. V naší aplikaci se s tímto problémem zpravidla nesetkáme. Máme jednoduché dotazy a naše databáze je velikostí spíše v řádech MB než GB.

4.2.7.Alternativy

V předchozím odstavci jste popsal využití SQLAlchemy, které poskytuje ORM (Object-relation mapping) pro efektivnější práci s databázemi v objektově orientovaném programovacím prostředí. Jednou z alternativ k SQL Alchemy je Djnago ORM, které je integrována do Django webového frameworku. Django ORM nabízí podobně jako SQLAlchemy abstrakci databázových tabulek do tříd, ale je navržena tak, aby byla úzce propojena s dalšími komponentami Django, což usnadňuje vývoj webových aplikací. Django ORM také poskytuje výkonný administrační rozhraní pro správu databáze, což může být velmi užitečné pro rychlý vývoj a testování (Django admin).

SQLAlchemy nabízí větší flexibilitu a je vhodnější pro aplikace, kde je potřeba práce s různými typy databází nebo pokud je potřeba komplexnější práce s transakcemi [17], Django ORM poskytuje lepší integraci v rámci Django frameworku, což může být výhodné pro rychlý vývoj a jednoduchost ve webových aplikacích [17]. Pokud je vaše aplikace postavena na Django, použití Django ORM může být efektivnější z hlediska integrace a konzistence celého projektu.

4.2.8. Návrh databáze v SQLAlchemy

Jednou z mnoha výhod SQLAlchemy je definování struktury databáze přímo v kódu, aniž bychom nejprve ručně generovali koncovou podobu databáze. Chceme, aby když se někdo rozhodne využívat naší aplikaci, aby byla co možná nejjednodušeji nasaditelná. Možností by bylo mít v kódu rovnou předvytvořenou databázovou strukturu a databázi rovnou dodávat s kódem. Tato varianta by byla vcelku jednoduše aplikovatelná na naší databází SQLite, která je v podstatě pouze soubor. V praxi se jde obvykle cestou, kdy dodáme s programem například SQL příkazy, které v databázi vytvoří požadovanou databázovou strukturu. Aby inicializace proběhla hladce je potom při instalaci (a nebo před ní) nutné definovat požadované parametry pro navázání komunikace s databází.

V mojí aplikaci se při prvním spuštění zkontroluje, zda databáze existuje a případně ji vytvoří. V budoucnu, pokud by někdo program rozšiřoval, a požadoval by složitější databáze typu například MySQL, tak by jednoduše jen upravil v konfiguraci connector, který se stará o navázání spojení. Celou ostatní přípravu vyřeší SQLAlchemy.

4.2.9. Ukázka implementace modelu

```
from sqlalchemy import Column, Integer, String
from pydantic import BaseModel

class BaseItem():
    """
    Attributes common for Response and Task.
    """
    id = Column(Integer, primary_key=True)
    address = Column(String(100))
```

Zdrojový kód 2 - implementace ORM modelu

Nejprve si definujeme BaseItem (Zdrojový kód 2), ten není sám o sobě ještě tabulkou v databázi. Pouze vytváříme jednotlivé části tabulky, které se dají recyklovat tak, abychom nemuseli v kódu dělat příliš změn, pokud bychom chtěli něco změnit. Téměř každá tabulka v aplikaci obsahuje unikátní identifikátor a adresu. V datastoru pak využíváme i relačních vazeb, které pak adresu nahradí, aby nebyla obsaženy v každé tabulce. V tomto případě

```
class BaseResponse(BaseItem):
    """
    Outcome of any task/test/measurement.
    """
    __tablename__ = 'responses'

    task = Column(String(100))
    time = Column(Integer)
    value = Column(Integer)
```

Zdrojový kód 3 - BaseResponse

ale slouží BaseItem jako takový základní stavební kámen pro ostatní tabulky. Jakmile máme definovaný BaseItem, můžeme definovat BaseResponse (Zdrojový kód 3). Vidíme, že obsahuje již BaseItem, takže výsledná tabulka se bude skládat ze sloupců: id, address, task, time, value.

```
from sqlalchemy.orm import declarative_base
from sqlalchemy import Column, Integer, Boolean, String

from modules.models import BaseTaskWorker, BaseResponse

Base = declarative_base()

class Response(BaseResponse, Base):
    """
    Outcome of any task/test/measurement.
    """
    synced = Column(Boolean)

    def sync_values(self):
        return {
            "id": self.id,
            "address": self.address,
            "task": self.task,
            "time": self.time,
            "value": self.value
        }



def make_tables(engine):
    """
    Crate tables if they do not exist.
    """
    Base.metadata.create_all(engine, checkfirst=True)
```

Zdrojový kód 4 - ORM - definice tabulky

Poslední částí je dokončit Response (Zdrojový kód 4). Poslední sloupec, který nám schází je synced. Dále definujeme funkci make_tables, která při zavolání vytvoří požadované tabulky, pokud neexistují.

Přichází zde asi dvě zásadní otázky – proč definujeme Response pomocí BaseResponse, přičemž bychom mohli rovnou definovat BaseResponse a ušetřit si jeden zápis. A jak SQLAlchemy rozhoduje o tom, jaké tabulky vytvoří.

Reponse vytváříme pomocí BaseResponse z toho důvodu, že tuto strukturu dědí jak Worker, tak Datastore. Obě tabulky jsou velmi podobné a na tomto případě lze hezky prezentovat vhodnost používání těchto oddělných struktur. Tabulky Reponse ve Workeru obsahuje ještě sloupec

synced. Tam se ukládá příznak, zda byl požadovaný záznam již synchronizován s Datastorem, nebo nikoliv. Celý tento řádek v tabulce Reponses je vzat a odeslán do Datastoru – ale bez jednoho parametru a to synced. Tyto tabulky se tedy liší pouze v tomto jednom sloupci. Pokud bychom se v budoucnosti rozhodli, že chceme přidat další sloupec (nebo nějaký odebrat), stačí tuto úpravu provést v BaseResponse a provede se na obou dvou stranách. Nakonec zbývá otázka podle čeho SQLAlchemy vytváří tabulky a v jaké databází. K tomu slouží dvě základní části. Pokud se podíváte do funkce make_tables, uvidíte, že jedním parametrem je engine. Pomocí něj se definuje připojení k databázi. Tím se rozhodne, zda bude vytvářet databáze pro Datastore, nebo Worker, protože pro každou z těchto variant bude engine rozdílný.

Dále využíváme objekt Base, ten je definován hned na začátku programu:

```
from sqlalchemy.orm import declarative_base
Base = declarative_base()
```

Zdrojový kód 5 - ORM - declarativeBase

Objekt declarative_base (Zdrojový kód 5) je tedy definován přímo v Sqlalchemy knihovně a my ho pouze voláme v požadované části kódu. Base pak slouží jako další parametr v třídě Response (Zdrojový kód 4). Pokud v objektu zavoláme metadata.create_all, SQLAlchemy už ví, k jakým třídám se vztahuje a které tabulky má vytvořit.

Podobným způsobem vytváříme celou strukturu databáze pro Workera i Datastore. Protože se kód víceméně celý opakuje podle daného vzorce, nebudu zde uvádět celý model. Jen ukážu ještě jednu důležitou část:
Pokud se podíváme zpět na třídu Response (Zdrojový kód 4), lze v ní vidět ještě funkci sync_values. Ta nám definuje, jak bude vypadat objekt v případě, že se vrací data z tabulky. Můžeme pak přímo volat v kódu tuto funkci:

```
response_row = session.query(Response).filter(Response.address ==
response["address"]).order_by(Response.id.desc()).first()
lastValue = response_row.sync_values()["value"]
```

Zdrojový kód 6 - ORM - výpis hodnot

Nejprve si najdeme první záznam v tabulce response podle daných parametrů, které jsou pro nás důležité (Zdrojový kód 6). Stejně tak bychom si mohli vyžádat jen první záznam bez jakýchkoliv dalších parametrů. Následně, pokud chceme získat value, tak lze zavolat na objekt reponse_row metodu sync_values (která je definována ve třídě Response) a dále přistoupit k hodnotě value. Předpokladem k fungování je samozřejmě definováni metody sync_values ve třídě Reponse – která zároveň definuje sloupce v tabulce. Zde vidíme, jak si správným návrhem modelu můžeme jednoduše definovat podobu, jak budeme přistupovat k datům v celém programu.

4.2.10. Struktura složek a souborů

Aby nám celé vysvětlování funkcionality kódu dávalo smysl, je potřeba nejprve vysvětlit strukturu složek. Jak jsem psal v předchozí kapitole týkající se databázové struktury, tak využíváme společné struktury jak pro Datastore, tak pro Worker.

Obecná struktura složek:

- Certificate
- Databases
- Logs
- Modules
    - Datastore
        - Main.py
    - Worker
        - Main.py
    - Models.py
- Run.py
- Settings.py
- Requirements.txt

Z hlediska rozdělení je kód strukturován do dvou modulů – DataStore a Worker. Tyto moduly dědí velmi podobné struktury, každý obsahuje hlavní soubor main.py, který spouští hlavní modul. Obsahuje SqlConnector, třídy pro správu dat apod. Konkrétní funkcionality budou dále vysvětleny.

Momentálně je potřeba poznamenat, že veškeré sdílené struktury jsou vytaženy mimo tyto moduly. Hlavními konfiguračními soubory je settings.py, kde nastavujeme hlavní parametry pro správné fungování obou částí. Requirements obsahují potřebné knihovny, které je potřeba nainstalovat.
Run.py je potom unifikovaný aplikační kód, který spouští oba moduly zároveň. Je navržen tak, aby co možná s nejméně starostmi bylo možné rozeběhnout obě části aplikace a dále nebylo třeba provádět žádné další změny – například generovat autorizační token workera, jednotlivě spouštět moduly apod.

Settings.py

```
"""
Hardcoded settings for the project
"""
import sys
import pathlib
import copy
import os
from modules.common import build_url

## User specified part
TESTING = False
WORKER_API = "9d207bf0-10f5-4d8f-a479-22ff5aeff8d1"
DATABASE_FOLDER = "databases"
LOG_FOLDER = "logs"
DATASTORE_DATABASE_FILE = "datastore.db"
DATASTORE_LOG_NAME = "datastore.log"
DATASTORE_APP_ADDRESS = ("127.0.0.1", 8000)
WORKER_DATABASE_FILE = "worker.db"
WORKER_LOG_NAME = 'worker.log'
## Testing override
if 'unittest' in sys.modules.keys():
    TESTING = True

    DATASTORE_DATABASE_FILE = "datastore_test.db"
    DATASTORE_LOG_NAME = "datastore_test.log"
    DATASTORE_APP_ADDRESS = ("127.0.0.1", 5000)

    WORKER_DATABASE_FILE = "worker_test.db"
    WORKER_LOG_NAME = 'worker_test.log'
## Calculated part

for folder_path in [DATABASE_FOLDER, LOG_FOLDER]:
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)
DATASTORE_APP_URL = build_url(*DATASTORE_APP_ADDRESS)
```

```
DATASTORE_DATABASE = pathlib.PurePath(DATABASE_FOLDER,
DATASTORE_DATABASE_FILE)
WORKER_DATABASE = pathlib.PurePath(DATABASE_FOLDER, WORKER_DATABASE_FILE)
```

Zdrojový kód 7 - Nastavení

Zde si můžeme povšimnout, že je snaha všechny programové proměnné definovat v této části. Mezi základní proměnné lze definovat IP adresu, kde běží API datastoru, dále Worker-API token, který je pak nutný pro správnou autentizaci Workera. V Datastoru musí být tento API token uložen. Jinak Worker nebude správně autentizován. Dále definujeme názvy databází, jejich umístění atd. Kód si lze prohlédnout na Zdrojový kód 7.

4.2.11. Spuštění a inicializace

Spuštění a inicializace Workera, jak jsem již naznačil, spočívá v synchronizaci s Datastorem. Vyměníme si nejprve data ohledně potřebných tasku, provedeme první otestování cílových stanic a stavy uložíme do databáze. V kapitole jsem nastínil databázovou strukturu. Jedním z inicializačních kroků je také zkontrolování databáze, zda existuje, a případně její vytvoření.

Celý inicializační proces začíná při spuštění souboru run.py, který spouští jak Datastore, tak Workera. V této části se budu věnovat Workeru. V souboru run.py se pouze spouští proces s Workerem, který je definován v /modules/worker/main.py - Zdrojový kód 8.

```
from multiprocessing import Process
from modules.worker.main import Worker
from settings import DATASTORE_APP_ADDRESS
from subprocess import Popen

def run_worker_app_process():
    """
    start the worker application as a process
    """
    worker_app_proc = Process(target=Worker)
```

```
    worker_app_proc.start()
    return worker_app_proc
```

Zdrojový kód 8 - spuštění procesu Workera

V souboru main.py je potom celý funkční blok, který obstarává hladký chod Workera.

Chronologický postup inicializace je následovný:

Nejprve se naváže spojení s Datastorem, který se následně udržuje a používá se pro kontinuální synchronizaci mezi Workerem a Datastorem. Pak se vytvoří spojení s lokální databází Workera a hned záhy následuje resetování všech tasku. Tento krok je vytvořen z důvodu návaznosti. Předpokládejme, že nejde již o první spuštění, kde Worker ještě nemá žádná data získaná z Datastoru. Pak databáze Workera obsahuje již tasky, které má periodicky provádět. Naše programová logika je navržena tak, že máme v databázi vytvořený sloupec s frekvencí, jak často má být cílová stanice testována. Po prvním testování si zapíšu čas dalšího otestování a čekám než dojde k aktivaci tohoto pravidla: čas příštího otestování <= aktuální čas. Pro zajištění kontinuálního a testování chci zajistit, že když dojde k restartování aplikace, budu mít aktuální hodnoty, v jakém je cílová stanice stavu (online / offline).

Kód by správně fungoval i v případě nenadálého výpadku systému, já ale požaduji, aby při restartu programu systém zjistil, v jakém stavu se cílové stanice aktuálně nacházejí, i kdyby ještě nenadešel čas na následující měření. Proto při inicializaci provádím resetování tasku (value: next_run = 0). To zajistí, že budou všechny stanice otestovány ihned po restartu.

Hned po tomto kroku následuje inicializace HostStatus, to je záležitost, která se týká zapisování stavů cílových stanic – vpodstatě log výpadků. Jedná se o tabulku, kde se zapisuje, v jakém časovém okně byla stanice online / offline. Abychom zajistili konzistentnost dat po restartu, tak je volána tato funkce. Ta je z důvodu složitosti přesunuta do samostatné kapitoly.

Dále se ve Workeru spustí nekonečná smyčka se tří sekundovým intervalem. Každé 3 sekundy se synchronizuje Worker s Datastorem pomocí metody presync. Tato metoda má hned několik úkolů, ale souhrně je jejím úkolem udržovat stav lokální databáze Workera aktuální s databází Datastoru. Toho dosáhne tímto postupem:

a) Nové tasky, které jsou v datastoru, ale schází v databázi Workera, jsou přidány
b) Stejné tasky jsou ponechány a pouze jsou aktualizovány okolní data (frekvence a další)
c) Ostatní tasky jsou z lokální databáze odstraněny

Těchto tří bodů je dosaženo tímto postupem:

- Od API Datastoru obdržíme objekt s příchozími daty, ten obsahuje všechna potřebná data, které budeme porovnávat a synchronizovat s Workerem.
- Tasks:
    - Každý jednotlivý task se snažíme najít v lokální databází podle dvou příznaků: adresa a typ tasku (kombinace těchto dvou údajů je v tabulce vždy unikátní)
    - Pokud se ho podaří nalézt, pak upravíme potřebné údaje: frekvence, retry, retry_time apod. Vpodstatě sjednotíme příchozí objekt task s lokální databází
    - Pokud v lokální db neexistuje, pak ho vytvoříme.
    - Smažeme ostatní, viz podrobný popis níže.

Smazání ostatních tasku, které nesplňují předchozí požadavky, je úkol s mnoha podobami řešení. Jedním z řešení by bylo při synchronizaci smazat všechny tasky a přidat nové. Problémem je, že potřebuje uchovat data, která se týkají času příštího testování. Pak bychom se mohli vydat tím způsobem, že bychom si vytvořili speciální tabulku s těmito údaji a propojili ji s tabulkou tasks. To by nám umožnilo aplikovat předchozí postup. Já jsem se vydal cestou vedlejšího sloupce v tabulce, kde se nepřímo objevuje příznak aktualizovaných sloupců. Pokud je tento task obsažen v příchozích datech od Datastoru, pak sloupec aktualizujeme s tímto datem. Při každé

aktualizaci dat se obnoví u požadovaného sloupce **active** časové razítko, kdy byl tento záznam naposledy změněn. Všechny záznamy, které nebudou aktualizovány (nebyly obsaženy v příchozích datech), zůstanou se starým časovým razítkem. Ty následně smažu dle pravidla: Task.active < aktuální čas synchronizace.

Úkol synchronizace ale není pouze jednosměrná synchronizace s Datastorem, ale i posílání jednotlivých Responses (log časových latencí cílové stanice) do Datastoru – protože ten by měl sloužit jako centrální úložiště pro všechny Workery a API potom bere data pouze z něj.

Funkce synchronizace je pak propojena, aby obstarávala obě dvě tyto části najednou. Jednotlivé kroky nekonečné smyčky (Zdrojový kód 9), kterou jsem nastínil výše:

1) Získat všechny Reponses z lokální databáze, které mají přívlastek, že ještě nebyly synchronizovány
2) Získat všechny HostStatus, které mají přívlastek, že ještě nebyly synchronizovány. Tato část se týká funkcionality, která loguje dostupnost služeb v čase a bude vysvětlena dále. Synchronizační proces je však implementován společně.
3) Vytvořit payload, který se bude odesílat Datastoru, obsahuje zmíněné dvě části (Responses, HostStatus), Worker API klíč a název Workera.
4) Odeslání payloadu do Datastoru.
5) Endpoint na synchronizaci dat z Workera automaticky vrací tasky, které se mají aktualizovat z Datastoru do Workera. Pokud jsou obsažena nějaká data, pak se zavolá metoda postsync, která byla nastíněna výše
6) Čeká se 3 sekundy a proces se opakuje.

```
    def __io_loop(self):
        """Filling up the storing queue with the data."""
        while True:
            responses = self.sql_conn.presync()
            host_availability = self.sql_conn.presyncHosts()
            payload = {
                "worker": self.name,
                "responses": responses,
                "hosts_availability": host_availability,
                "api": WORKER_API,
            }
            try:
                requests.packages.urllib3.disable_warnings() #later it needs
to be removed
                                tasks = requests.post(self.datastore_url,
json=payload,verify=False).json() #Attention: verify:False
                if tasks is not None:
                  self.sql_conn.postsync(tasks, responses, host_availability)
            except requests.exceptions.RequestException as e:
                            logging.critical("Cannot contact datastore -
{}".format(str(e)))
                self.sql_conn.postsync([], [], [])
            time.sleep(3)
```

Zdrojový kód 9 - loop pro testování adres

4.2.12. Dostupnost služeb – Log

V předcházejících kapitolách jsem se několikrát zmínil o funkcionalitě, která loguje v čase, kdy bylo cílové zařízení dostupné a kdy nikoliv. V běžném režimu aplikace funguje na principu, že pouze testuje dostupnost služby a do databáze se zapisují jednotlivé časy s odezvou systému. Z těchto dat by šlo kdykoliv zpětně zjistit, zda byla služba dostupná a v jakém časovém intervalu. Ve svém programu jsem se ale rozhodl jít cestou druhé tabulky, která bude dostupnost evidovat. Je to z toho důvodu, že při hledání jednotlivých intervalů by byl algoritmus poměrně složitý (přesto ho zkusím nastínit) a také by docházelo ke zbytečnému zátěži hardwaru.

Jak může vypadat tabulka z dostupností:

Datum	IP	Odezva
28.10.2022 8:51:10	8.8.8.8	15
28.10.2022 8:51:20	8.8.8.8	20
28.10.2022 8:51:30	8.8.8.8	-1
28.10.2022 8:51:40	8.8.8.8	-1
28.10.2022 8:51:50	8.8.8.8	20
28.10.2022 8:52:00	8.8.8.8	11

Tabulka 1 - ukázka tabulky Response

Pro zjednodušení jsem se rozhodl mít pouze jednu IP adresu a vynechal jsem pro tento případ další sloupce jako timeout a podobně.
Výsledek, který chceme získat by mohl vypadat následovně:

DateFrom	DateTo	active	IP
28.10.2022 8:51:10	28.10.2022 8:51:20	1	8.8.8.8
28.10.2022 8:51:30	28.10.2022 8:51:40	0	8.8.8.8
28.10.2022 8:51:50	28.10.2022 8:52:00	1	8.8.8.8

Tabulka 2 - ukázka tabulky HostStatus

Snažíme se z původní tabulky vyextrahovat data, v jakém intervalu byl host aktivní a kdy naopak služba byla nedostupná. Tuto tabulku sám vytvářím už při běhu programu, ale nyní můžeme předpokládat, že tento výstup z původní tabulky Responses chceme získat vždy při požadavky na výstup. Chtěli bychom získat data znázorněná v tabulce HostStatus z tabulky Responses.

Algoritmus by mohl být navržen následujícím postupem:

a) Iteračně podle řádků:
   1) Výsledky seřadím podle času sestupně
   2) Zjistím, zda byla služba aktivní a čas DateTo (protože je tabulka seřazena sestupně, mám časový údaj, do kterého byla služba aktivní).

Aktivní je služba dle navrženého programového kódu, pokud se odezva nerovná -1

3) Postupuji na další řádek. Pokud se stav active nezměnil, pokračuji dále. Pokud se změní active, tak zjistím čase DateFrom a mám první řádek tabulky HostStatus.
4) Takto proces opakuji, dokud nedojdu na poslední záznam

b) Využitím SQL dotazů na další řádek, který obsahuje změnu active. Opět tabulku Responses seřadíme sestupně podle času. Pak najdeme první záznam:
   1) První záznam obsahuje čas DateTo stejně jako v předchozím případě. Pak podle odezvy vyhodnotíme stav služby.
   2) Vytvoříme následující SQL dotaz podle těchto parametru: čas předchází datu DateTo, odezva je -1 v případě, že v prvním kroku se odezva nerovná -1. A naopak: Pokud odezva v kroku jedna je -1, pak v sestavovaném SQL dotazu hledáme odezvu takovou, že se odezva rovná -1.
   3) Tímto zajistíme, že nalezený řádek má ve výsledku jiný status active (vyhodnocujeme pomocí odezvy).

Výhoda použití metody b) je, že zpravidla u cílových stanic nebude docházet k tzv. Flappingu – kdy se hostu z velkou frekvenci mění stavy z aktivního na neaktivní a obráceně. Takže místo toho, abychom procházeli mnoho řádků, které pro nás nemají valný smysl, tak si rovnou najdeme ten potřebný. Algoritmus takto nevypadá příliš složitě, ale musíme si uvědomit, že cílových stanic bude zpravidla velké množství a tento postup bychom museli aplikovat tolikrát, kolik máme hostů. A to při každém požadavku klienta na výpis těchto hodnot. Pokud si uživatel bude chtít zobrazit například log 10 hostů za posledních 7 dní, znamená to projet tímto algoritmem vstupní data a to při každém takovém požadavku. Z těchto důvodů jsem se rozhodl, že budu tabulku sestavovat postupně při změně stavů cílové stanice. Později pak můžu pouze brát tato data z již předpřipravené tabulky.

Nevýhodou totoho systému je fakt, že v databázi mám duplicitní data. Kdykoliv bych byl schopný zpětně z tabulky Responses novou tabulku HostStatus sestavit, ale na úkor výpočetního výkonu. Přitom očekávám, že velikost této tabulky bude značně menší než tabulka Responses.

Princip tvorby tabulky HostStatus

Tabulku HostStatus jsem nazval podle toho, že vyjadřuje průběžný status cílové stanice (hosta) v čase. Jak jsem již nastínil v předchozí kapitola tak slouží hlavně pro to, abychom si kdykoliv a jednoduše mohli filtrovat data podle cílové stanice a času.
Pro uživatele aplikace bude důležité sledovat, kdy a na jak dlouho byla služba neaktivní. Tato data pak lze pomocí jednoduchých SQL dotazů získávat přes API a uživateli zobrazovat.

Podstatná část logiky se odehrává v metodě add_response, která přidává do tabulky Reponses jednotlivé záznamy s informací o odezvě. Druhým úkolem ale je zároveň už v tomto kroku zajistit průběžného zapisování do tabulky HostStatus.

Celou metodu přikládám v této kapitole. Nejprve podle IP adresy si zjistím veškeré informace o tasku – hlavně posledně známý stav o dostupnosti adresy a retry_count, který nám nese informaci o tom, kolikrát máme IP adresu znovu otestovat pro změnu stavu active.
Abych připomněl, jak se vyhodnocuje stav active: Hlavní tabulka Tasks obsahuje informace o cílových stanicích – IP adresa, frekvence testování, stav aktivní / neaktivní, a podobně. V předepsaných intervalech se cílová adresa testuje a získává odezva, jakmile dojde ke změně stavu (například host na chvíli přestane odpovídat), tak máme ještě připravené parametry retry_count. Tento parametr označuje, kolikrát máme test na odezvu opakovat než začneme považovat cílovou stanici za neaktivní (nebo znovu za aktivní v případě obrácené situace). K tomu evidujeme ještě jeden parametr – retry_time. Tento čas nám označuje, za jak dlouho máme test na odezvu zopakovat. Obecně se může lišit od frekvence testování. Například za běžných podmínek adresu testujeme jednou za minutu, jakmile se změní

hodnota odezvu na -1 (to znamená, že odezva nebyla zjištěna v časovém intervalu timeout), pak se test zopakuje například 3x v intervalu 5s. Jakmile všechny pokusy jsou negativní, tak až poté považujeme, že se stav active u hosta změnil.

Tímto procesem předcházíme nenadálým ztrátám paketům a chvilkovým výpadkům, které pro nás nemají při tomto vyhodnocování značný smysl. Tyto poruchy můžeme diagnostikovat správnou filtrací z tabulky Reponses.

Přidávání do tabulky HostStatus funguje tímto procesem:

Zjistím si poslední řádek z tabulky Responses náležící k dané IP adrese. Pokud tento řádek naleznu (pro správné fungování musí být v tabulce alespoň jeden záznam), pak zkontroluji, že se změnil stav available, který mám uložený v tabulce Tasks – označuje, zda je poslední stav cílové stanice aktivní / neaktivní. Pomocí hodnoty odezvy získané z tabulky Responses mohu zjistit, zda se tento stav změnil oproti hodnotě available v tabulce Tasks (Zdrojový kód 10).

Pokud se hodnota skutečně změnila, pak přistupuje následující logika opakování testu za předem definovaných intervalech – retry_count, retry, a retry_time (oba sloupce obsahuje tabulka Tasks).

- **Retry**– kolikrát se má adresa otestovat, aby se hostovi změnil aktivity stav
- **Retry_count** – aktuální counter, při testování ho postupně zvětšuji až se dosáhne požadované hodnoty retry_count a změním stav
- **Retry_time** – časový interval v jakém se mají provádět opakované testy

V metodě ověřuji, zda hodnota retry_count dosáhla úrovně retry. Pokud nedosáhla, pak pouze zvětším v tabulce tasks hodnotu retry_count. V případě, že retry_count se rovná a nebo je dokonce větší než hodnota retry, pak provedu vynulování tohoto počítadla a zapíšu informaci o změně stavu cílové stanice do tabulky Tasks a HostStatus.

```
def add_response(self, response):
    """
    write response of tested address to database
```

```
    """
    return_value = False
    result = Response(
        address=response["address"],
        time=response["time"],
        value=response["value"],
        task=response["task"],
        synced=False,
    )

    with self.sessions.begin() as session:
#Najdu task row
              task_row   =   session.query(Task).filter(Task.address   ==
response["address"]).first()
#najdu posledne zapsany response
         response_row = session.query(Response).filter(Response.address
== response["address"]).order_by(Response.id.desc()).first()

        if response_row:
            lastValue =  response_row.sync_values()["value"]
            newValue = response["value"]

# Pokud se od posledniho pingu zmenila odezva, pak zapisu do DB
            if (response["available"] != task_row.available):
                logging.debug("Task availability se neshoduje s namerenym
a DB")
                 if response["retry_count"] >= response["retry"]:
                      logging.debug("retry_count byl prekrocen, zapisuji
zmenu do DB")
                    logging.debug("Adresa: " + str(response["address"])
+  "  retryCount:  "  +  str(response["retry_count"])  +  "  retryDB:  "  +
str(response["retry"]))
                    result_host = HostStatus(
                        address=response["address"],
                        time_from=task_row.available_from,
                        time_to=response["time"],
                        available=task_row.available,
```

```
                    )
                    task_row.available_from = response["time"]
                    session.add(result_host)
                    task_row.retry_count = 0
                    task_row.available = response["available"]
                else:
                    logging.debug("Address: " + response["address"] + "
zvusuji retryCount na " + str(task_row.retry_count + 1))
                    task_row.retry_count = task_row.retry_count + 1
                    return_value = True

        session.add(result)
    #return value znamena, ze je adresa v cyklu return
    return return_value
```

Zdrojový kód 10 - přidání Response do databáze

Problémy

Tento systém přináší jisté nevýhody. Princip vyhodnocování a zápisu do tabulky HostStatus ve své podstatě kopíruje navržený algoritmus pro tvorbu tabulky HostStatus z Reponses zmíněný v kapitole výše. Pouze namísto procházení tabulky si ukládáme metadata (retry_count, available, available_from) do tabulky tasks. A pak z ní data čerpáme a postupně tabulku aktualizujeme. Vidíme, že v případě rozšiřování zmíněného algoritmu výše o funkcionalitu opakování testů a až následné změně stavu cílového zařízení by požadavky na výpočetní výkon ještě vzrostly.

Mysleme ale na případ, že program spadne a nebo bude uživatelem ukončen. Pak máme záznam o posledním provedeném testu odezvy, ale nemáme ještě žádnou informaci o tom, kdy se ukončil systém. Do tabulky HostStatus se zapisuje až při změně stavu cílového zařízení. Pokud bychom program ukončili korektně, pak bychom mohli připravit metody na to, aby se do této tabulky záznamy vytvořily, ale stále nám to neřeší problém s nenadálým ukončením programu.

V tomto případě bychom ztratili informaci o tom, že cílová stanice byla do té doby například aktivní. Předpokládejme, že zařízení bylo 2 dny v kuse aktivní, postupně jsme zapisovali do tabulky Responses v předepsaných intervalech odezvy systému. Pak se systém vypne a za 4 hodiny se opět zapne. Stav zařízení se v tomto čase mohl kdykoliv změnit (a třeba klidně i znovu vrátit zpátky do původní hodnoty). Těmto oknům se chceme vyvarovat. Proto vždy při startu programu volám metodu initTasks. Která se postará o správné vyhodnocení tabulky Responses a chybějící záznam HostStatus doplní.

Princip je poměrně jednoduchý. Nejprve si naleznu z lokální databáze všechny tasky, které mám k dispozici. Postupně přes ně Iteruji a přiřazuji si ke každému z nich poslední řádek z tabulky Reponses.

Pokud existuje novější testování adresy než bylo posledně vyhodnoceno do tabulky HostStatus, tak vytvořím nový řádek s informacemi o dostupnosti služby. Tato podmínka doplňuje jednu podstatnou drobnost. Teoreticky existuje možnost, že dojde k vyhodnocení adresy na odezvu a zároveň s tím budou splněny všechny potřebné podmínky na změnu stavu cílové stanice. (Změna bude zapsána do tabulky HostStatus.) Zrovna v tomto okamžiku bude program ukončen, takže nebude po znovu spuštění programu dostupný žádný novější záznam v tabulce Reponses a tím nebude nutné vytvářet záznam do tabulky HostStatus (Zdrojový kód 11).

```
def initTasks(self):
        logging.debug("init Tasks - generate HostStatus")
        with self.sessions.begin() as session:
            tasks = session.query(Task).all()

            for task in tasks:
                logging.debug("Process task address" + task.address)
                                                    hostStatus      =
session.query(HostStatus).filter(HostStatus.address                 ==
task.address).order_by(HostStatus.id.desc()).first()
                                                lastResponse      =
session.query(Response).filter(Response.address                     ==
task.address).order_by(Response.id.desc()).first()
```

```
                        if not hostStatus or lastResponse.time >
task.available_from:
                    result_host = HostStatus(
                            address=task.address,
                            time_from=task.available_from,
                            time_to=lastResponse.time,
                            available=task.available,
                        )
                     task_row = session.query(Task).filter(Task.address
== task.address).first()
                    session.add(result_host)
                         logging.debug("Add HostStatus time_from: " +
str(task.available_from) + " time_to " + str(lastResponse.time) + "
available " + str(task.available))
                    task.available_from = ms_time()
```

Zdrojový kód 11 - inicializace Tasku

Tímto byly shrnuty a vysvětleny zásadní funkcionality Workera.

4.3. DataStore

Worker se stará o těžkou práci získávání dat s informace o dostupnosti jednotlivých cílových stanic. Ale uživatel aplikace přímo nekomunikuje s těmito podružnými systémy. Uživatel má k dispozici své grafické prostředí (UI – user interface), které komunikuje prostřednictvím HTTP API s datastorem. Datastore, jak už bylo naznačeno, je centrální úložiště dat, kde se koncentrují veškeré informace z jednotlivých Workerů. Při dotazu na konkrétní IP adresu pak není potřeba se jednotlivě dotazovat Workerů na data a dávat je dohromady, ale pouze jednoho místa – datastoru.

Nic ale nevylučuje možnost, aby obě části běžely na stejném zařízení v případě, že bude požadavek na to, aby se Worker od Datastoru hardwarově neoddělovaly. Přesto ale poběží oba systémy nezávisle a k přesunu dat bude docházet (přestože jenom lokálně). V nasazení se očekává, že bude pracovat více Workerů.

Ve struktuře aplikace jsem zmínil, že kód je rozdělen do tří částí – dva moduly (Worker, Datastore) a centrální část pro nastavení. Nasazení aplikace se budu věnovat v další kapitole. V této kapitole se budu zabývat funkcionalitou Datastoru jako nezávislým modulem, který zprostředkovává autentizaci uživatelů i Workerů, API přístup k datům, úložiště pro Workery, ale zároveň i jako centrální bod nastavení pro všechny ostatní subsystémy.

Pokud se uživatel rozhodne, že chce testovat další cílovou stanici (například server), pak ho stačí definovat v Datastoru a pomocí automatické propagace dat se nastavení roznese do všech Workerů, které mají úkol na starosti. Systém umožňuje i takové nastavení jako přiřazování jednotlivých úkolů jednotlivým Workerům, takže ne každý úkol musí zpracovávat všechny Workery.

4.3.1. FastAPI

Z naznačeného konceptu je zřejmé, že je nutné komunikovat s uživatelským prostředím. Toto grafické prostředí vyvíjel kolega Martin Cé v jiné diplomové práci [18] (Identifikátor: KOS-1241007080505).

Pro výměnu dat jsme se rozhodli využít knihovnu FastAPI, která je dostupná pro programovací jazyk Python. Veškerá komunikace probíhá přes HTTP protokol (přesněji přes HTTPS, protože je zpravidla zabezpečená) a data si vyměňujeme skrze objekty JSON.

Stručně je FastAPI framework, který byl navržen jako výkonný nástroj při budování vlastního API. Mělo by zaručovat vysokou rychlost a hlavně i bezpečnost. Využívá typovou kontrolu, kterou si později ukážeme v kódu, takže můžeme rovnou přijatá data kontrolovat, zda se shodují s očekávaným formátem. Jednou z nevýhod je, že se používá formát Pydantic a SQLAlchemy zase používá typový objekt ORM. Tyto modely nejsou vzájemně kompatibilní, takže přestože zpravidla pracujeme s formálně stejným modelem, tak musíme v programu pracovat s dvěma modely.

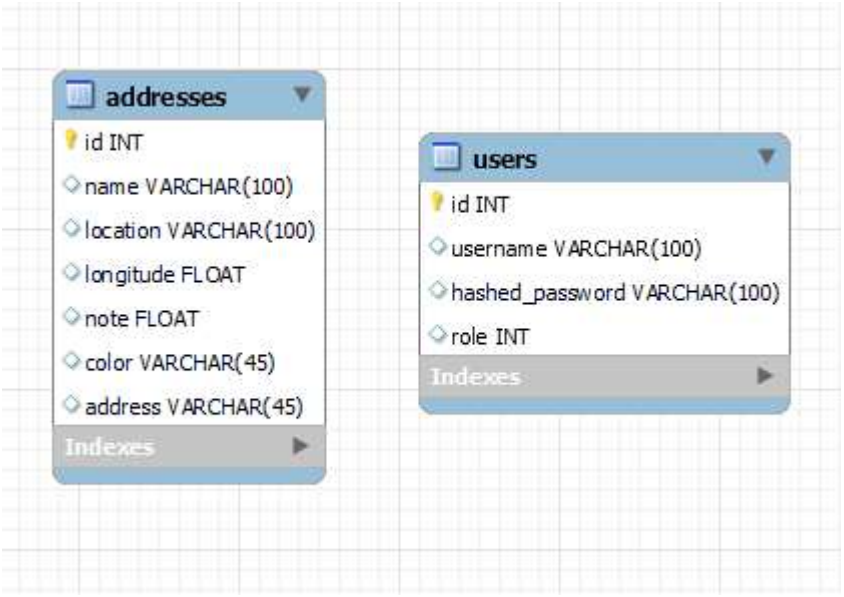
4.3.2. Bezpečnost

Otázka bezpečnosti se díky FastAPI poměrně zjednodušuje. Má již předpřipravené modely a postupy, jak řešit autentizaci uživatelů. Využívá k tomu i otevřené protokoly jako OAuth2, který řeší omezení přístupů pro

jednotlivé kategorie uživatelů, aniž by nutně muselo dojít ke sdílení přihlašovacích údajů. I když obvykle se token přesto získá autentizací přes uživatelské jméno a heslo, v pozdějších dotazech se už využívá pouze autorizační token. Podrobněji budu kapitolu autentizace z praktického hlediska rozebírat později, teoreticky byla část zabezpečení zpracována v první části diplomové práce.

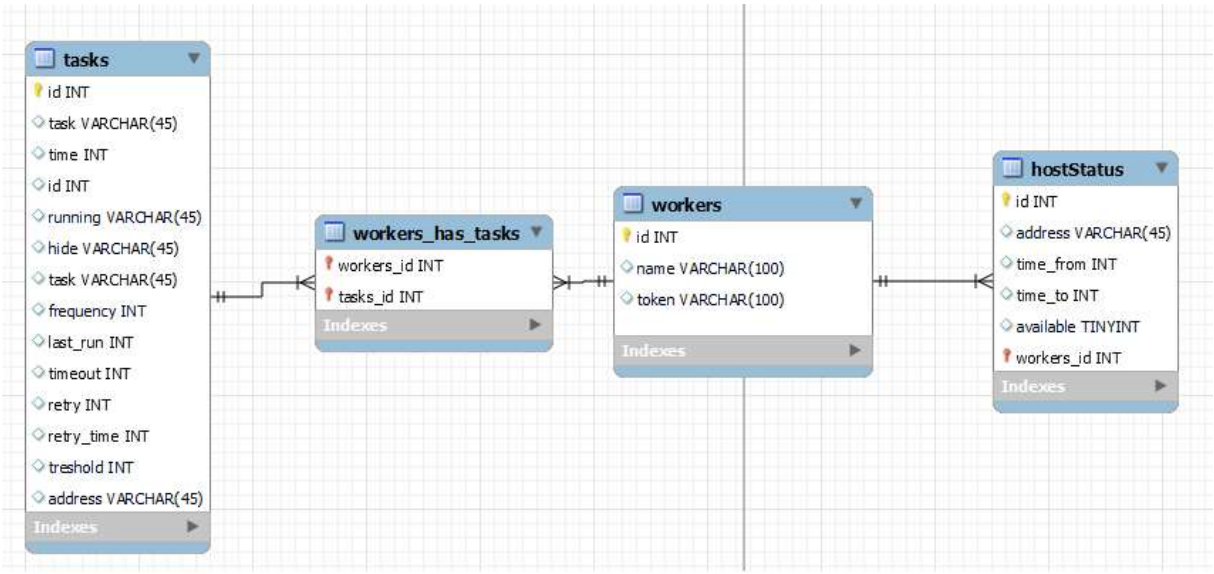
4.3.3.Databáze

Databáze je středobodem práce s daty. Podstatná část nastavení pro Workery se uchovává právě v ní. Zároveň z této databáze získává FastAPI data pro koncové uživatele. Názorné schéma jsem rozdělil do dvou části. První část obsahuje tabulky users a addresses (Obrázek 5). Obě jsou potřebné pro koncového uživatele aplikace. Druhá část s tabulkami tasks, workers a hostStatus je potřebná pro správný chod aplikace (Obrázek 6). Jinými slovy, kdybychom vynechali první část, pak by jádro aplikace fungovalo nadále, ale ztížil by se přístup k datům.



Obrázek 5 - Návrh databáze DataStore, uživatelská část

Tabulka users obsahuje hlavně username, hashed_password a role. Všechny tři sloupce jsou potřebné pro autentizaci uživatelů. Tabulka addresses obsahuje data pro grafické uživatelské rozhraní. Každá adresa má nějaké své geolokační údaje, název, poznámku a barvu, kterou se zobrazuje v grafu. Podrobnosti nejsem schopen poskytnout, moje aplikace pouze zprostředkovává data grafickému rozhraní a jeho zpracování měl na starosti



Obrázek 6 - Návrh databáze DataStore - aplikační část

kolega Cé.

Druhá část databáze je zajímavější. Obsahuje hlavní tabulku tasks, která má z velké části již zmíněné sloupce identické s Workerem. Podobnost je z toho důvodu, že se z datastoru propagují data do lokální databáze Workera. Druhou významnou tabulkou je workers. Obsahuje záznam všech Workerů, které mají přístup k datastoru, pomocí tokenu se autorizují a pak mají přístup ke svým datům. Spojovací tabulkou je workers_has_tasks, která zaručuje vazbu n:m mezi tabulkami tasks a workers. Systém je navržen tak, že můžeme vytvořit mnoho tasku (dokonce ani IP adresa ve sloupci address nemusí být unikátní). Je to z toho důvodu, že můžeme chtít u jednoho Workera testovat zařízení jednou za 5s a na druhém Workeru například jednou za minutu. Nebo můžeme nastavovat rozdílné hodnoty timeout. Celý systém je připraven tak, aby se mohly nastavovat jednotlivé parametry podle libosti – jeden Worker může pracovat v lokální síti a

testovat dostupnost server on-site. Tam budou zřejmě kladeny vyšší nároky na dostupnost než pro Workera, který bude off-site.

Díky této spojovací tabulce jsme schopni funkcionalitu bezpečně zprostředkovat, aniž bychom se museli bát, že některé parametry nebudeme schopni nastavit.

Poslední důležitou tabulkou je HostStatus, který jsem zmínil a popsal u Workera. Zde není žádná zásadní změna, až na jeden parametr a to je workers_id. Kdy se každý hostStatus přímo váže k danému Workeru.

K samotné struktuře ještě dodám, že program je strukturován do tzv. route. Jednotlivé endpointy, které volám jsem navrhl podle tzv. best practices pro API. Tuto konvenci se snažím dodržovat. Existuje 5 typů HTTP metod, které voláme:

a) GET – používá se k získání dat
b) POST – používá se pro vkládání dat
c) PUT / PATCH – používá se pro aktualizaci dat
d) DELETE – používá se pro odstranění dat

Tvorba URL adresy

URL adresy přímo specifikují požadavek, který posíláme na API. Například: (GET) http://localhost/users/1 vrací uživatele s ID = 1. Jak navrhneme schéma URL adresy je libovolné (až na některé pevně dané struktury), ale klidně bychom si mohli vytvořit adresu http://localhost/users/get/1, která by vracela totéž. Abychom při vývoji udrželi daná pravidla napříč celým systémem, je vhodné dodržovat určitá pravidla, která zajistí, že napříč programem bude při tvorbě URL adres udržena soudržnost. To, co následuje za „localhost“ se ve FastAPI definuje jakou routa. Díky ní můžeme strukturovat HTTP požadavky podle objektu, kterým se zabýváme. Například users naznačuje, že se bude jednat o pole uživatele, response se bude zabývat jednotlivými odezvami a podobně. Při vytváření responses pošlu JSON objekt na (POST) http://localhost/response/.

Best practices naznačují způsob, jak URL adresy tvořit:

Objekt	POST	GET	PUT	DELETE
/response	Vytvoří response	Vrátí všechny response	Hromadný update response	Smaže všechny response
/response/1	Chyba	Vrátí response s ID = 1	Updatuje reponse s ID = 1 (pokud existuje)	Smaže Reponse s ID = 1

Tabulka 3 - Response endpointy

V Tabulka 3 na druhém řádku vidíme, že za objektem (v našem případě response) následuje ID, není to nutností, ale z důvodu operací jako je smazání, update a podobně je vhodné využívat nějaký unikátní identifikátor. V tabulce je zvykem ID uvádět, a tak ho využívám.

Pokud za objektem unikátní identifikátor není obsažen, potom se HTTP request týká všech záznamů (provádíme hromadné operace, nebo vypisujeme všechny záznamy z databáze).

4.3.4. Autorizace uživatelů

Autorizace uživatelů je podstatnou částí, kterou Datastore zajišťuje. V aplikační části pracuji se dvěma rolemi – viewer a administrátor. Administrátor má neomezené možnosti, zatímco viewer má oprávnění pouze k prohlížení dat. Nemůže žádná data měnit, ani přidávat. Role obecně lze velmi jednoduše přidávat v případě, že by byly potřeba, ale momentálně jsem nenašel jiná potřebná uživatelská oprávnění pro tento rozsah aplikace.

Prvním krokem je přidávání uživatelů do databáze. Vůči ní se potom uživatelé autorizují. Prvního uživatele je tedy nutné přidat ručně v databázi, další uživatele lze přidávat už pomocí HTTP requestu pomocí grafického rozhraní aplikace (je-li v ní tato funkcionalita implementována). API endpoint je na tuto registraci uživatelů připraven.

Metoda na vytvoření uživatele je jednoduchá, v příchozích datech je obsaženo uživatelské jméno, heslo a role. Uživatelské jméno je unikátní,

v databázi se unikátnost zkontroluje, z hesla se vytvoří hash a celý objekt se uloží do databáze.

Zajímavější kapitolou je samotná autorizace. K autorizaci používáme knihovnu FastAPI, která dále využívá protokolu OAuth2 (teoreticky rozebráno v první kapitole). OAuth2 je navrženo tak, aby mohl být autentizační server mimo server API, v tomto případě je ale implementován přímo do FastAPI, takže autorizace probíhá na stejném místě. Obecně autorizace probíhá tímto způsobem:

- Uživatel zadá své uživatelské jméno a heslo, tyto údaje se odešlou z frontendu na server na specifickou adresu.
- API zkontroluje uživatelské jméno a heslo a vrací token. Za normální okolností tento token expiruje a frontend si musí jeho expiraci pohlídat a případně požádat o obnovu tokenu.
- V dalších HTTP požadavcích se pak uživatel autorizuje pomocí tokenu.

4.3.5. Implementace ověření

V kořenovém souboru je potřebné vytvořit instanci třídy OAuth2Bearer, která je obsažena přímo v knihovně FastAPI (modul security) - Zdrojový kód 12.

```
oauth2_scheme = OAuth2PasswordBearer(
    tokenUrl="token",
)
```

Zdrojový kód 12 - FastApi url cesta

TokenURL přímo definuje URL adresu, na kterou uživatel bude odesílat uživatelské jméno a heslo. V našem případě se bude jednat například o http://localhost/token. Můžeme si ale všimnout podstatného detailu, že tokenURL je definován jako relativní cesta. To je z toho důvodu, že později budeme využívat reverse proxy, celá aplikace se schová za doménu www.itdoc.cz. To nám umožní kompatibilitu napříč DNS.

Námi definované oauth2_scheme můžeme přímo používat jeho závislost (dependency) v dalších metodách - Zdrojový kód 13.

```
@app.get("/items/")
async def read_items(token: Annotated[str, Depends(oauth2_scheme)]):
    return {"token": token}
Zdroj: [19]
```

Zdrojový kód 13 - OAuth2 závislost

Takto můžeme přímo přistupovat k tokenu, později této vlastnosti, která umožňuje používat oauth2 jako závislost využijeme při autorizaci uživatelů k jednotlivým API endpointům. Závislost udělá to, že se podívá na hlavičku požadavku a hledá token (v hlavičce se definuje jako Authorization: Bearer TOKEN). Pokud ji nenajde vrátí se chyba 401 (neautorizováno). V případě, že metoda token najde, tak ho vrátí jako string.

Celý tento proces pro nás ještě nemá ale význam, musíme nadefinovat samotnou autorizaci uživatelů. V mém kódu na tom mám vytvořenou celou řadu tříd, které se o autorizaci starají. Jedná se o OAuth2.py soubor, kde jsou všechny třídy popsány.

Prvně si definuji Pydantic model uživatele - Zdrojový kód 14.

```
class User(BaseModel):
    username: str
    role: Union[int, None] = None
```

Zdrojový kód 14 - Definice modelu uživatele

V kódu mám připravené i další parametry, protože je aplikace připravena na rozšíření o registraci uživatelů, ověření emailu a podobně. Ale nakonec bylo usouzeno, že tyto funkcionality v nynější části nemají smysl, a tak opravdu slouží jen jako příprava.

V dalších částech ještě definuji modely tokenu, uživatele v databázi, ale nyní nemají velký význam. Významná je až třída OAuth2.

V hlavním programu (main.py) mám definovanou URL adresu, která slouží pro přihlašování uživatelů. V té se volá metoda umístěná právě ve třídě OAuth2.

```
@app.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm =
Depends()):
    # find user by username and password
            user     =     oauth2.authenticate_user(form_data.username,
form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    # set expires, create oauth2 token
    access_token_expires = oauth2.get_access_token_expires()
    access_token = oauth2.create_access_token(
        data={"sub": user.username, "scopes": [user.role]},
        expires_delta=access_token_expires,
    )
    return {"access_token": access_token, "token_type": "bearer"}
```

Zdrojový kód 15 - přihlášení, žádost o AccessToken

Jednotlivé vysvětlení metod třídy oauth2 bude následovat, nyní si pouze shrňme, co řeší URL endpoint /token. Jak jsem zmínil na tento endpoint se odkazuje logika při přihlášení. Jakmile se zavolá, očekává v podobě formuláře příchozí data – username a password. OAuth2 přímo definuje tyto dva parametry a nelze tedy odesílat například „user-name“, musí se opravdu jednat o „username“. Při zavolání endpointu token, se zavolá metoda login_for_access_token (Zdrojový kód 15). V podstatě vstupem jsou přihlašovací údaje a výstupem je access_token (OAuth2 token). Nejprve si najdeme uživatele v databázi, pokud neexistuje, tak se vrátí API status UNAUTHORIZED. Pokud uživatel existuje a heslo je správné, tak vygeneruji přístupový token.

Z této části je zřejmé, že jsou hlavní dvě metody ve třídě OAuth2. Názvy trochu matou, protože moje třída zajišťující autentizaci se nazývá OAuth2, zatímco třída pro zabezpečení, kterou využívá FastApi prostřednictvím

OAuth2 se nazývá OAuth2PasswordBearer a já ji v počátku diplomové práce zkráceně nazýval taktéž OAuth2. V dalších částech, pokud budu zmiňovat OAuth2 třídu, tak se jedná o moji třídu definovanou v souboru OAuth2.py.
V této třídě probíhá autorizace těmito kroky:

1) Vyhledání uživatele v databázi
2) Zkontroluji heslo
3) Vytvářím access_token s předem definovanou expirační dobou
4) Do tokenu přidávám ještě payload (jedná se o dodatečná data jako username a roli, v budoucnosti se může jednat o dříve zmíněný email, jméno, příjmení nebo odkaz na uživatelský emotikon).

Dále se ve třídě vyskytují metody určené k získání payload z tokenu, ověření uživatele, že je aktivní – v databází lze uživatele deaktivovat a zamezit mu tím přístup do systému, získání expirační doby tokenu a dalších jednotlivých metod pro zajištění správné funkcionality předchozích metod.

4.3.6. Routes

Stejně jako ve Workeru i zde máme k dispozici několik route (lze si je prohlédnout v Tabulka 4. Každá z nich vytváří segmentaci url adres tak, aby byly rozumně tříděné a přehledné. Datastore obsahuje tyto routy:

- /address
- /response
- /task
- /user

4.3.7. Address

URL	TYP	Účel
/address	GET	Vrací všechny adresy
/address/id	GET	Vrací adresu s konkrétním ID
/address	PUT	Updatuje adresu
/address	POST	Vytváří novou adresu
/address	DELETE	Smaže konkrétní adresu

Tabulka 4 - Address endpointy

Protože jsem v diplomové práci ještě nevysvětlil tvoření těchto endpointů, tak na příkladu této routy address, ji popíšu konkrétněji. Pozdější routy jsou jen kombinacemi dále zmíněných prvků.

FastAPI vytváří tzv. routy a konkretní endpointy vázané na URL adresu. Například při zavolání http://localhost/address/5 se zavolá endpoint, který vrátí adresu s konkrétním ID. Ale jak tento proces funguje.

V Pythonu pomocí FastApi se tyto endpointy poměrně lehce programují. Využívá se přitom knihovny APIRouter importované z FastApi, dále je potřeba tyto routy definovat při spuštění programu (Zdrojový kód 16).

```
from modules.datastore.routers import user, address
app = FastAPI()
app.include_router(user.router)
app.include_router(address.router)
```

Zdrojový kód 16 - importování Routes

Tímto inicializačním prvkem dáme programu informaci o tom, jaké routy má FastAPI využívat. Pokud bychom tento proces neudělali napsali nějakou adresu, kterou FastAPI nezná, tak dostaneme chybovou hlášku, že URL adresa neexistuje.

Routu si lze prohlédnout v Zdrojový kód 17.

```
from fastapi import APIRouter, Depends, HTTPException,Security,Request
from modules.datastore.sql_connector import SqlConnection, SqlAddress
from typing import Optional
from modules.datastore.models import Response, Task, Address
from typing import Any
from modules.datastore.schema import AddressIn, AddressOut,
AddressDelete


router = APIRouter(
    prefix="/address",
    tags=["address"],
    responses={404: {"description": "Not found"}},
```

```
)

from modules.datastore.OAuth2 import OAuth2, User, Token

oauth2 = OAuth2()
sql_connection = SqlConnection()
session = sql_connection.getSession()
sqlAddress = SqlAddress(session)

@router.get("/", status_code=200)
def get_all():
    """
    Get all addresses
    """
    return sqlAddress.get_all()
```

Zdrojový kód 17 - definice ApiRouter

Nejprve naimportuje závislosti a dále definujeme Router, její prefix značí začátek URL adresy. V tomto případě se jedná o /address, který definuje, že výsledná URL adresa bude například: http://localhost/address Tento prefix do routy uvádíme z toho důvodu, že později nemusíme psát před každý endpoint /address, ale můžeme v kódu uvést například jen „/move“, která se zmíněným prefixem vytváří url adresu http://localhost/address/move

Můžeme si všimnout v kódu definováni jednotlivé routy pomocí @router.get("/", status_code=200), první parametry značí právě zmíněnou adresu. Díky definovanému router, už FastAPI ví, že před tento endpoint má umístit prefix /address, takže výsledná adresa vypadá http://localhost/address/

V definici přímo uvádíme i metodu požadavku (GET, POST, DELETE…). Druhý parametr uvádí, jaký výsledný HTTP status se vrátí v případě, že je endpoint úspešný. Nemusíme tedy pak duplicitně při volání return ještě posílat statusový kód. V případě chyby potom vyvoláme status kód speciálně pro konkrétní chybu. Například v metodě sqlAddress.get_all(), který si lze prohlédnout v Zdrojový kód 18.

```
def get_all(self):
        """
        Return all addresses
        """
        try:
            with self.sessions.begin() as session:
                query = session.query(Address)

                address_result = [AddressOut(
                        id = address.id,
                        address = address.address,
                        name = address.name,
                        location = address.location,
                        latitude = address.latitude,
                        longitude = address.longitude,
                        note = address.note,
                        color = address.color,
                    ) for address in query]
                return address_result
        except:
            raise HTTPException(
                status_code=500,
                detail="Database unknown error"
            )
```

Zdrojový kód 18 - vrácení všech definovaných adres

Vidíme, že v případě jakéhokoliv neznámého chybové stavu (v tomto případě se může jednat vpodstatě pouze o chybu týkající se databáze), pak vyvolám HTTP error se statusovým kódem 500 a chybovou hláškou Database unknown error. Tímto byla popsána jednotlivá tvorba URL adres, endpointu a jak se v kódu volají jednotlivé metody přiřazené k endpointům.

4.3.8. Scope

Metoda zmíněná výše by mohla býti volána kýmkoliv nezávisle na tom, jestli k endpointu má nějaké specifické oprávnění. Je zřejmé, že tento stav

není žádoucí. Nechceme, aby si uživatelé bez autorizace mohli zobrazovat všechna data, měnit obsah v databázi a podobně. Přichází tedy na řadu jednotlivé přiřazování rolí a oprávnění uživatelům. V našem datastoru máme definovány dvě role – admin a viewer. Popsány pomocí číselné hodnoty 1 a 2. Ke každému uživateli je tato hodnota pak přiřazena v databázi.

Pokud chceme omezit endpoint na specifickou roli, pak můžeme definovat tyto role přímo v Route ve FastApi způsobem, který lze vidět v Zdrojový kód 19.

```
from fastapi import APIRouter, Depends, HTTPException,Security
@router.post("/", status_code=200)
def    add_response(data:    ResponseAdd,    current_user:    User    =
Security(oauth2.get_current_active_user, scopes=["1","2"])):
    """
    create new row (response) into db
    """
    return sqlResponse.add(data)
```

Zdrojový kód 19 - omezení route (oprávnění)

Importujeme Security obsažené v knihovně FastApi. Pak pouze v parametru metody uvedeme přímo Scopes, které mají mít k metodě přístup. Ve své podstatě je Security také závislostí se dvěma parametry a to Scopes a uživatelem, kterého definuje přístupový token. Pokud zavoláme get_current_active_user, pak si pomocí závislostí zkontrolujeme, zda uživatel patří do příslušného scopu, nebo ne.

Popíšu princip, metoda get_current_active_user se odkazuje na další metody, struktura kódu pro nás ale není příliš zásadní. Jedná se o to, že přeneseně máme v této metodě přístup k tokenu uživatele (ten získáme z HTTP requestu). Díky tomu můžeme zjistit i jaké scope má uživatel k dispozici. Dále je při volání endpointu definováno, jaké scope mají mít k endpointu přístup. Po ošetření všech různých případů (uživatel už neexistuje, token expiroval a podobně) stačí zjistit, zda mají tyto dva listy

scopů průnik. Pokud nemají, kód vrátí HTTPException, jinak uživatele považujeme za ověřeného.

4.3.9. Responses

Jedná se o jednu z nejdůležitějších route. Uživatel aplikace bude požadovat různá data o cílových stanicích a bude v nich chtít různě filtrovat dle parametrů. Cílem programu je poskytnout frontendu data, která bude žádat. Já jsem napsal všechny endpointy, které byly požadovány mým kolegou Martinem Cé. Je ale zřejmé, že není kdykoliv problém jakýkoliv endpoint dopsat. Aktuálně dostupné endpointy lze najít v Tabulka 5

URL	Typ	Parametry	Účel
/response	POST	Response model	Vytváří response
/response/get-all	POST	Time_from, time_to, limit	Vrací responses dle parametrů
/response/delete-all	DELETE	-	Smaže všechny responses
/response/ID	DELETE	Response ID	Smaže response s konkrétním ID
/response/get-average	POST	Time_from, time_to, limit	Vrátí průměrné odezvy dle zadaných parametrů
/response/get-summary	POST	Time_from, time_to, worker_id	Vrátí první testování, poslední testování, průměrnou dobu odezvy, počet testování a to ke všem adresám a přiřazeného workera

Tabulka 5 - Response endpointy

Zde si můžeme všimnout jedné nesrovnalosti, best-practices pro tvorbu URL adres a endpointů nám říká, že bychom měli volat metody pomocí GET, pokud získáváme data, a nikoliv pomocí POST, ten slouží pro insert dat. V tomto případě ale používáme více parametrů (nejedná se pouze

o unikátní identifikátor). Tyto parametry zpravidla žádají svou validaci, aby nebylo umožněno poslat například ve worker_id string „neconeco“. Validaci obvykle provádíme pomocí Pydantic modelu, který nám dává silný a hlavně poměrně jednoduchý nástroj na validaci dat a to včetně custom validace.

Lze napsat vlastní validační procesy – jeden z nich uvedu při validaci IP adresy.

Při definování metody beru jako vstupní data Pydantic model, například:

def get_avrg_all(self, data: ResponseGet):

Který je definován:

```
class ResponseGet(BaseModel):
    time_from: Optional[int]
    time_to: Optional[int]
    limit: Optional[int]
```

Zdrojový kód 20 - Pydantic validace

V Zdrojový kód 20 vidíme, že všechny parametry jsou volitelné, ale zároveň u všech je žádaný formát int. Pokud by se v příchozím JSONu objevil parametr například limit, ale v jiném formátu, došlo by k vyvolání chybové hlášky.

Ve většině metod uvádím hlavní tří parametry: time_from, time_to a limit. Jedná se o to, že můžeme filtrovat Reponses dle časových oken (od – do). Zároveň počet Responses může mít zpravidla velký objem, takže je ještě volitelný parametr limit, který omezí počet výsledků na žádanou úroveň.

Vidíme, že i pomocí poměrně malého množství metod jsme schopni poměrně široce nastavovat parametry výchozích dat tak, aby bylo možné s nimi nadále pracovat na frontendu a uživateli umožnit dostatečné možnosti filtrování.

4.3.10. Tasks

Tasks routy obstarávají jednotlivé úkony ohledně úkolů obstarávající Workery. Uživatel vytváří jednotlivé úkoly, definuje jejich parametry a má

možnost jejich úpravy. Dále se asociují jednotlivým Workerům dle schématu, které jsem zmínil v předchozích kapitolách. Veškeré endpointy lze najít v Tabulka 6.

URL	Typ	Parametry	Účel
/task	POST	Model TaskIn	Vytváření tasku
/task/asssociate	POST	Model TaskAssociate	Asociování Workera
/task/id	DELETE		Smaže task
/task/associate-delete	POST	Model TaskAssociate	Smaže přiřazení tasku k Workeru
/task	PUT	Model TaskOut	Upravuje parametry
/task/pause/id	PUT		Pozastavuje Task
/task/active/id	PUT		Zaktivňuje Task
/task/hide/id	PUT		Skryje Task
/task/unhide/id	PUT		Znovu zobrazí Task
/task/worker-tasks/id	GET	ID workera	Získá přiřezené tasky k Workeru
/task/active-task/id	GET	ID workera	Získá aktivní tasky přiřazené k Workeru
/task	GET		Získá všechny tasky

Tabulka 6 - Task endpointy

Většina requestů je na vyžádání od mého kolegy Martina Cé. Jedná se hlavně o možnosti získávat data z databáze pro grafické rozhrání. S tím souvisí i úprava hide/unhide. Tím se může uživatel měnit nastavení tak, jestli ho chce ve výchozím nastavení zobrazovat ve výpisu, nebo nikoliv.

4.3.11. Custom Validace

V jedné z předchozích kapitol jsem se zmínil v možnosti vytváření vlastních validací v modelu Pydantic. Pro samotnou validaci je připraveno několik možností, které můžeme využít – například validaci na celé číslo, desetinné, email apodobně. V některých případech si ale s těmito předpřipravenými validacemi nevystačíme. Já jsem takovou custom validaci připravil pro IP adresu. Ta může vypadat například takto 192.168.10.10, jedná se vždy o čtyři bloky čísel, každý o velikosti čísla 0-255. (Jedná se o IP adresu verze 4). Validaci bychom tedy mohli řešit tím způsobem, že bychom zkontrolovali tento formát. Já jsem přistoupil na sofistikovanější způsob.

Nejprve jsem si vytvořil třídu DataValidation, kterou si můžete prohlédnout v Zdrojový kód 21.

```
import ipaddress

class DataValidation():

    def validate_ip_address(self, value):
        try:
            ip = ipaddress.IPv4Address(value)
            return str(ip)
        except ipaddress.AddressValueError:
            raise ValueError("Neplatná IPv4 adresa")
```

Zdrojový kód 21 – Třída DataValidation

Zde je uvedena funkce validate_ip_address, která má jako vstup value (IP adresa). Potom pomocí knihovny ipaddress se snažím zavolat funkci IPv4Address z knihovny ipaddress s parametrem IP adresy. Tato funkce vrací znovu objekt IP adresy, ale v případě, že se nejedná o validní IP adresu, potom se vrací ValueError.

V pydantic modelu potom mohu definovat vlastní validátor (Zdrojový kód 22) .

```
@validator('address')
def validateAddress(cls, value):
    validation = DataValidation()
```

```
        return validation.validate_ip_address(value)
```

Zdrojový kód 22 - Custom Validátor

(Poznámka: je nutné importovat závislosti).
V tomto případě validátor vrátí IP adresu, pokud je IP adresa validní, v jiném případě dojde k vyvolání chyby.

Momentálně program funguje na testování pouze IPv4 adres, takže validuji pouze tuto verzi. V případě budoucího rozšíření je možné i validátor jednoduše rozšířit, protože knihovny ipaddress obsahuje i funkci IPv6Address na validaci adres verze 6.

4.3.12. Autenzizace Workera

Autentizace Workera oproti autentizaci uživatelů probíhá jiným způsobem než přes přihlašovací jméno a heslo. Jak jsem už nastínil v teoretické kapitole zabývající se autentizací, tak probíhá ověřování pomocí API tokenu. Tento způsob má systémové výhody, které nám umožní přidávat další Workery a dále je jednoduše identifikovat. (Identifikace probíhá právě přes unikátní identifikátor). Nemusíme zbytečně generovat uživatelská jména a k nim hesla. Pokud potřebujeme Workera vyřadit z provozu, stačí zneplatnit API token.

V našem systémů probíhá komunikace DataStore-Worker vpodstatě pouze v jedné rovině a tou je funkcí synchronizace. Kdy DataStore nabízí data zabývající se úkolů, které má Worker zpracovávat. Naopak Worker pak synchronizuje s DataStorem jednotlivé Responses a HostStatus.

V každém požadavku na synchronizaci, který požaduje Worker, se odesílá rovnou jako jeden z parametrů v requestu POST API token. Díky němu mohu na stránce API najít konkrétního Workera v databázi a tím zjistit i jeho ID, které je potřebné pro další zpracování požadavku.

Pokud Worker není nalezen, potom vracím chybu 401, unauthorized.

5. Nasazení

Program máme hotový a nyní je potřeba celou záležitost nasadit a spustit do provozuschopného stavu. K tomu budeme potřebovat jakýkoliv hardware, na kterém poběží operační systém Windows nebo jakýkoliv linuxový stroj, na kterém jsme schopni zprovoznit Python. Vzhledem k tomu, že se budu věnovat i nasazení v kontejnerizovaném prostředí Docker, který spolehlivě lze nainstalovat na linuxovém stroji, tak běžné nasazení provedu na stroji s Windows a kontejnerizované nasazení na stroji s linuxovou distribucí Ubuntu. Dalším nutným požadavkem pro spuštění aplikace je mít nainstalovaný Python a další nutné závislosti, které jsou po stažení repozitáře k nalezení v souboru requirements.txt v kořenovém adresáři. Vzhledem k tomu, že naše aplikace může běžet i v zabezpečeném režimu v protokolu HTTPS, bude potřeba si vygenerovat SSL certifikát.

Program lze najít na zveřejněném GIT repositáři na adrese https://github.com/matousc89/network_monitor.

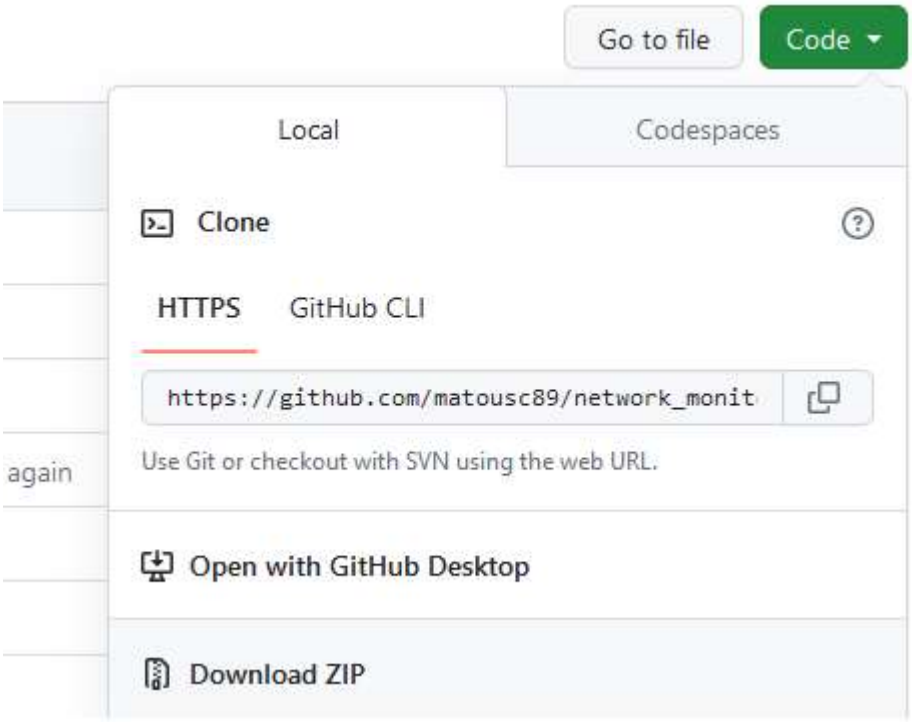
5.1. Instalace Windows

5.1.1. Instalace Pythonu

Instalace Pythonu je poměrně jednoduchá, stačí jít na stránky www.python.org a v sekci Downloads si stáhnout odpovídají Installer pro Váš operační systém. Pomocí instalátoru je instalace velmi jednoduchá, pouze doporučuji zvolit možnost přidat Python do cesty Windows. Díky této možnosti se nám zjednoduší práce v příkazové řádce, protože Windows bude znát cestu k Pythonu.

5.1.2. Stažení repositáře

Stažení repositáře můžeme provést dle námi preferované možnosti. Můžeme využít prostředí Gitu, pokud ho máme nainstalované, nebo nějaké grafické prostředí jako GitKraken. Nejrychlejší cestou, pokud nemáme nainstalovaný žádný Git klient, je navštívit WWW adresu, kde je nahrán repozitář a celý ho stáhnout v ZIPu. V GitLabu stačí kliknout na tlačítko Code a dále Download ZIP.



Obrázek 7 - stažení repositáře

Poté stačí už jen extrahovat složku a v ní si spustit příkazový řádek.

5.1.3. Instalace knihoven

Instalaci knihoven provedeme velmi jednoduše díky souboru requirements.txt, který obsahuje všechny knihovny potřebné pro správný běh programu.

Stačí napsat tento příkaz:

```
pip install -r requirements.txt
```

Poznámka: Aby příkaz fungoval, je nutné mít příkazový řádek spuštěný v kořenovém adresáři staženého programu. Pokud zadáte příkaz ***dir***, tak soubor requirements.txt musí být mezi výsledky.

5.1.4. Spuštění programu

Jakmile se nainstalovaly všechny knihovny, už nám nic nebrání program spustit. Aby byla inicializace, co nejjednodušší, je připraven soubor run.py v kořenovém adresáři. V příkazové řádce stačí zadat příkaz:

```
Python run.py
```

V tuto chvíli by mělo být dostupné API na lokální adrese 127.0.0.1 a výchozím portu 8000.
Pokud zadáme do webové adresní řádky localhost:8000, měl by se zobrazit status API. Viz Obrázek 8.

Obrázek 8 - Status API

V tuto chvíli ale nemáme nastavené žádné testování adres. Při zapnutí programu je automaticky vytvořena databáze, která se nachází ve složce databases. Vzhledem k tomu, že přidávání uživatelů je umožněno pouze administrátorům, tak bychom nemohli po instalaci využívat systém a uživatele bychom museli ručně vytvářet v databázi. Abychom se tomuto kroku vyhnuli, je program napsán tak, že při vytváření databáze automaticky vytvoří uživatele **admin** s heslem: **passpass**

Těmito přihlašovacími údaji se můžeme autorizovat vůči API. V grafickém rozhraní by nám mělo být umožněno kompletně ovládat program.
V případě, že ho nemáme k dispozici, můžeme využít nástroje poskytovaném FastAPI, který je k nalezení na www adrese localhost:8000/docs.

5.1.5. Zprovoznění testování adres

5.1.6.HTTPS

Abychom program alespoň částečně otestovali, zkusíme si vytvořit do databáze jednu IP adresu, vytvořit Workera a IP adresu přiřadit Workerovi.

1) Vytvoření Workera v databázi

Tento krok je nutný k tomu, aby byl DataStore schopen Workera autorizovat. Díky tomu musí mít informaci o tom, že Worker má oprávnění zapisovat do databáze DataStoru. Záznam vytvoříme přes endpoint /create-worker.
Celý http request by mohl vypadat například takto:

http://localhost:8000/create-worker?worker_name=Default&api_key=9d207bf0-10f5-4d8f-a479-22ff5aeff8d1'

Tímto vytvoříme do databáze Workera, který se jmenuje Default. API klíč je zvolen tak, jak je nastaven výchozí API klíč pro Workera v souboru settings.py. Samozřejmě můžeme vytvořit jakýkoliv jiný a následně v souboru settings.py upravit proměnnou WORKER_API. Pro přidání můžeme využít nástroj na adrese http://localhost:8000/docs

2) Vytvořit IP adresu, kterou chceme testovat

Já si zvolil adresu patřící Googlu (8.8.8.8), obsah requestu by mohl vypadat například takto:

```
{
  "address": "8.8.8.8",
  "name": "Google",
  "location": "Bahama",
  "latitude": 0,
  "longitude": 0,
  "note": "Adresa Google",
  "color": "black"
}
```

Adresa endpointu je: /address (pro vytvoření, type POST), v našem případě by mohla vypadat takto: http://localhost:8000/address/

Výstupem je:

```
{
  "address": "8.8.8.8",
  "name": "Google",
  "location": "Bahama",
  "latitude": 0,
  "longitude": 0,
  "note": "Adresa Google",
  "color": "black",
  "id": 1
}
```

3) Vytvoření tasku s IP adresu

Jak jsem zmínil v předchozích kapitolách, tak je umožněno, aby každému Workeru bylo možné jednu konkrétní adresu přiřadit s různou periodou testování. K tomu slouží tasks, který definují jako adresu a s jakými parametry má být adresa testována. Cílový endpoint je /tasks

Obsah ukázkového requestu:


```
{
  "address_id": 1,
  "running": true,
  "hide": false,
  "task": "ping",
  "frequency": "10s",
  "retry": 2,
  "timeout": 1,
  "treshold": 1,
  "retry_data": {}
}
```


Odpověď:


```
{
  "id": 1,
  "running": true,
  "hide": false,
  "address": "8.8.8.8",
  "task": "ping",
  "frequency": "10s",
  "name": "Google",
  "location": "Bahama",
  "latitude": 0,
  "longitude": 0,
  "note": "Adresa Google",
  "color": "black",
  "timeout": 1,
  "retry": 0,
  "treshold": 50,
  "retry_data": "{}"
}
```


4) Přiřazení tasku Workeru

Posledním krokem je přiřazení tasku Workeru. Konkrétní task může být přiřazen více Workerům, takže tímto dáme vědět, jaký Worker má task zpracovávat.

Endpoint je /task/associate

Request:

```
{
  "taskId": 1,
  "workerId": 1
}
```

Odpovědí je pouze status code = 200.

Tímto máme vše potřebné nastavené a pokud zavoláme například endpoint /response/get-summary získáme tento možný výsledek:

```
[
  {
    "address": "8.8.8.8",
    "first_response": 1704544928409,
    "last_response": 1704545018409,
    "average": 3.8,
    "count": 10
  }
]
```

Vidíme, že Worker zpracovává periodicky task, odesílá výsledky do DataStoru a posláním requestu na výpis souhrnu nám vrací průměrnou odezvu.

5.1.7.HTTPS

Abychom komunikaci zabezpečili, musíme nejprve vygenerovat SSL certifikát, můžeme využít nějakou certifikační autoritu a nebo si ho vygenerovat sami (self-signed certifikát). Já jsem pro testovací účely jeden certifikát vygeneroval a přímo uložil do GIT repositáře. Není vhodné ho používat pro ostrou verzi, protože tento certifikát je veřejně dostupný, a tak kdokoliv, kdo k certifikátu získá přístup, bude moci komunikace dešifrovat. Pro ilustrační účely ale nyní dostačuje. Ať už používáme jakýkoliv certifikát, tak pro zabezpečení komunikace stačí v souboru run.py odkomentovat řádky, které zmiňuji ve Zdrojový kód 23.

```
    uvicorn.run("modules.datastore.main:app", port=port, host=address)
```

```
#             reload=True, reload_dirs=['html_files'],
#             ssl_keyfile='certificate/Local1Key.pem',
#             ssl_certfile='certificate/Local1crt.pem')
```
Zdrojový kód 23 - Zabezpečení komunikace

5.2. Docker

Nasazení v kontejnerizovaném prostředí provedu v Dockeru. Nejdříve než se k nasazení dostaneme, uvedu několik málo technických informací ohledně kontejnerizace jako je image, baselmage a co je samotný kontejner.

Baselmage je základní obraz, na kterém je postavena konkrétní aplikace. Obsahuje minimální konfigurace a nástroje nutné pro běh aplikace. Jedná se v podstatě o takový jednoduchý operační systém pro aplikaci. Každý si můžeme tento Baselmage vytvořit sami, ale většinou můžeme použít už nějaký vytvořený například komunitou, nebo přímo Dockerem jako je CentOS.

Image: Jedná se o virtualizovaný obraz obsahující veškerý potřebný software a konfigurace pro běh aplikace. Image je nezávislý na konkrétním prostředí a poskytuje konzistentní prostředí pro aplikaci [20]. Ve své podstatě si to můžeme představit tak, jako bychom si nainstalovali operační systém (Baselmage), nahráli celý program a zprovoznili ho tak jako jsem udělal v předchozí kapitole při nasazení na OS Windows. Pokud bychom program měli už k distribuci, mohli bychom komunitě tento Image připravit a tím celé usnadnit nasazení aplikace. Uživatel by si jen stáhl Image a spustil.

Kontejner: Je instance běžící aplikace v izolovaném prostředí. Kontejner sdílí jádro operačního systému s hostitelským systémem. Každá aplikace je pak téměř izolovaná od ostatní [20]. Pokud bychom chtěli zprovoznit na jednom stroji více instanci naší aplikace, bylo by to možné a navzájem by na sebe „neviděli“. Samozřejmostí je ale použití odlišných portů. Zmínil jsem, že aplikace je téměř izolovaná. Jedná se o to, že stále sdílí stejné jádro operačního systému a tím tedy i kernel systému. Na rozdíl

od virtualizace, která je softwarově zcela izolovaná, tak v případě kontejnerizace toto neplatí.

5.2.1. Návrh systému

Při tvorbě vlastního Image využijeme skutečnosti, že je možné tvořit řetěz – za výchozí Image vezmu jeden a z něj udělám druhý (ale s přidanou funkcionalitou). Mohl bych využít BaseImage, nainstalovat Python, závislosti a podobně a z něj udělat svůj vlastní Image. Ale namísto toho, můžeme už využít předpřipravený Image, který Python obsahuje. Ten si pak upravíme k našim potřebám.

Obecnou nevýhodou používání Pythonu v kontejnerizovaném prostředí je, že Image má poměrně velkou velikost (v řádech nižších GB). Existují i odlehčené verze, ale bohužel ty pro naši aplikaci nejsou dostatečné, protože tyto upravené varianty nepodporují některé z našich knihoven.

5.2.2. Dockerfile

Dockerfile je textový soubor obsahující sérii instrukcí, které popisují, jak vytvořit Docker image. Tento soubor definuje kroky potřebné k sestavení Image pro konkrétní aplikaci nebo službu. Dockerfile může obsahovat instrukce pro stahování potřebných závislostí, konfiguraci prostředí, kopírování souborů a mnoho dalších úkonů [13]. Já jsem Dockerfile vytvořil a nahrál přímo do kořenového adresáře programu. Pokud bychom chtěli vytvořit dockerImage, stačí zadat tento příkaz:

```
docker build -t network .
```

Tím máme obraz vytvořený.

5.2.3. Traefik

Většina webových aplikací běží na portu 80. A v případě, že chceme například provozovat více webových aplikacích na jednom stroji, pak je uživatelský vhodné, aby všechny aplikace běžely na tom stejném portu.

Například webové prohlížeče bez uvedení portu automaticky předpokládají, že se jedná o port 80. Bylo by zbytečné, abychom pro každou aplikaci museli mít vlastní IP adresu. Namísto toho můžeme využít reverse proxy.

Reverse proxy směruje provoz od klienta k serveru tak, jak potřebujeme. Může sloužit i jako třeba load balancer, kdy směřuje komunikace klienta k více serverům. My reverse proxy využijeme ze dvou důvodů. Umožní nám provoz více webových aplikacích na jednom stroji s jednou IP adresou a na stejném portu. A dále některé reverse proxy umí automaticky získávat SSL certifikáty a samostatně je i obnovovat.

Já jsem zvyklý používat Traefik, ale existují i různé alternativy jako Nginx. Funkcionalitou jsou v podstatě srovnatelné. Tento krok využít Traefik je však nepovinný pro zprovoznění aplikace.

5.2.4.Docker-compose

Docker-compose je nástroj pro definování a spouštění multi-container Docker aplikací. Umožňuje popsat všechny služby, sítě, úložišť (volumes) a další konfigurace naší aplikace v jednom souboru nazývaném docker-compose.yml [13].

S docker-compose můžeme definovat a spravovat celý stack služeb pro náš projekt jako jednu jednotku, což usnadňuje jejich spuštění, zastavení a správu. Pokud provozujeme nějakou aplikaci, zpravidla potřebujeme například frontend, backend a nějakou databázi. Abychom tento celek mohli spravovat, využíváme docker-compose.
Pro příklad uvádím můj docker-compose ve Zdrojový kód 24.

```
version: '3'
services:
  web-fastapi:
    image: python-fastapi:latest
    build: python-fastapi
    ports: ["8005:80"]
    volumes:
     - python-fastapi:/usr/src/app
```

```
    command: bash -c "python /usr/src/app/run.py"
    restart: always
    labels:
     - traefik.enable=true
     - traefik.http.routers.python-fastapi.rule=Host(`www.itdoc.cz`,`itdoc.cz` )
     - traefik.http.routers.python-fastapi.entrypoints=web
     - traefik.http.middlewares.test-redirectscheme.redirectscheme.scheme=https
     - traefik.http.routers.python-fastapi-https.tls=true
     - traefik.http.routers.python-fastapi-https.tls.certresolver=myresolver

     - traefik.http.services.python-fastapi.loadbalancer.server.port=80
     - traefik.http.routers.python-fastapi.service=python-fastapi
     - traefik.docker.network=traefik_default

     -traefik.http.middlewares.mywebserver-redirect-web-
secure.redirectscheme.scheme=https
     -traefik.http.routers.python-fastapi.middlewares=mywebserver-redirect-web-
secure



     - traefik.http.routers.python-fastapi-https.rule=Host(`itdoc.cz`,`www.itdoc.cz`)
     - traefik.http.routers.python-fastapi-https.entrypoints=websecure
    networks:
     - traefik_default
volumes:
  python-fastapi:
    external: true

networks:
  traefik_default:
    external: true
```

Zdrojový kód 24 - Docker-compose

Stručný popis:

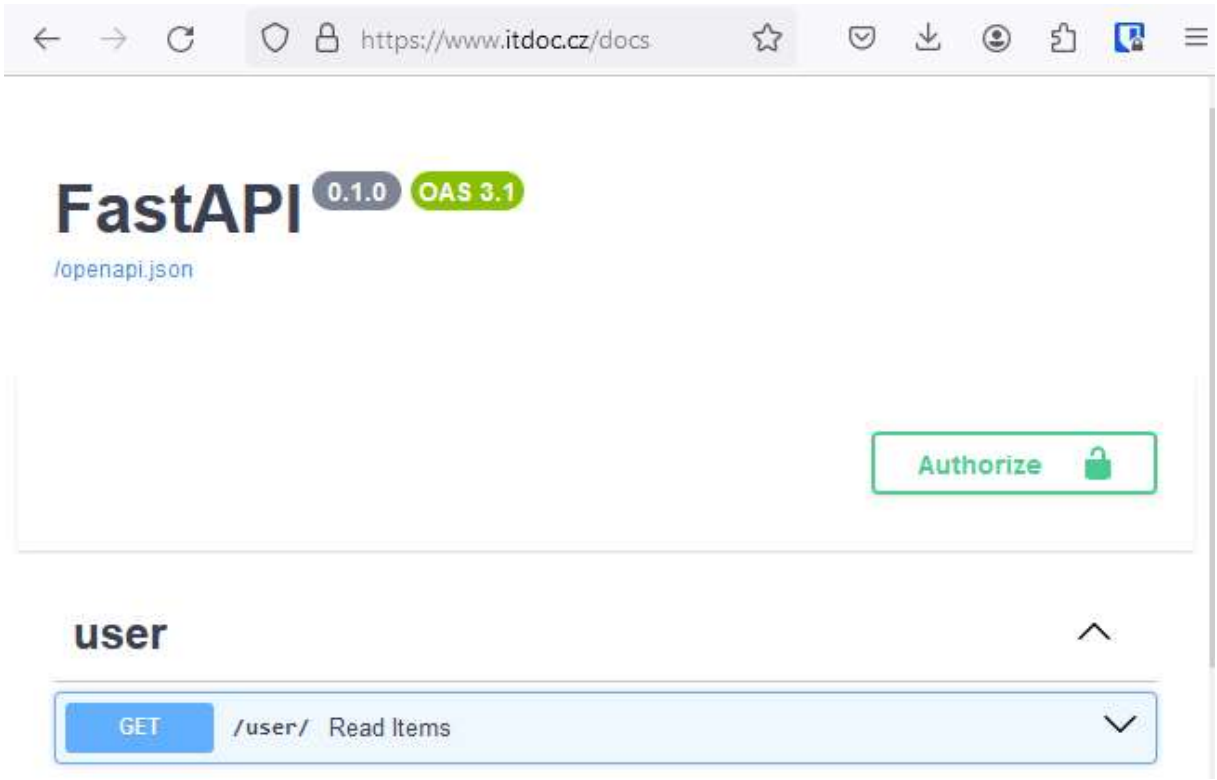
Nejprve určím image, který jsem vygeneroval dříve. Následně definuji porty – port 8005 se přesměruje na port 80 aplikace. Volumes je persistentní úložiště, kam můžeme ukládat data, aniž bychom si je redyploymentem přemazali. V labels si definuji informace pro traefik, můžeme si hlavně

všimnout, že probíhá přesměrování (routers) z doménového jména itdoc.cz (a jejich variace jako www.itdoc.cz), zároveň veškerá komunikace z http se přesměruje na https. Jednoduchým příkazem pomocí tls.certresolver si zajistíme certifikát vydaný LetsEncryptem.

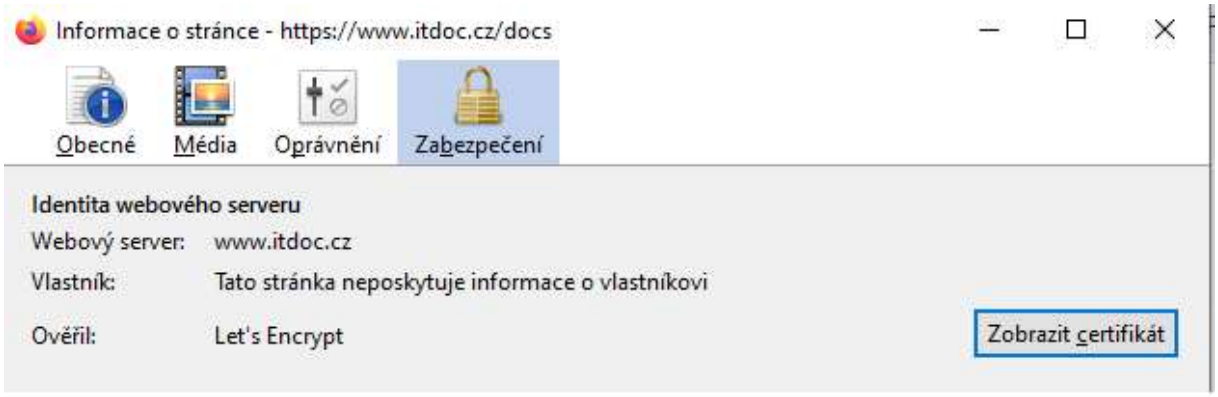
5.2.5. Ověření funkčnosti

Aplikaci jsem nasadil na veřejné IP adrese, zároveň jsem nastavil DNS záznamy v doméně itdoc.cz. Komunikace by měla být zabezpečena pomocí SSL certifikátu, který se bude automaticky obnovovat. Když tedy přistoupíme na adresu http://www.itdoc.cz/docs, měla by se nám zobrazit stránka našeho API. Výsledek si můžeme prohlédnout na **Chyba! Nenalezen zdroj odkazů.**.

Pokud si rozklikneme zabezpečení, mělo bychom najít informace o LetEncrypt certifikátu. Ukázku můžete najít na Obrázek 10.



Obrázek 9 – Ukázka aplikace na doméně ItDoc.cz



Obrázek 10 - zabezpečení na doméně ItDoc.cz

6. Závěr

V této diplomové práci jsem se zaměřil na rozvoj a implementaci rozšíření pro aplikaci monitorující dostupnost zařízení na síti. Cílem bylo nejen rozšíření funkčnosti stávající aplikace, ale také zajištění vyšší úrovně zabezpečení a autentizace v rámci procesu monitorování.

Během práce jsem představil teoretické základy, které jsou nezbytné pro pochopení fungování a bezpečnosti síťové komunikace. V praktické části jsem detailně popsal proces vývoje rozšíření od návrhu až po implementaci, včetně použitých technologií a metodik. Zásadní částí práce bylo také testování a validace funkčnosti a bezpečnosti navrženého řešení.

Hlavními přínosy této diplomové práce jsou:

- rozšíření existující aplikace o nové funkcionality a bezpečnostní prvky
- důkladná analýza a implementace pokročilých bezpečnostních mechanismů
- demonstrace praktické aplikace teoretických principů v oblasti síťového monitoringu a zabezpečení

Výsledky mé práce přispívají k lepšímu pochopení a řešení problémů spojených s monitorováním a zabezpečením zařízení v síti. Rozšíření aplikace nabízí nejen vylepšené funkcionality, ale také klade důraz na zabezpečení, což je v dnešním digitálním světě nezbytné.

Pro budoucí rozvoj v této oblasti navrhuji zaměřit se na další rozšíření funkčnosti aplikace, včetně integrace s dalšími nástroji pro správu sítě a bezpečnost. Dále by bylo vhodné zkoumat možnosti využití umělé inteligence a strojového učení pro lepší detekci a reakci na neobvyklé události v síti.

Tato diplomová práce tak představuje krok v rozvoji nástrojů pro monitorování a zabezpečení síťových zařízení a otevírá cestu pro další inovace v tomto odvětví.

Seznam tabulek

Seznam obrázků

Bibliografie

1. SSL Dragon. *SSL Dragon.* [Online] 27. 05 2023. https://www.ssldragon.com/blog/ssl-stats/.

2. Foltýn, Tomáš. WeLiveSecurity. *WeLiveSecurity by Eset.* [Online] 03. 9 2018. https://www.welivesecurity.com/2018/09/03/majority-worlds-top-websites-https/.

3. Nicolas. Types of Encryption: Symmetric or Asymmetric? RSA or AES? *preyproject.* [Online] 15. 6 2021. https://preyproject.com/blog/types-of-encryption-symmetric-or-asymmetric-rsa-or-aes.

4. Anicas, Mitchell. DigitalOcean. *DigitalOcean.* [Online] 21. 06 2014. https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2.

5. mozzila.org. [Online] [Citace: 2. 10 2023.] https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication.

6. Levin, Guy. 4 Most Used REST API Authentication Methods. *Blog Restcase.* [Online] 26. 07 2019. https://blog.restcase.com/4-most-used-rest-api-authentication-methods/.

7. Arias, Dan. Adding Salt to Hashing: A Better Way to Store Passwords. *auth0.* [Online] 25. 02 2021. https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/.

8. Grigutytė, Monika. What is bcrypt and how does it work? *NordVPN.* [Online] 16. 06 2023. https://nordvpn.com/blog/what-is-bcrypt/.

9. Koďousková, Barbora. HTTPS v kostce: co to je, jak funguje a jak na něj přejít. *rascacone.com.* [Online] 17. 11 2021. https://www.rascasone.com/cs/blog/co-je-https-http-ssl-tls.

10. Prodromou, Agathoklis. TLS Security 2: A Brief History of SSL/TLS. *acunetix.com.* [Online] 31. 3 2019. https://www.acunetix.com/blog/articles/history-of-tls-ssl-part-2/.

11. What's the difference between SSL and TLS? *amazon.* [Online] https://aws.amazon.com/compare/the-difference-between-ssl-and-tls/.

12. Transport Layer Security (TLS). [Online] https://hpbn.co/transport-layer-security-tls/.

13. Lim, Matt. Diffie-Hellman Key Exchange . *medium.com.* [Online] 12. 10 2020. https://pencilflip.medium.com/diffie-hellman-key-exchange-7dba8e9e59d6.

14. Man in the middle (MITM) attack. *Imperva.* [Online] https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/.

15. Awati, Rahul. What is a certificate authority (CA)? *TechTarget.* [Online] https://www.techtarget.com/searchsecurity/definition/certificate-authority.
16. *medium.com.* [Online] 5 2023. [Citace: 19. 12 2023.] https://medium.com/@kadergenc/what-is-orm-why-is-it-used-what-are-its-pros-and-cons-3ed77c0e6ed2.
17. vegibit.com. *Vegibit.* [Online] [Citace: 01. 08 2024.] https://vegibit.com/django-orm-vs-sqlalchemy/.
18. Cé, Martin. čvut dspace. *čvut.* [Online] https://dspace.cvut.cz/handle/10467/110057.
19. fastapi.tiangolo.com. *FastApi.* [Online] [Citace: 2. 11 2023.] https://fastapi.tiangolo.com/tutorial/security/first-steps/.
20. Foster, Liam. *docker for Beginners.* místo neznámé : Lightning Source Inc. 978-1-80149-049-8.
21. Chipeta, Catherine. What is HTTPS? How it Works and Why It's So Important. *upguard.com.* [Online] 07. 02 2023. https://www.upguard.com/blog/what-is-https.
22. ClickSSL. *ClickSSL.* [Online] 10. 8 2023. https://www.clickssl.net/blog/ssl-statistics.