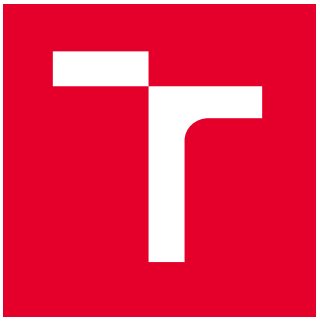


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

MONITOROVACÍ PRVEK SOFTWAROVÝCH APLIKACÍ

MONITORING MODULE FOR SOFTWARE APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Denys Partnov**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Václav Oujezský, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Bc. Denys Partnov ***ID:*** 206683

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Monitorovací prvek softwarových aplikací

POKYNY PRO VYPRACOVÁNÍ:

Práce se zabývá návrhem monitorovacího software běžících procesů v operačním systému Linux a softwarových aplikací. Jako výsledný frontend pro uživatelské ovládání bude využit webový framework Flask a pro backend řešení framework Spring a programovací jazyk Kotlin. Komunikace mezi jednotlivými prvky systému bude zajištěna za pomoci API (Application Programming Interface), který musí být efektivně zabezpečen proti jeho zneužití. V teoretické části práce stručně popište využité frameworky a uveďte návrh aplikace. Výstupem praktické části práce je funkční a na reálném provozu ověřený vytvořený monitorovací modul s instancemi dohledu pro dohodnuté procesy a aplikace.

DOPORUČENÁ LITERATURA:

[1] Building web applications with Spring Boot and Kotlin, [online], 2022, [cit. 2022-9-1], dostupné z: https://spring.io/guides/tutorials/spring-boot-kotlin/

[2] Mark Pilgrim, Ponořme se do Pythonu 3, [online], 2001-11, [cit. 2022-9-1], dostupné z: http://diveintopython3.py.cz/index.html

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 19.5.2023

Vedoucí práce: Ing. Václav Oujezský, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Práce se zabývá návrhem a implementací řešení pro monitorování a ovládání aplikací, běžících v operačním systému Linux nebo pomocí platformy Docker. Implementované řešení se skládá z agentů v programovacím jazyce Kotlin s využitím frameworku Spring, který odpovídá za ovládání a monitorování aplikace. Další částí řešení je Flask aplikace v Python, která nabízí webové rozhraní pro reprezentaci výsledků monitorování a možnost ovládání běžících procesů. Komunikace mezi agenty a webovým rozhraním je zajištěna pomocí API, které je vhodným způsobem zabezpečeno. Implementované řešení bylo otestováno v reálném provozu a byla prokázána jeho funkčnost.

KLÍČOVÁ SLOVA

Monitorování, bezpečnost, Linux, Kotlin, Spring, Flask, Docker, API, OpenAPI

ABSTRACT

The master's thesis deals with the design and implementation of solution for monitoring and controlling applications running in the Linux operating system or using the Docker platform. The implemented solution consists of agents in the Kotlin programming language using the Spring framework, which is responsible for controlling and monitoring the application. Another part of the solution is the Flask application in Python, which offers a web interface for representing monitoring results and the possibility of controlling running processes. Communication between the agents and the web interface is ensured using an API that is appropriately secured. The implemented solution was tested in real production environment and its functionality was proven.

KEYWORDS

Monitoring, security, Linux, Kotlin, Spring, Flask, Docker, API, OpenAPI

PARTNOV, Denys. *Monitorovací prvek softwarových aplikací*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 76 s. Diplomová práce. Vedoucí práce: Ing. Václav Oujezský, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Denys Partnov
VUT ID autora:	206683
Typ práce:	Diplomová práce
Akademický rok:	2022/23
Téma závěrečné práce:	Monitorovací prvek softwarových aplikací

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Václav Oujezský, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam výpisů

Úvod

Diplomová práce se věnuje problematice monitorování softwarových aplikací v operačním systému Linux. V rámci teoretické části diplomové práce jsou popsané způsoby monitorování a možnosti ovládání procesů v Linux. Následně jsou popsané frameworky Spring v programovacím jazyce Kotlin a microframewrok Flask v Python a způsoby jejich využití pro implementace řešení zadání práce. V neposlední řadě teoretická část práce se věnuje standardům při návrhu *Representational State Transfer* (REST) *Application Programming Interface* (API) rozhraní a přehledům zranitelnosti a způsobům zabezpečení API. Jsou zde popsané principy specifikace OpenAPI a následně nejčastěji vyskytující bezpečnostní problémy při návrhu API rozhraní a způsoby jak se vyhnout možným bezpečnostním chybám.

V rámci praktické části diplomové práce je řešeno zadání práce, které spočívá v monitorování softwarových prvků. Jako výsledný frontend pro uživatelské ovládání je využít webový framework Flask a pro backend řešení framework Spring a programovací jazyk Kotlin. Komunikace mezi jednotlivými prvky systému je zajištěna za pomoci REST API a protokolu *Hypertext Transfer Protocol Secure* (HTTPS). Praktická část se skládá ze třech kapitol: návrh řešení, implementace a testovací provoz. Návrh řešení obsahuje popis problémů které jsou řešené během implementace, navíc jsou zde uvedené struktury backend a frontend částí. V rámci implementační kapitoly jsou popsané jednotlivé moduly v backend a frontend řešení. Každý modul řeší předem definovaný cíl a určitým způsobem komunikuje s ostatními moduly. Jsou zde také uvedené příklady implementace pro nejdůležitější body v systému. V neposlední řadě je popsaná implementace zabezpečení celého řešení, která spočívá v autorizaci požadavku pomocí API klíčů. Jako poslední krok byla připravená dokumentace REST API rozhraní podle OpenAPI specifikace.

Poslední kapitola se věnuje testovacímu provozu kde jsou uvedené podmínky pro spuštění serverů a obecný postup přípravy a konfiguraci implementovaného řešení. Dále je uveden popis testovací infrastruktury, která se skládá ze třech serverů v společné síti. Následně byla znázorněna funkčnost monitorování všech 5 aplikací a to jak systémových aplikací tak i Docker kontejnerů na testovacím provozu.

1 Procesy v operačním systému Linux

Kapitola se věnuje problematice procesům v operačním systému Linux, správě procesů a monitorování stavu aplikace. Důležitou součástí kapitoly je popis vzniku procesů, průběh a zánik procesů v operačním systému.

1.1 Popis a struktura procesu

Proces v jádře operačního systému Linux je jednoduše reprezentován jako struktura s mnoha poli. Mezi nejdůležitější poli procesu, patří následující:

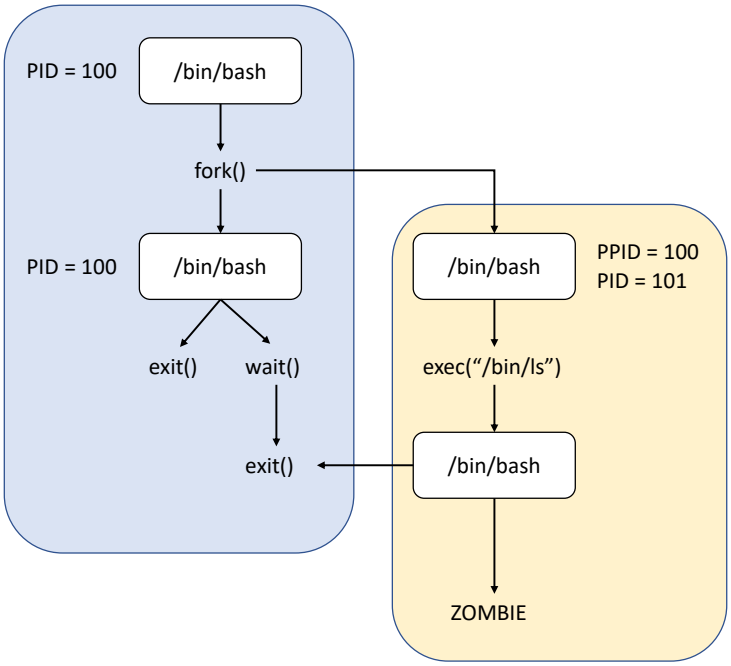
- Identifikátor procesu: *Process Identification* (PID)
- Otevřené deskriptory souborů: *File Descriptor* (FD)
- Obsluhy signálů: *Signal Handler* (SH)
- Aktuální pracovní adresář: *Current Working Directory* (CWD)
- Proměnné prostředí: environ
- Návratový kód

V operačním systému ve stejnou chvílí existují stovky a tisíce procesů, přičemž pouze jeden proces může být zpracován procesorem v určitý okamžik. To znamená, že procesor umožní používat systémové prostředky jako jsou operační paměť nebo přístup do souborového systémů, pouze jednomu procesů a to na dobu do několika stovek milisekund. Pořadí, ve kterém se budou zpracovávat procesy, se řídí určitou frontou. Tuto frontu zpracovává zvláštní část operačního systému – plánovač procesů. Zároveň se při přerušení jednoho procesu a spuštění (obnovení) jiného procesu, zapamatuje se stav přerušeného procesu a zapíše se do oblasti v paměti. Plánovač v operačním systému Linux je součástí jádra odpovědného za tuto funkci. Mezi úkoly plánovače dále patří sledování a přidělování určité priority běžícím procesům, aby se procesy navzájem „nerušily“, a také přidělování paměťového prostoru tak, aby se paměťový prostor jednoho procesu neprotínal s prostorem jiného.[23]

1.2 Životní cyklus procesu

Všechny nové procesy v Linuxu jsou vytvořeny klonováním některého existujícího procesu voláním systémových funkcí `clone()` a `fork()`. Nový proces má stejné prostředí jako rodič, liší se pouze ID procesu. Specialitou nového procesu je položka *Parent Process Identification* (PPID), což je identifikátor procesu rodiče. Výjimkou je první proces v systému, který je spuštěn při inicializaci jádra. Tento proces se nazývá – `init` a má `PID = 1`. Je rodičem všech procesů v systému.[23]

Na obrázku 1.1 je znázorněn příklad životního cyklu procesu `/bin/bash` s použitím příkazu `/bin/ls`. Tento příklad lze popsat pomocí dalších kroků:



Obr. 1.1: Životní cyklus procesů na příkladě `/bin/ls`

- Proces `/bin/bash` se klonuje pomocí systémového volání `fork()`
- Tím se vytvoří klon `/bin/bash` s novým `PID` a `PPID` rovným `PID` rodiče
- Klon provede systémové volání `exec` ukazující na spustitelný soubor `/bin/ls` a nahradí svůj kód kódem spustitelného souboru (rodičovský proces pak čeká na dokončení potomka – `wait()`
- Pokud z nějakého důvodu potomek dokončil svou práci a nadřazený proces o tom nedostal signál, pak tento proces (`PID = 101`) neuvolní obsazené místo ve struktuře jádra a stav procesu se stane `zombie`

Tento příklad znázorňuje pouze část možných stavů procesů v operačním systému Linux. Podrobnější popis stavu procesů a jejich komunikace mezi sebou, je v následující částí kapitoly.

1.3 Stavy procesů

Každý běžící proces je v kteroukoli dobu v jednom z následujících stavů:

- **Aktivní (R)** – proces je ve frontě na spuštění, to znamená, že buď právě běží, nebo čeká, až mu bude přiděleno další kvantum času procesoru.
- **„Spící“ (S)** – proces je ve stavu přerušitelného čekání, to znamená, že čeká na nějakou událost, signál nebo uvolnění požadovaného zdroje

- Je ve stavu **nepřerušovaného čekání (D)** – proces čeká na určitý („přímý“) signál z hardwaru a na ostatní signály nereaguje
- **Pozastaven (T)** – proces je v režimu trasování (obvykle tento stav nastává při ladění programů)
- **„Zombie“ (Z)** je proces, jehož provádění skončilo, ale struktury jádra, které s ním souvisí, nebyly z nějakého důvodu uvolněny. Jedním z důvodů jejich výskytu v systému může být následující situace. Struktury jádra specifické pro procesy jsou typicky uvolněny nadřazeným procesem poté, co obdrží signál ukončení od potomka. Ale jsou případy, kdy rodičovský proces skončí dříve než potomek. Procesy, které nemají rodiče, se nazývají „sirotci“. Sirotci jsou automaticky přijati procesem `init`, který přijímá signály o jejich dokončení. Pokud nadřazený proces nebo `init` z nějakého důvodu nemůže přijmout signál o ukončení podřízeného procesu, pak se podřízený proces promění v „zombie“. Zombie procesy nezabírají čas procesoru, ale jejich odpovídající struktury jádra nejsou uvolněny.[15]

Také existuje zvláštní typ procesů – démony. Tento typ procesu běží na pozadí, jako služby ve Windows, bez terminálu a provádí úkoly pro jiné procesy. Tento typ procesů na serverových systémech je hlavní.

Protože ve většině případů jsou démoni v Linuxu nečinní žádnou akce a čekají na příchod jakýchkoli dat, respektive jsou potřeba relativně zřídka, takže udržovat je neustále načtené v paměti a zahlcovat tím systémové prostředky je iracionální. Pro organizaci práce démonů byl vynalezen démon `inetd` nebo jeho bezpečnější modifikace `xinetd` (*eXtended Internet Daemon*). Mezi hlavní funkce `inetd` resp. `xinetd` patří:

- Nastavení limitu počtu démonů ke spuštění
- Monitorování připojení na konkrétních portech a zpracovávat příchozí požadavky
- Omezení přístupu ke službám založeným na *Access control list* (ACL)[15]

1.4 Meziprocesové interakce

Všechny procesy v systému, ať už jde o Linux nebo jiný OS, si mezi sebou vyměňují nějaké informace. V Linux existuje několik typů nástrojů meziprocesové komunikace *Interprocess Communication* (IPC), které lze rozdělit do několika úrovní:

- Lokální
- Vzdálený

1.4.1 Lokální meziprocesová komunikace

Tento typ komunikace je navázány na procesor a je možný pouze v rámci jednoho počítače. Existuje dva druhy lokální komunikace dvou procesů, první je kanál, druhý `signal`. Pomocí kanálů lze předávat informace mezi procesy jako pomocí konvertoru. Například v jednom okně terminálu lze vytvořit kanál příkazem `mkfifo` a přiřadit novému kanálu nějakou akci, která bude provedena při přijetí dat do kanálu. Podle příkladu ve výpisu 1.1 do nového kanálu `pipe` je přiřazená akce archivování dat pomocí `gzip`.

Výpis 1.1: Vytvoření nového kanálu pro gzip

```
[root@ubuntu]:~$ mkfifo pipe
[root@ubuntu]:~$ gzip -9 -c < pipe > out
```

V jiném okně terminálu, pak je možné odkázat na vytvořený kanál a předat data. Tento příklad je znázorněn ve výpisu 1.2 což způsobí archivace obsahu souborů `test.txt` pomocí programu `gzip`.

Výpis 1.2: Přesměrování obsahu souboru do kanálu

```
[root@ubuntu]:~$ cat test.txt > pipe
```

Meziprocesová komunikace pomocí signálů znamená oznamování procesu o nějaké události.Když je procesu odeslán signál, operační systém proces přeruší. Pokud má proces nainstalovaný vlastní obráběč signálu, operační systém ho spustí a předá mu informace o signálu. Pokud proces nenastavil obráběč, provede se výchozí zpracování signálu systémem. Všechny signály mají prefix `SIG` a mají číselné shody definované v hlavičkovém souboru `signal.h`. Signály lze odesílat následujícími způsoby:

- z terminálu stisknutím speciálních kláves nebo kombinací, například stisknutím Ctrl-C se vygeneruje signál `SIGINT`.
- jádrem systému: když nastanou hardwarové výjimky nebo chybná systémová volání
- jeden proces druhému pomocí systémového volání `/bin/kill`

1.4.2 Vzdálená meziprocesová komunikace

Do vzdálené meziprocesové komunikace patří vzdálené volání procedur nebo *Remote Procedure Call* (RPC) a Unixové sokety. [15]

RPC je typ technologie, která umožňuje počítačovým programům volat funkce nebo procedury v jiném adresovém prostoru, obvykle na vzdálených počítačích. Implementace technologie RPC obvykle zahrnuje dvě součásti: síťový protokol *Transmission Control Protocol* (TCP) nebo *User Datagram Protocol* (UDP) pomocí kterého se

vymění data, příkladem může být klient-server řešení a jazyk pro serializaci objektů nebo struktury dat RPC.

Unixové sokety jsou v podstatě virtuální objekty, které existují, pokud na něj odkazuje alespoň jeden z procesů. Existuje dva typy soketu: místní (lokální) a síťové. Při použití lokálního socketu se mu přiřadí UNIXová adresa a na dané cestě se vytvoří speciální soubor, socket soubor, přes který mohou komunikovat libovolné lokální procesy pomocí operace čtení/zápis z něj. Při použití síťového soketu se vytvoří abstraktní objekt vázaný na naslouchací port operačního systému a síťové rozhraní, je mu přidělena `INET` adresa, která má adresu rozhraní a naslouchací port.

1.5 Správa procesů

Další část této kapitoly se věnuje způsobům získání informace o procesu a možnostem tento proces ovládat.

1.5.1 Získání informací o procesu

Linux má pseudosouborový systém `procfs`, který je u většiny distribucí připojen ke sdílenému souborovému systému v adresáři `/proc`. Tento souborový systém nemá žádné fyzické umístění, žádné *Hardware* (HW) zařízení, jako je pevný disk. Veškeré informace uložené v tomto adresáři jsou umístěny v operační paměti počítače, řízené jádrem OS a nejsou určeny k ukládání uživatelských souborů. Tento souborový systém obsahuje dostatek informací, aby zjistit o procesech v systému jako celku. Pro prohlížení informace o procesu existuje dva programy v Linux: `ps` a `top`.[26]

Aby získat seznam všech procesů, stačí použít příkaz který je uveden ve výpisu 1.3:

Výpis 1.3: Příklad získání seznamu procesů

```
[root@ubuntu]:~$ ps aux
```

V následujícím výpisu lze vidět příklad seznamu procesů:

Výpis 1.4: Příklad seznamu procesů pomocí příkazu ps

```
USER     PID %CPU %MEM    VSZ   RSS TTY    STAT START   TIME
root       1  0.0  0.0   8948   396 ?      Ssl  11:39   0:00
root      53  0.0  0.0   9304   236 tty1   Ss   11:39   0:00
denys     54  0.3  0.0  18096  3604 tty1   S    11:39   0:00
root     119  0.0  0.0  19180  2844 tty1   S    11:40   0:00
root     120  0.0  0.0  18032  2028 tty1   S    11:40   0:00
root     121  0.1  0.0  17008  2384 tty1   S    11:40   0:00
root     128  0.0  0.0  18660  1884 tty1   R    11:40   0:00
```

Výše uvedený výpis obsahuje hodně užitečné informace, patří sem následující:

- **PID** – identifikátor procesu, lze vidět proces s `PID = 1`, což je inicializační proces v systému který byl popsán v kapitole *Životní cyklus procesu*
- **%CPU** – je procento času procesoru přiděleného procesu
- **%MEM** – je procento využité operační paměti
- **VSZ** – je velikost paměti přiřazené pro tento proces
- **RSS** – je velikost paměti kterou skutečně využívá proces
- **TTY** – ovládací terminál
- **STAT** – stav procesu
- **START** – čas zahájení
- **TIME** – doba provádění na procesoru

Příkaz `ps` pořídí snímek aktuálních procesů. Naproti tomu příkaz `top` dynamicky zobrazuje stav procesů a jejich aktivitu v reálném čase.[26]

Výpis 1.5: Příklad seznamu procesů pomocí příkazu top

```
top - 12:04:01 up 24 min, 0 users, load average: 0.52
Tasks: 7 total, 1 running, 6 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.9 us, 0.7 sy, 0.0 ni, 98.4 id, 0.0 wa, 0.0 hi
MiB Mem : 16314.0 total, 9813.8 free,  6276.2 used
MiB Swap: 16384.0 total, 16254.2 free, 29.8 used.

PID USER    PR  NI    VIRT    RES    SHR S  %CPU  %MEM   TIME
  1 root    20   0    8948    396    324 S   0.0   0.0   0:00
 53 root    20   0    9304    236    180 S   0.0   0.0   0:00
 54 denys   20   0   18096   3604   3500 S   0.0   0.0   0:00
119 root    20   0   19180   2844   2744 S   0.0   0.0   0:00
120 root    20   0   18032   2028   2004 S   0.0   0.0   0:00
121 root    20   0   17008   2384   2296 S   0.0   0.0   0:00
129 root    20   0   18920   2140   1520 R   0.0   0.0   0:00
```

Jak lze vidět z výpisu 1.5 program top uvádí více užitečné informace o procesech a využité kapacitě systému. Tak lze vidět přehled o počtu běžících a „spících“ procesů, využití operační paměti a procesoru systému.

1.5.2 Ovládání procesu

V operačním systému Linux, každému procesu je při spuštění přiřazena specifická priorita, která má hodnotu mezi -20 a +20, kde +20 je nejnižší. Priorita nového procesu se rovná prioritě nadřazeného procesu. Pro změnu priority běžícího programu existuje utilita `nice`. Příklad použití tohoto programu je znázorněn ve výpisu 1.6:

Výpis 1.6: Příklad změny priority procesu

```
[root@ubuntu]:~$ nice [- adnice] command [args]
```

Parametr `adnice` je hodnota -20 až +19, která se přidává k hodnotě `nice` nadřazeného procesu. Záporné hodnoty může nastavit pouze superuživatel. Pokud není zadán parametr `adnice`, výchozí hodnota podřízeného procesu je o 10 vyšší než hodnota `nice` nadřazeného procesu. Pro všechny procesy spuštěné uživatelem je výchozí hodnota priority nula.[26]

Příkaz `renice` se používá ke změně hodnoty `nice` pro již běžící procesy. Superuživatel může změnit prioritu libovolného procesu v systému. Ostatní uživatelé mohou změnit hodnotu priority pouze u těch procesů, jejichž vlastníkem je tento uživatel. Dalším způsobem ovládání procesů je program `kill`, který posílá určitý signál na běžící proces, příklad použití programu `kill` je uveden ve výpisu 1.7:

Výpis 1.7: Příklad odeslání signálu

```
[root@ubuntu]:~$ kill [-SIG] PID [PID]
```

Parametr `SIG` je číslo signálu nebo název signálu a pokud tento parametr nebude zadán, bude odeslán výchozí signál 15 což je `SIGTERM` – ukončení procesu. Existuje podobný signál s identifikátorem 9 – `KILL`, pomocí kterého může super uživatel okamžitě ukončit jakýkoli proces. Tento signál oproti `SIGTERM` funguje trochu jinak a okamžitě ukončí proces a nedává mu čas na správné uložení všech zpracovaných dat.

Dva signály – 9 (`KILL`) a 19 (`STOP`) – jsou systémem vždy zpracovávány. První je potřeba, aby se proces definitivně zabil. Signál `STOP` pozastaví proces: v tomto stavu není proces odstraněn z tabulky procesů, ale není vykonán, dokud nepřijme signál 18 (`CONT`) – po kterém pokračuje v práci.[26]

2 Přehled technologie Spring Boot, Kotlin, Flask a Docker

Kapitola obsahuje základní popis frameworku Spring a Flask pro účely vývoje API, zejména s použitím programovacích jazyků Kotlin a Python. Další část kapitoly se věnuje technologii Docker pro izolaci aplikací do kontejnerů.

2.1 Spring Boot a Kotlin

2.1.1 Úvod do Spring

Existuje více pojmů spojených se Spring, například Spring framework a Spring Boot. Spring framework je odlehčený aplikační framework který poskytuje podporu pro další různé frameworky jako jsou *Jakarta Server Pages* (JSP), Hibernate atd. První verzi frameworku Spring napsal Rod Johnson v roce 2002 a framework byl poprvé vydán v roce 2003 pod licencí Apache verze 2.0. Spring framework poskytuje komplexní podporu infrastruktury pro vývoj Java aplikací. Navíc framework obsahuje v sobě celou řadu připravených modulů pro zjednodušení vývoje aplikace, jako například *Java Database Connectivity* (JDBC) pro práce s databázovým serverem, Spring Security pro snadnější konfigurace zabezpečení aplikace a nebo Spring Test pro vývoj testovacích prvků aplikace. Tyto a další moduly mohou výrazně zkrátit dobu vývoje aplikace.

Spring Boot je v podstatě rozšířením frameworku Spring, které eliminuje standardní konfigurace potřebné pro nastavení aplikace Spring. Spring Boot nabízí programátorům možnost automatické konfigurace Spring aplikace, například předem připravenou konfiguraci a vložený Tomcat server, automatické metriky a logování a další.[2]

2.1.2 Úvod do Kotlin

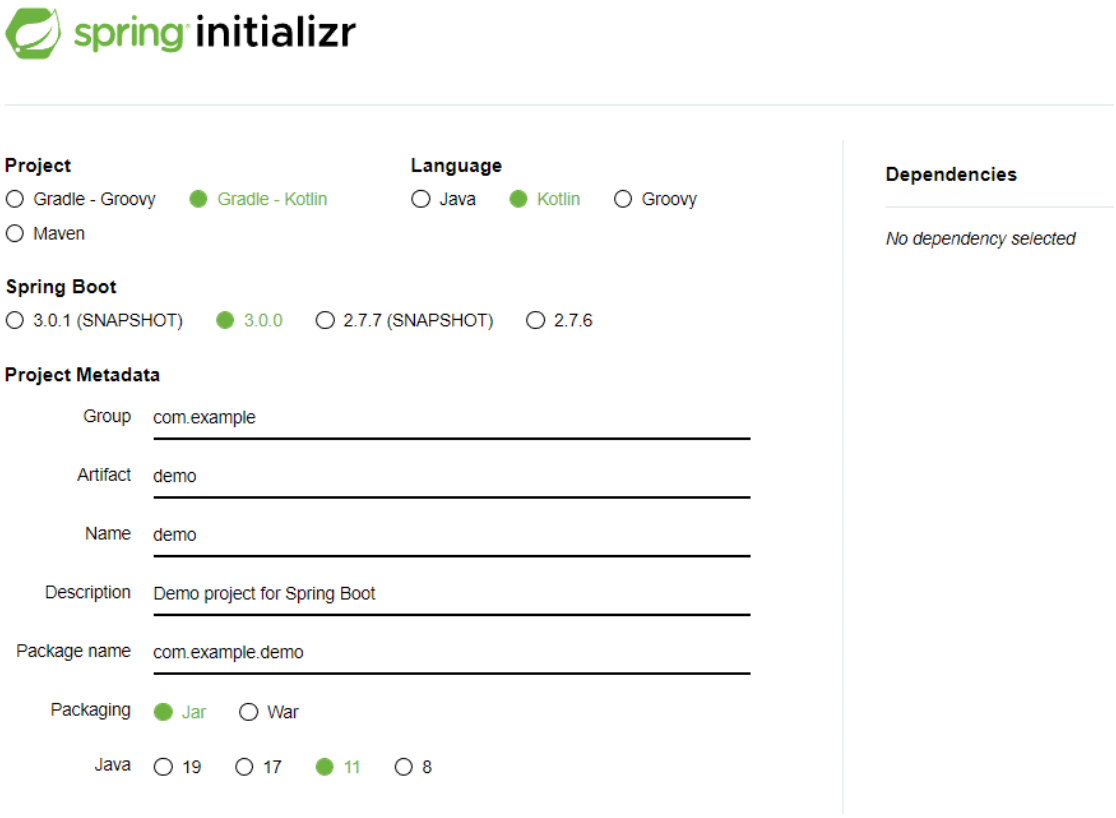
Kotlin je programovací jazyk vytvořený společností JetBrains. Byl vyvinut v roce 2011, aby nahradil Javu. Kotlin je zároveň plně kompatibilní s Javou, protože běží na jejím virtuálním stroji – *Java Virtual Machine* (JVM).[5]

Kotlin je staticky typovaný objektově orientovaný jazyk. **Objektově orientované jazyky** jsou jazyky, ve kterých probíhají všechny operace s objekty – bloky kódu. Objektem může být jakákoli entita s určitou sadou vlastností. Všechny objekty jsou navrženy podle speciálních vzorů nazývaných třídy. **Staticky typovaný jazyk** znamená, že typy proměnných nastavuje vývojář před spuštěním programu. Pokud

proměnná je deklarována jako celočíselná, už do ní nelze vložit text – překladač okamžitě oznámí chybu.[5]

2.1.3 Integrace Spring Boot a Kotlin

Díky tomu, že Spring Boot má podporu programovacího jazyku Kotlin, pro vytvoření programu na Spring Boot a Kotlin lze použít webové rozhraní `spring initializr`, příklad vytvoření projektu je znázorněn na obrázku 2.1.



Obr. 2.1: Příklad použití spring initializr

`Spring initializr` kromě vyplnění informace o projektu, umožní taky vybrat typ projektu, programovací jazyk, verze Spring Boot a přidat další knihovny. Výsledkem pak je hotový ukázkový projekt, který lze naimportovat do vývojového prostředí.

Základní struktura Gradle projektu se zklada z složky zdrojových kódů `src`, `build.gradle` a `settings.gradle` souboru, kde lze specifikovat nastavení projektu, zapojit doplňky a externí knihovny. Při použití projektu typu Maven, podobné nastavení jsou uvedené v souboru `pom.xml`. Ovšem pro programovací jazyk Kotlin se nejčastěji používá Gradle.

Příklad API v Kotlin a Spring

API – popis toho, jak jeden počítačový program interaguje s ostatními. V případě Spring Boot a Kotlin jedná se o popis *Hypertext Transfer Protocol* (HTTP) rozhraní, přes které server přijímá požadavek a vrátí výsledek zpracování požadavků.

Výpis 2.1: Příklad HelloWorld API v Kotlin a Spring

```
@RestController
@RequestMapping("/hello")
class HelloController {

  @GetMapping("/world")
  fun sayHelloWorld(request: HttpServletRequest): String {
    return "Hello␣World!"
  }
```

Ve výpisu 2.1 lze vidět jednoduché API v programovacím jazyce Kotlin s použitím frameworku Spring. Použitá anotace `@RestController` říká Springu, že třída `HelloController` je kontrolérem, který může obsahovat v sobě různé endpointy, které se definují pomocí anotace mapování, v příkladě výše je to `@GetMapping`.

Uniform Resource Locator (URL) cesta rozhraní se definuje cestou kontroléru, která je uvedena jako parametr anotace `@RequestMapping` a cestou funkce, uvedenou pomocí anotace mapování. Podle příkladu, ve výpisu 2.1 lze vyčíst jeden endpoint který přijímá HTTP GET požadavek na URL adrese `/hello/world` a vrátí data typu `String` s obsahem: *Hello World!*

Důležitou součástí funkce `sayHelloWorld` je vstupní parametr `request` typu `HttpServletRequest` který je naplněn frameworkem Spring. Tento parametr umožní získat další informace z příchozího požadavku, jako například hlavičky HTTP požadavků nebo informace o relaci s uživatelem (*session*).[25]

Výše uvedený příklad, je pouze jednoduchou ukázkou jak vytvořit API rozhraní pomocí Spring a Kotlin. Framework Spring nabízí další typy mapování, mezi nejčastěji používané patří:

- GetMapping – odpovídá HTTP GET požadavku
- PostMapping – odpovídá HTTP POST požadavku
- PutMapping – odpovídá HTTP PUT požadavku
- DeleteMapping – odpovídá HTTP DELETE požadavku
- PatchMapping – odpovídá HTTP PATCH požadavku

Poslední součástí definice API, je typ odpovědi požadavků. Ve výchozím nastavení Spring automaticky převede výsledek volání funkce do formátu `application/json` což je obyčejný Json objekt. V případě, kdy funkce vrátí instance třídy, tento objekt

bude přepsán do Json objektu, kde název položky v Json odpovídá názvu proměnné v instanci třídy. Spring taky podporuje vice typu návratové hodnoty, může to být nejen Json objekt ale například soubor.[8]

2.2 Flask framework

Flask je mikroframework pro vytvoření jednoduchého a rychlého projektu v programovacím jazyce Python s možností škálování na složité aplikace. Koncept microframework znamená, že v mikroframeworku není žádná sada nástrojů a knihoven, programátor si je může nainstalovat sám podle potřeby. Výhoda takové platformy je v tom, že obsahuje minimum pro svoji funkčnost, tedy není zde nadbytečné věci, které můžou zpomalit celé aplikační řešení. Flask umožňuje rychle vytvořit webovou aplikaci pomocí pouze jednoho souboru Python. Microframework lze použít k vývoji jak testovacích projektů nebo malých webů, které nepotřebují složitý backend, tak i API a komplexních e-commerce projektů.[6]

2.2.1 Příklad API v Flask

Ve výpisu 2.2 uveden příklad jednoduchého API, který vrátí *HyperText Markup Language* (HTML) text *"Hello, World!"* po zavolání URL adresy `/hello`.

Výpis 2.2: Příklad HelloWorld API v Flask

```
from flask import Flask

app = Flask(__name__)

@app.route("/hello")
def helloWorld():
    return "<p>Hello, World!</p>"
```

Výše uvedený Python kód provádí import třídy Flask. Následně je potřeba vytvořit instance třídy pomocí konstruktoru, kde první argument je název modulu nebo aplikace. Dále, se používá dekorátor `route()` k tomu, aby Flask věděl, jaké URL by mělo spustit funkci `helloWorld`. Funkce vrátí zprávu, která se zobrazí v prohlížeči.

Ke spuštění aplikace lze příkaz použít `flask run` nebo `python -m flask`. Ale předtím je potřeba v terminálu nastavit, s jakou aplikací má pracovat, exportováním proměnné prostředí `FLASK_APP`. Příklad spuštění Flask aplikace je uveden ve výpisu 2.3.

Výpis 2.3: Příklad spuštění Flask aplikace

```
[ root@ubuntu ]:~ $ export FLASK_APP=helloWorld
[ root@ubuntu ]:~ $ flask run
```

Výše uvedeným způsobem se spustí aplikace pouze lokálně, tedy nebude přístupná v síti, je to výchozí chování Flask. Aby API bylo veřejně dostupné v síti, je potřeba specifikovat IP adresu, příklad je uveden ve výpisu 2.4.

Výpis 2.4: Příklad spuštění Flask aplikace

```
[ root@ubuntu ]:~ $ flask run --host=0.0.0.0
```

Pro své fungování Flask Framework používá Jinja2, aplikací pro zpracování šablon, a Werkzeug, nástroj pro práci s WSGI, standard pro interakci mezi programem Python, který běží na straně serveru a samotným webovým serverem. Python používá modul Virtualenv k vytvoření izolovaného prostředí pro Flask aplikace. Vzhledem k tomu, že Flask je microframework a obsahuje jen základní funkcionalitu, používá se různé knihovny pro rozšíření funkcionality. Jedním z nejčastěji používaných Python knihoven spolu s frameworkem Flask je Socket.IO. Jedná se o přenosový protokol, který umožňuje obousměrnou komunikaci založenou na událostech v reálném čase mezi klienty a serverem. Oficiální implementace klientských a serverových komponent jsou napsány v JavaScriptu.[4]

2.3 Docker

Technologie kontejnerizace aplikací jsou široce používány v oblastech vývoje softwaru a analýzy dat. Tyto technologie pomáhají zvýšit bezpečnost aplikací, usnadňují jejich nasazení a zlepšují jejich škálovatelnost. Docker je platforma pro vývoj, nasazení a spouštění aplikací v kontejnerech. Docker je engine, na kterém běží virtuální operační systém, který je extrémně lehký, obsahuje v sobě jen nejdůležitější části systému a funguje na základě konfiguračních souborů `Dockerfile`.

Soubor `Dockerfile` obsahuje sadu pokynů, které bude Docker dodržovat při vytváření bitové kopie kontejneru. Tento soubor obsahuje popis základního stavu virtuálního prostředí, který bude zdrojovou vrstvou cílového `image`. `Docker image` je neměnná šablona, která se používá k vytvoření identických kontejnerů. Docker image obsahuje popis základního operačního systému, kód aplikace, externí knihovny, které jsou potřebné pro fungování aplikace. To vše je uspořádáno do podoby jediné entity, na jejímž základě lze vytvořit kontejner. Kontejnerem se pak rozumí instance `Docker image`, který je definován v souboru `Dockerfile`. Je to už běžící prostředí, vytvořené na základě postupu v `Dockerfile`. [24]

Následující výpis 2.5 obsahuje příklad souboru `Dockerfile` pro spuštění Spring Boot aplikace popsané v kapitole 2.1.3 jako Docker kontejner.

Výpis 2.5: Příklad Dockerfile

```
FROM java:11
WORKDIR /
ADD HelloWorldAPI.jar HelloWorldAPI.jar
EXPOSE 8080
CMD java - jar HelloWorldAPI.jar
```

Výše uvedený výpis obsahuje definice pro `Dockerfile` který používá předem připravený jiný `image` java ve verzi 11. Jedná se o minimální distribuce Linuxu ve kterém je instalován balík OpenJDK 11, díky čemu, lze uvnitř spustit java aplikace. Každý řádek uvnitř `Dockerfile` definuje novou vrstvu výsledného `Docker image`. Tak, následující vrstva nastavuje pracovní adresář uvnitř kontejneru pomocí příkazu `WORKDIR`. Následně proběhne kopie `jar` souboru do `image` a nastaví se zveřejnění portu 8080 z kontejnerů do vnějšího světa. Posledním krokem je definice spustitelného příkazu, který proběhne až kontejner se nastartuje a tím se spustí HelloWorldAPI aplikace.

Dalším krokem je potřeba vytvořit `Docker image` podle výše uvedené konfigurace `Dockerfile`. Toho lze dosáhnout pomocí příkazu ve výpisu 2.6.

Výpis 2.6: Příklad vytvoření Docker image

```
[ root@ubuntu ]:~ $ docker build -t helloWorldImage .
```

Kde příkaz `docker build` obsahuje parametr `-t`, který definuje název výsledného `image` a následně cestu do souboru `Dockerfile`, v případě, že příkaz je spuštěn ve stejné složce, musí být zadána tečka.

Nyní, je možné vytvořený `helloWorldImage` spustit pomocí příkazu podle výpisu 2.7.

Výpis 2.7: Příklad spuštění kontejneru

```
[ root@ubuntu ]:~ $ docker run \
    --name helloWorldContainer helloWorldImage
```

Pro příkaz `docker run` je potřeba uvést název kontejneru a název `image` ze kterého bude vytvořen kontejner. Až kontejner bude vytvořen a úspěšně nastartován, lze ho vidět v seznamu běžících kontejneru, který lze získat pomocí příkazu ve výpisu 2.8.

Výpis 2.8: Příklad získání seznamu běžících kontejnerů

```
[ root@ubuntu ]:~ $ docker ps -a
```

2.3.1 Stav procesů uvnitř Docker kontejneru

Docker kontejner je obdoba virtuálnímu stroji, díky tomu, lze se připojit dovnitř kontejneru a provádět jakékoliv operace, v závislosti na použitém operačním systému uvnitř kontejnerů. Pro připojení do Docker kontejneru existuje příkaz `docker exec`, příklad jeho použití uveden ve výpisu 2.9.

Výpis 2.9: Příklad připojení do terminálu kontejneru

```
[ root@ubuntu ]:~ $ docker exec -it  \
    helloWorldContainer /bin/bash
```

Jako první parametr se zadává buď název kontejneru nebo jeho ID, další parametr je příkaz který Docker provede uvnitř kontejneru, díky tomu, že byl zadán argument `-it`, příkaz `/bin/bash` bude spuštěn v interaktivním módu, což umožňuje ovládat příkazový řádek uvnitř kontejneru.[24]

Pro získání seznamu běžících procesů a jejich ovládání lze používat příkazy uvedené v kapitole 1.5, pokud kontejner má v sobě operační systém Linux.

2.4 Správa procesů v OS Linux pomocí Kotlin

Programovací jazyk Kotlin, nabízí několik možností jak ovládat procesy v operačním systému, ve kterém běží Kotlin program. Pro zajištění přístupu k nativním službám operačního systému, obsahuje distribuce Kotlin pro nativní kompilátor sadu předem sestavených knihoven specifických pro každý cíl. Tyto knihovny se jmenují knihovny platformy (*platform libraries*). Pro všechny operační systémy založené na Unixu nebo Windows, včetně Android a iOS, Kotlin poskytuje platformu `POSIX lib`. Tato knihovna obsahuje implementaci standardu *Portable Operating System Interface* (POSIX) – jedná se o rozhraní které umožňuje spouštět shell scripty a příkazy do operačního systému.[5]

Před samotným použitím platformy `POSIX`, je potřeba tuto knihovnu naimportovat podle výpisu 2.10.

Výpis 2.10: Import knihovny POSIX

```
import platform.posix.*
```

Následně, je možné knihovnu použít například pro získání obsahu složky. Příklad použití uveden ve výpisu 2.11.

Výpis 2.11: Získání obsahu složky pomocí knihovny POSIX

```
platform.posix.system("ls")
```

Funkce `system`, ve výše uvedeném výpisu, vrátí instance třídy `Process`, který lze pak zpracovávat a číst návratové hodnoty procesu.

Existuje další možnost jak spravovat procesy pomocí Kotlin a to v případě, že program je kompilován pro Java platformu, tedy bude běžet v JVM. V takovém případě není potřeba importovat žádné knihovny ale lze využít třídu `Runtime`, pomocí které vytvořit nový proces v operačním systému. Příklad použití `Runtime` je uveden ve výpisu 2.12, stejně jako v případě knihovny `POSIX`, funkce `exec` vrátí instance třídy `Process` což umožňuje plnou kontrolu nad novým procesem.[5]

Výpis 2.12: Získání obsahu složky pomocí třídy Runtime

```
Runtime.getRuntime().exec("ls")
```

3 Standardy návrhu API

Tato kapitola se věnuje standardům navrhnu REST API, vysvětluje princip REST a popisuje kroky pro návrh kvalitního REST API rozhraní. V neposlední řadě v kapitole jsou probrány způsoby dokumentaci API, a existující metody číslování verzí pro REST API.

3.1 Úvod do REST

REST je zkratka pro *Representational State Transfer*, v podstatě to znamená prezentace dat ve formátu vhodném pro uživatele API. Toto je aktuálně jeden z nejpopulárnějších přístupu pro vytváření API. Pojem „REST“ byl vytvořen Royem Fieldingem v roce 2000, který byl také jedním z tvůrců protokolu HTTP. Model REST nezávisí na žádných základních protokolech a nevyžaduje vazbu na HTTP. Nejběžnější implementace REST API však používají HTTP jako aplikační protokol. Hlavní výhodou použití REST s HTTP je, že používá otevřené standardy a nevyžaduje specifickou implementaci API nebo klientských aplikací. Například webová služba REST může být napsána v Java a klientské aplikace mohou používat jakýkoli jazyk nebo sadu nástrojů, které umožňují vytvářet požadavky HTTP a analyzovat odpovědi.

Základní myšlenkou REST je, že každé volání služby přenese klientskou aplikaci do nového stavu. Ve skutečnosti REST není protokol nebo standard, ale přístup, architektonický styl designu API.[14]

Jak již bylo řečeno v kapitole 2.1.3, API v podstatě znamená soubor pravidel a mechanismů, pomocí kterých jedna aplikace nebo komponenta interaguje s ostatními. Přístup návrhu API REST ovšem není jediný, existují také jiné typy webového API: RPC v kombinací s *Extensible Markup Language* (XML), *Simple Object Access Protocol* (SOAP).[1]

Přístup RPC je velmi starý koncept, který kombinuje staré, střední a moderní protokoly, které umožňují volat metodu v jiné aplikaci. XML-RPC je protokol, který se objevil v roce 1998 krátce po vzniku XML. Původně byl podporován Microsoftem, ale brzy Microsoft úplně přešel na SOAP. Ale XML-RPC nadále žije v různých jazycích, jako například PHP.

SOAP se také objevil v roce 1998 a byl navržen společností Microsoft. Protokol se dále vyvíjel a získal desítky nových specifikací, až v roce 2003 W3C schválilo jako standard SOAP 1.2, které je nyní poslední.[20]

3.2 Vlastnosti REST API

Většina moderních webových aplikací poskytuje rozhraní API, které mohou klienti používat k interakci s aplikací. Kvalitně navržené API by mělo podporovat následující:

- **Nezávislost na platformě.** Každý klient by měl být schopen volat rozhraní API bez ohledu na to, jak je rozhraní API interně implementováno. To vyžaduje použití standardních protokolů a také mechanismus, pomocí kterého se klient a webové služby mohou dohodnout na formátu dat, která mají být vyměňována.
- **Zpětná kompatibilita a vývoj.** Webové rozhraní API musí být schopno vyvíjet a rozšiřovat svou sadu funkcí nezávisle na klientských aplikacích. Jak se rozhraní API vyvíjí, stávající klientské aplikace by měly nadále fungovat beze změn. Všechny funkce musí být dostupné, aby je klientské aplikace mohly plně využívat.[20]

Níže jsou uvedeny některé základní principy pro navrhování REST API pomocí protokolu HTTP:

- Aplikační řešení má být typu klient - server, jinak REST nemá smysl používat.
- Rozhraní REST API jsou vyvíjena kolem zdroje dat, což může být jakýkoli typ objektu, text, obrázek nebo služby, ke které má klient přístup.
- Jakýkoli zdroj dat musí mít unikátní identifikátor který jednoznačně identifikuje. Například číslo objednávky podle kterého lze načíst další zdroje dat – detail objednávky.
- Zdroje mohou být vzájemně propojeny – za tímto účelem se jako součást odpovědi přenáší buď ID, nebo jak se častěji doporučuje, odkaz.
- Server neukládá stav – to znamená, že server neodděluje jednou relací od druhé, neukládá všechny relace do paměti.[20]

3.3 Definování operací API z hlediska HTTP metod

Protokol HTTP definuje několik metod, které přiřazují sémantický význam požadavku. Níže jsou uvedeny nejběžnější metody HTTP používané většinou webových rozhraní API REST:

- **GET.** Vrátí reprezentaci zdroje na zadaném identifikátoru *Uniform Resource Identifier* (URI). Tělo zprávy odpovědi obsahuje informace o požadovaném zdroji.
- **POST.** Vytvoří nový prostředek na zadaném URI. Tělo požadavku obsahuje informace o novém zdroji. Metodu POST lze také použít k zahájení operací, které přímo nesouvisí s vytvářením dat.

- **PUT.** Vytvoří nebo nahradí data na zadaném URI. Tělo zprávy požadavku specifikuje vytvářený nebo aktualizovaný zdroj.
- **PATCH.** Provede částečnou aktualizaci zdroje. Tělo požadavku definuje sadu změn použitých na zdroj.
- **DELETE.** Odebere zdroj na zadaném URI.

Výsledek konkrétního dotazu by měl záviset na tom, zda je cílovým zdrojem kolekce nebo jeden prvek. Níže jsou uvedené příklady správně navrženého REST API:

- Vytvořit uživatele: POST /uzivatel.
- Aktualizace údajů uživatele s ID 1: PUT /uzivatel/1.
- Smazat uživatele s ID 1: DELETE /uzivatel/1
- Získat seznam všech uživatelů: GET /uzivatel
- Získat detail jednoho uživatele s ID 1: GET /uzivatel/1

3.4 Číslování verzí REST API

Správa verzí umožňuje webovému rozhraní API určit, které funkce a prostředky se mají vystavit, takže klientská aplikace může odesílat požadavky na konkrétní verzi funkce nebo prostředků. Existuje několik různých přístupů číslování verzí API s vlastními výhodami a nevýhodami.

První způsob je neuvádět číslo verze vůbec ale používat princip bezpečné zpětné kompatibility. Tento princip spočívá v tom, že při změně API se nebudou odstraňovat položky v odpovědi na požadavek. Tedy pokud existuje nějaké API které vrátí identifikátor objektu a například jeho název, pak lze bezpečně přidat nový atribut, protože uživatele API budou ho ignorovat bez nutnosti cokoliv měnit. Provádění významnějších změn v API jako například odstranění nebo přejmenování atributů nebo změna vztahů mezi zdroje však mohou být významnými změnami, které brání správnému fungování stávajících klientských aplikací.

Další způsob je řízení verze pomocí URI. Tento způsob předpokládá, že součástí URI bude atribut odpovídající za číslo verze, například `/v1` a `/v2`, kde API `/v2` může vrátit jiný typ objektů než ve verzi `/v1`. Tento mechanismus verzování je velmi jednoduchý, ale vyžaduje, aby server směroval požadavek na příslušný koncový bod. Tento přístup se však může stát příliš složitým, protože webové rozhraní API prochází několika iteracemi a server musí podporovat více různých verzí.

Verzování pomocí řetězce dotazu je dalším způsobem. Aby se vyhnout zadávání více identifikátorů URI, lze zadat verzi API pomocí parametru řetězce dotazu připojeného k požadavku HTTP, jako například `/uzivatele?verze=2`. Přitom výchozí hodnota parametru verze by měla mít smysl, například 1, pokud je vynechána staršími klientskými aplikacemi. Tento přístup má sémanticky výhodu, že stejný zdroj

je vždy vrácen pod stejným identifikátorem URI.

Posledním způsobem číslování verze je předání verze v hlavičce požadavku. Namísto přidání čísla verze jako parametru řetězce dotazu lze implementovat vlastní záhlaví, které určuje verzi API. Tento přístup vyžaduje, aby klientská aplikace přidala ke všem požadavkům příslušnou hlavičku. Kód na straně serveru, který zpracovává požadavek z klientské aplikace, však může použít výchozí hodnotu například verze 1, pokud je vynechána hlavička čísla verze. Příkladem implemntace může být hlavička `api-version=1`.

3.5 Open API

Společnost Open API byla vytvořena za účelem standardizace popisů REST API napříč dodavateli. V rámci této iniciativy byla specifikace Swagger 2.0 přejmenována na *Open API Specification* (OAS) a přesunuta do projektu Open API Initiative. Specifikace Open API přichází se sadou doporučení a pokynů pro vývoj REST API. Poskytuje řadu výhod v dokumentaci API, ale vyžaduje zvláštní péči při navrhování rozhraní API, aby bylo v souladu se specifikací. Open API doporučuje, jako první krok začít vytvořením popisu API, nikoli implementací. Pomocí Swaggeru a dalších nástrojů lze vytvářet klientské knihovny a dokumentaci na základě popisu API.

Popis REST API pomocí Swagger se představuje jako YAML soubor, podle kterého lze zjistit kompletní informace o API. Příklad Swagger dokumentace je uveden ve výpisu 3.1.

Podle uvedeného výpisu lze přečíst kompletní informace o `/pet/findByStatus` API. Lze vidět že se jedná o HTTP GET požadavek, který vrátí data ve formátu Json a přijímá jeden povinný URI parametr `status` který může nabývat hodnot, uvedených v poli `enum`.

Další součástí popisu API je definice odpovědi, tak lze zjistit, že v případě úspěšného zpracování požadavku, API vrátí objekt typu Json jehož struktura je definována v popisu `/definitions/Pet`, který musí být uveden v další části dokumentace. Důležitou součástí dokumentace Swagger je atribut `operationId` který musí mít unikátní hodnotu pro jednoznačné určení koncového bodu.

Výpis 3.1: Příklad dokumentace REST API pomocí Swagger

```
/pet/findByStatus:
    get:
      description: Finds Pets by status
      operationId: findPetsByStatus
      produces:
        - application/json
      parameters:
        - name: status
          in: query
          required: true
          type: string
          enum:
            - available
            - pending
            - sold
      responses:
        '200':
          description: successful operation
          schema:
            type: array
            items:
              $ref: '#/definitions/Pet'
        '400':
          description: Invalid status value
```

4 Přehled zranitelnosti Web API a způsobů zabezpečení

Kapitola se věnuje teoretickému úvodu do problematiky bezpečnosti Web API, zejména REST API, probírá nejvýznamnější zranitelnosti a způsoby zabezpečení API proti zneužití.

4.1 Úvod do zabezpečení API

Existuje více API rozhraní v závislosti na formátu dat, počtu poskytovaných koncových bodů a počtu uživatelů, metody útoku na API se budou velmi lišit v závislosti na těchto vlastnostech. Podle principu distribuce API lze rozdělit na další skupiny:

- Privátní - API se používají v rámci stejné síťové infrastruktury, nejsou nijak integrována se systémy třetích stran. Hlavním cílem je úplná kontrola jedné společnosti nad API.
- Partnerské – API jsou otevřená pro partnerské infrastruktury, nejsou nijak integrována s veřejnými uživateli. Obvykle se používá k integraci dvou nebo více systémů v rámci partnerských dohod nebo geograficky rozdělených kanceláří jedné společnosti.
- Externí – API používají vývojáři třetích stran, hlavně pro veřejnou integrace se systémy.

Historicky byla API původně používána vývojáři jenom v rámci jedné společností nebo partnerství, postupem času větší počet služeb začal zveřejňovat svoji API pro propojení do dalších systémů od jiných dodavatelů. Kvůli tomu začal růst zájem útočníků o nové bezpečnostní slabiny API. Mezi známé útoky API patří:

- Parametrový útok: Nejběžnější typ útoku zaměřený na manipulaci s deserializátorem dat cílového webového aplikačního serveru.
- Útok zveřejněním dat (*exposure data attack*): samozřejmě kompromitace dat a jejich další využití, ať už se zlým nebo dobrým úmyslem, je také jedním z vektorů útoků.
- Útoky *Man In The Middle* (MITM): útok uprostřed mezi serverem API a klientem.
- Zneužívání aplikací: princip útoku spočívá v takovém zneužití API aby následně narušit obchodní proces.

Výše uvedené útoky API jsou pouze běžně známé, ovšem existuje standardizace útoků, které pomáhají sestavit seznam požadavků na zabezpečení API. Jedním z takových standardů je *Open Web Application Security Project* (OWASP) který obsahuje seznamy rizik v různých softwarových technologiích. Tento standard definuje

deset nejnebezpečnějších zranitelnosti API, se kterými je potřeba počítat bezem vývoje, mezi nimi patří následující:

- Slabiny v řízení přístupu k objektům (*Broken Object Level Authorization*).
- Slabiny v autentizaci uživatele (*Broken User Authentication*).
- Zpřístupnění důvěrných údajů (*Excessive Data Exposure*).
- Nedostatek kontroly a omezení (*Lack of Resources & Rate Limiting*).
- Slabiny v řízení přístupu na funkční úrovni (*Broken Function Level Authorization*).
- Nezabezpečená deserializace (*Mass Assignment*).
- Nesprávné nastavení zabezpečení (*Security Misconfiguration*).
- SQL, NoSQL, Command injekce (*Injection*).
- Slabiny ve správě API (*Improper Assets Management*).
- Slabiny v logování a monitorování (*Insufficient Logging & Monitoring*).

V další části kapitoly jsou probrané výše uvedené zranitelnosti spolu se způsobem zabezpečení proti útokům.[18]

4.2 Slabiny v řízení přístupu k objektům

Tento typ zranitelnosti API má ještě další název – „Nebezpečné přímé odkazy na objekt“ (*Insecure Direct Object References*). Jedná se o jeden z nejběžnějších problémů s API který spočívá v tom, že přístup ke každému objektu pomocí API má být řízen kontrolou oprávnění uživatele, který tento API vyvolal. Například, existuje několik API pro správu uživatelů, podle výpisu 4.1 jsou to API pro načtení všech uživatelů, načtení detailů pouze jednoho uživatele a API pro smazání uživatele podle identifikátoru.[18]

Výpis 4.1: Příklad API pro správu uživatelů

```
GET /uzivatel
GET /uzivatel/1
DELETE /uzivatel/1
```

V případě návrhu výše uvedeného API, může nastat problém, kdy uživatel s ID = 1 zkusí změnit identifikátor na náhodou hodnotu a zavolá API pro smazání jiného uživatele, tedy například s ID = 5. Aby zabránit takovému typu útoku, lze vylepšit implementaci serveru následujícím způsobem:

- Přidat ověření oprávnění k objektům při každém požadavku.
- Zkontrolovat, že přihlášený uživatel má přístup pouze k povoleným objektům.
- ID objektů by měla být komplexní, aby znesnadnit jejich odhadnutí, například ve formě *Universally Unique Identifier* (UUID), a nikoli jednoduchá sekvence.[13]

4.3 Slabiny v autentizaci uživatele

Autentizační mechanismy jsou často implementovány nesprávně, což útočníkům umožňuje kompromitovat autentizační tokeny nebo zneužít implementační chyby k dočasnému nebo trvalému převzetí identity jiného uživatele. Mezi existující způsoby autentizace patří následující[22]:

- **API key** – je znakový řetězec, který klient odesílá spolu s požadavkem na server. Pro úspěšné ověření musí být řetězec na klientovi a serveru shodný. Toto schéma poskytuje ochranu proti neoprávněnému použití API a umožňuje například kontrolu limitů používání API.
- **Basic Authentication** – základní autentizace používá dva parametry pro ověřování, jako například uživatelské jméno a heslo. K přenosu informací se používá hlavička HTTP `Authorization` s klíčovým slovem `Basic` následovaným mezerou a řetězcem `jmeno:heslo` zakódovaným v `base64` formátu.
- **Cookie-Based Authentication** – autentizace založená na datech cookie využívá mechanismus předávání cookie v požadavcích HTTP. V odpovědi na požadavek klienta, server odešle hlavičku `Set-Cookie`, která obsahuje položku `JSESSIONID`. Tento parametr následně bude se odesílat na server při každém požadavku, podle čeho server bude schopen identifikovat uživatele a ověřit přístup. Pro správné fungování je zapotřebí ukládat hodnotu tohoto parametru na serveru, pro každého uživatele.
- **Token-Based Authentication** – používá serverem podepsaný token (`bearer token`), který klient odešle na server v autorizační HTTP hlavičce s klíčovým slovem `Bearer` nebo v těle požadavku. Při příjmu tokenu musí server zkontrolovat jeho platnost - zda uživatel existuje, neuplynula doba použití atd. Token lze použít jako součást protokolů OAuth 2.0 nebo OpenID Connect, nebo může server vygenerovat token sám. Jako jedním ze způsobů zjednodušení systému lze použít autorizační autoritu třetí strany, například umožnit uživatelovi přihlásit se pomocí Google, tím pádem není potřeba ukládat na serveru přihlašovací údaje uživatele.

Pro výše uvedené způsoby autentizaci uživatele existují dva dílčí problémy[22]:

- Nedostatek ochranných mechanismů. S koncovými body API, které jsou odpovědné za autentizaci, se musí zacházet jinak než s běžnými koncovými body a server musí implementovat další vrstvy ochrany.
- Nesprávná implementace mechanismu. Mechanismus je používán nebo implementován bez ohledu na útočné vektory, nebo jde o nesprávný případ použití.

4.4 Zpřístupnění důvěrných údajů

Útok spočívá v tom, že server poskytuje redundantní data v odpovědi na dotaz. Je to z toho důvodu, že server pro většinu příchozích požadavků provádí dotaz do databáze a výsledek vrátí v odpovědi API bez toho, aby provedl filtrování citlivých dat. Následně útočník může využít tyto údaje pro své účely. Problém se zhorší, pokud redundantních dat je příliš hodně. Při velkém zatížení to povede k problémům se sítí.[18]

Pro zabezpečení API je potřeba při vývoji nespoléhat na filtrování dat v klientovi, všechna data musí být filtrována na serveru.

4.5 Nedostatek kontroly a omezení

Jedná se o útok autorizace API hrubou silou. Pro zabezpečení proti takovému typu útoku, server by měl implementovat následující omezení:

- Omezit počet neúspěšných pokusů o autorizaci pro stejného uživatele. Případně použít reCapture nebo podobný mechanismus.
- Blokovat zdrojovou IP adresu pokud počet neúspěšných pokusů z ní překročí určitou hodnotu pro všechny uživatele.

Pro programovací jazyk JavaScript existují nástroje, které umožňují provádět takové kontroly automaticky, například Rate limiter, a okamžitě odeslat odpověď „*429 Too Many Requests*“ bez zatížení serveru. Dalším obdobným typem útoku je *Denial-of-Service* (DoS) útok. Pro zabezpečení, server by měl omezovat počet příchozích požadavků od jednoho uživatele na stejnou API adresu v průběhu krátké časové doby.

Dalším aspektem je zabezpečení serverů proti zneužití parametrů v API tak, že útočník zadá příliš velkou hodnotu. Tedy server by mel validovat všechny příchozí parametry podle nějakého pravidla, buď minimální a maximální hodnoty, pokud se jedná o číslo, nebo pomocí regulárních výrazů v případě textového pole.[13]

4.6 Slabiny v řízení přístupu na funkční úrovni

Složité zásady řízení přístupu s různými hierarchiemi, skupinami a rolemi a nejasné oddělení mezi administrativními a běžnými funkcemi mají tendenci vést k autorizačním chybám. Využitím těchto problémů získají útočníci přístup ke zdrojům jiných uživatelů nebo administrativním funkcím. Měl by být vytvořen jasný systém rozlišování přístupu mezi rolemi uživatelů API. Například existuje dvě skupiny roli: běžný uživatel a administrátor. Příkaz pro zobrazení všech uživatelů může volat pouze administrátor. Tak, při každém vyvolání příkazu je vyžadována kontrola přístupu,

aby běžný uživatel nemohl příkaz vyvolat pouze změnou formátu. Navíc, administrační API se dá oddělit od uživatelských tak, že se přidá další úroveň v cestě API, například `/administrace/uzivatele`.[18]

4.7 Nezabezpečená deserializace

V tomto případě přenášena data na server útočníkem, mají za účel neoprávněně změnit hodnotu. Například, detail uživatele se skládá, podle výpisu 4.2, z položek: jméno, věk a známka.

Výpis 4.2: Příklad odpovědi API pro detail uživatele

```
{"jmeno":"Jan","vek":18,"znamka":"D"}
```

Uživatel má oprávnění pomocí POST metody změnit svůj věk. Ale útočník může využít POST metodu a místo změny věku, pošle změnu známky na jinou hodnotu. Kvůli tomu, že server nekontroluje zdroj požadavku a automaticky provádí deserializaci vstupního Json, požadavek bude úspěšně zpracován.

Aby zabránit tomuto útoku, server musí omezit seznam atributů, které může uživatel změnit a neprovádět automatickou deserializaci celého vstupního Json ale jenom položek, které jsou povolené pro editace.[13]

4.8 Nesprávné nastavení zabezpečení

Chybná konfigurace zabezpečení je obvykle důsledkem nezabezpečených výchozích konfigurací, neúplných nebo ad-hoc konfigurací, otevřeného cloudového úložiště, nesprávně nakonfigurovaných hlaviček HTTP, zbytečných metod HTTP, povolného sdílení zdrojů *Cross Origin Resource Sharing* (CORS) a podrobných chybových zpráv obsahujících citlivé informace. Mezi nejčastější problémy chybně bezpečnostní konfigurace patří:

- Použití výchozího nastavení aplikací, které nemusí být bezpečné.
- Použití veřejného datového úložiště.
- Do internetu se chybně dostala citlivá informace, jako je konfigurace systému nebo nastavení přístupu.
- Regulární výrazy jsou použity nesprávně, což umožňuje *Regular expression Denial of Service* (ReDoS) útok.
- Špatně nakonfigurované hlavičky HTTP.
- Autentizační údaje jako například jemno a heslo, token, API klíč jsou odesílány v URL. To není bezpečné, protože parametry URL mohou zůstat v log souborech webového serveru.

- Chybějící nebo nesprávně použitá CORS.
- Nepoužití HTTPS, chybějící nastavení certifikátů.
- Při provozu produkčního systému se používají nastavení, která jsou určena pro vývoj a ladění.
- Chybové zprávy obsahují citlivé informace, například `stack trace`.
- Na serveru jsou otevřené zbytečně porty.
- Použití zastaralého operačního systému nebo aplikace.

Pro zvýšení bezpečnosti celého systému a API je potřeba vyhnout se výše uvedeným chybám.[13]

4.9 SQL, NoSQL, Command injekce

Injekce je provádění programového kódu, který není poskytován systémem. Existuje dva druhy injekce: *Structured Query Language* (SQL) příkazů a příkazy operačního systému. Útok bude úspěšný, pokud server provede přijaté příkazy bez kontroly. Injekce SQL může vést k neoprávněnému přístupu k datům. Pomocí injekce příkazů operačního systému může útočník získat přístup k serveru. V takovém případě, pokud například API podle jména uživatele vytváří nějakou složku na souborovém systému pomocí příkazu `mkdir`, útočník může upravit hodnotu tak, aby došlo ke smazání celého souborového systému. [18]

Aby zabránit takovému typu útoku, server nesmí provádět žádné systémové nebo databázové akce s otevřeným textem, který dostal od uživatele API. Dalším způsobem zabezpečení je validace vstupních dat.[1]

4.10 Slabiny ve správě API

API může mít více koncových bodů s různými verzemi a funkcemi, například `/ucet/v1` a `/ucet/v2`. Je nutné zajistit správu verzování a kontrolu API podle pravidel:

- Je potřeba udržovat seznam dostupných API, jejich verze, účel (produkce, test, vývoj) a kdo k nim má přístup (veřejné, interní).
- Je potřeba řídit životní cyklus API a včas spouštět nové verze, odstranit staré z podpory.
- Ve veřejné doméně vystavovat pouze nejnovější verze rozhraní API.
- Koncové body pro ladění nebo vývoj měli by být veřejně nedostupné.

V případě nedodržování kontroly ve stravě API, může být API použito k neplánovaným účelům. Například, útočník může najít starší koncový bod `/ucet/v1` který obsahuje bezpečnostní zranitelnost ale po nasazení opravy a povýšení verze, předchozí implementace koncového bodu je stále veřejně přístupná.[1]

4.11 Slabiny v logování a monitorování

Nedostatečné logování a monitorování systému, může přivést k situaci, kdy útočník už delší dobu využívá bezpečnostní zranitelnost ale není to žádným způsobem detekováno. Aby bylo možné detekovat útok nebo podezřelé chování uživatele, musí být systém monitorován a událostí by měli být logování s dostatečnou úrovní podrobností. Níže jsou uvedené kroky pro zlepšení monitorování a logování[13]:

- Zaznamenat všechny neúspěšné pokusy o ověření, zamítnutí přístupu, chyby ve vstupních datech.
- Zajistit integritu logu, aby zabránit možnosti jejich padělání.
- Je nutné sledovat nejen aplikace a příchozí požadavky na API, ale také infrastrukturu, aktivitu sítě, vytížení procesoru počítače a událostí v operačním systému.
- Je nutné zajistit nejen sledování, ale i rychlé upozorňování na porušení pravidelného provozu systému, například použitím *Security Information and Event Management* (SIEM) systému.

5 Výsledky studentské práce

Kapitola se věnuje praktickému řešení zadání práce a je rozdělena do třech částí: návrh, implementace a testování.

Cílem zadání práce je navrhnout a implementovat monitorovací software pro běžící procesy v operačním systému Linux. Každý proces odpovídá specifickému programu, který běží na určitém prostředí, přičemž každý program běží ve zvláštním prostředí, buď je to instance serveru Linux nebo kontejner v Docker. Všechna prostředí jsou umístěné za společnou síťovou infrastrukturou, což umožňuje vzájemnou komunikace dvou a více procesů.

Podle specifikací ve zadání práce, jako výsledný frontend pro uživatelské ovládání bude využít webový framework Flask a pro backend řešení framework Spring a programovací jazyk Kotlin. Komunikace mezi jednotlivými prvky systému bude zajištěna za pomoci API, který musí být efektivně zabezpečen proti jeho zneužití.

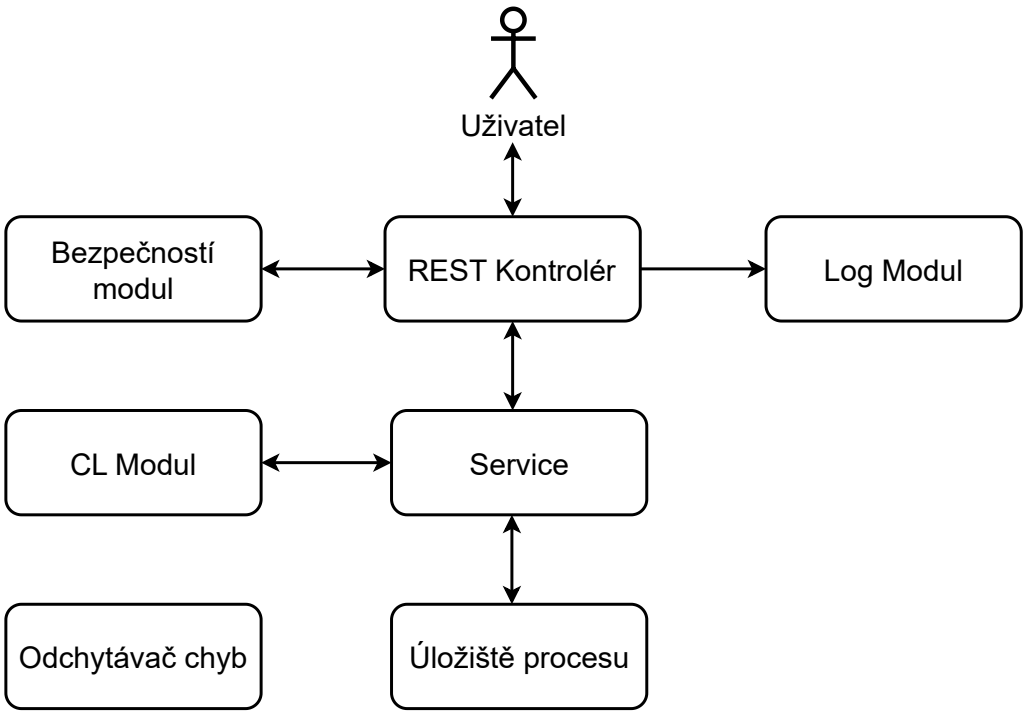
5.1 Návrh řešení

Pro stanovení návrhu řešení zadání práce, je potřeba nejdřív přesně specifikovat problémy, které musí být vyřešené. Jako první problém, který je potřeba vyřešit, lze uvést problém dohledu procesu v Linuxu nebo v prostředí Docker. Dohled procesu by měl být realizován na straně backendu v Kotlin, protože frontend vrstva by měla odpovídat jen za reprezentaci stavů procesu a odeslání požadavku do backendu. Backend tedy měl by být schopen procesy spouštět, monitorovat a zastavovat. Pro spuštění procesu v Kotlin lze využít možnost vytvoření nového procesu, pomocí třídy `Runtime`, následně identifikátor nového procesu by měl být uložen do dočasné paměti. Systém by měl umět detekovat běžící proces, který byl spuštěn například administrátorem, před spuštěním samotného backendu.

Dalším problémem je to, že různé aplikace musí běžet na různých serverech, tedy bude potřeba udržovat více instancí backendu. Proto, je potřeba aby backend server byl navržen tak, aby podporoval více aplikací, tedy aby byl univerzální. Tento backend by šlo spustit na více serverech a upravit pouze konfigurační soubor. Následně, frontend ve Flask by měl rozlišovat, kde běží každá aplikace a na který backend se má odeslat požadavek.

Frontend server by měl počítat s více instancemi backendu a je zde další problém, jak monitorovat více stavů najednou. Řešením může být asynchronní volání každého backendu za jeden časový okamžik.

Nakonec, je potřeba zajistit bezpečnostní stránku řešení, aby jen oprávněný uživatel mohl využívat API backend serveru. Jedna z možností zabezpečení REST API je autorizace na základě API klíčů. Jedná se o často používaný způsob zabezpečení



Obr. 5.1: Návrh struktury backendu

aplikačního rozhraní serveru, který spočívá v tom, že server zpracovává požadavky, které mají v sobě platný API klíč, jinak zpracování odmítá. Tento způsob je vhodný pro případy, kdy nad aplikačním rozhraním je potřeba postavit další aplikaci, kterou může být webová aplikace, mobilní aplikace nebo jiné řešení, které bude pravidelně automaticky provolávat API pro své účely.

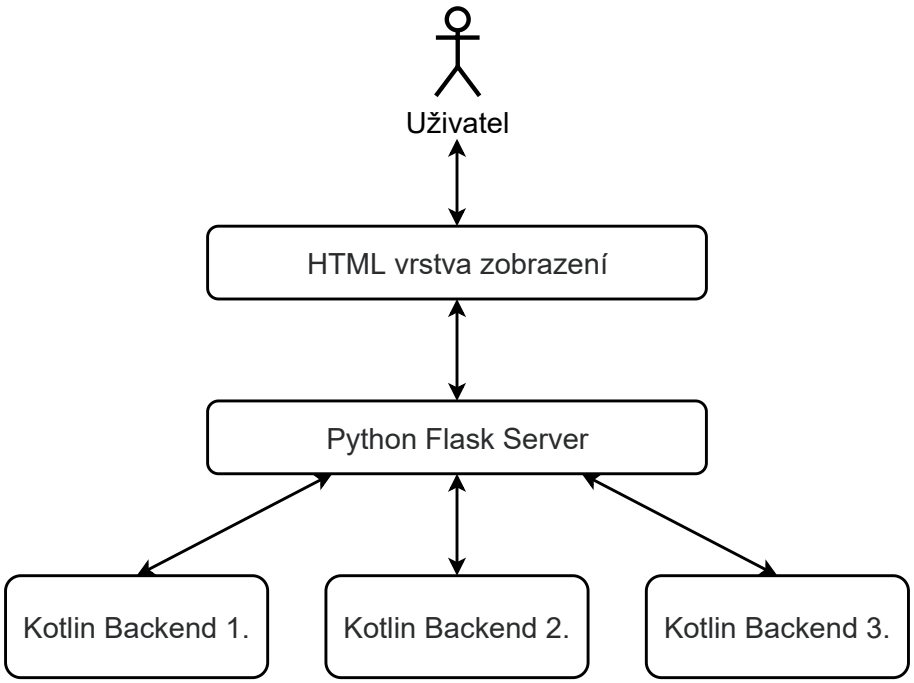
Na obrázku 5.1 je znázorněn návrh struktury backendu, který se skládá z jednotlivých modulů. Každý modul má určitou funkci a nějakým způsobem spolupracuje s dalšími moduly. Základní funkčnost každého modulu lze popsat následně:

- REST Kontrolér – je to třída která definuje API rozhraní podle frameworku Spring. Obsahuje seznam metod, které nabízí API a definice parametrů a návratové hodnoty pro každou metodu.
- Log Modul – jedná se o prvek který loguje všechny příchozí požadavky do kontroléru. Dělá zápis nejen typu požadavku ale i IP adresu uživatele a všechny URL parametry.
- Bezpečností modul – modul, který nastavuje bezpečnostní konfigurace a řeší autorizaci požadavku na základě API klíče.
- Service – třída která implementuje samotné řešení zadání. Přijímá požadavek od kontroléru, zpracovává ho a předává výsledek zpět, který je pak odeslán v HTTP odpovědi.
- CL Modul – znamená Command Line nebo modul který odesílá požadavky do

operačního systému pomocí příkazové řádky. Hlavním úkolem je spustit příkaz a vrátit výsledek z příkazového řádku.

- Úložiště procesu – uchovává identifikátory procesů v mapě kde klíčem je název aplikace a hodnota je PID, jedná se o jedinečnou instanci třídy přes celý backend.
- Odchytávač chyb – modul který přímo nekomunikuje s dalšími moduly ale v případě chyby v systému, odchytí chybu a vrátí uživatelovi správnou informační hlášku, tak systém ne bude posílat citlivé údaje v případě chybného požadavku.

V další části kapitoly je uveden návrh struktury frontendu, který lze vidět na obrázku 5.2. Frontend se skládá z prezentační vrstvy v HTML spolu s *Cascading Style Sheets* (CSS) a vrstvy v Python kde je implementována frontend aplikace pomocí frameworku Flask. Tato vrstva provádí dotazy do backend serverů a následně předává data do HTML ke zobrazení. Pro předávání informace lze použít JavaScript knihovny.



Obr. 5.2: Návrh struktury frontendu

5.2 Backend implementace

Jak již bylo uvedeno výše, implementace serverové části řešení je v programovacím jazyce Kotlin s využitím frameworku Spring. Projekt byl sestaven na platformě

Gradle, což je nástroj pro automatizaci sestavování programu, který podporuje jazyk Kotlin. V průběhu vývoje byl použít verzovací nástroj GitLab, pro snadnější vývoj a možnost jednoduše předávat zdrojový kód.

Další část kapitoly se věnuje podrobnějšímu popisu jednotlivých modulů, včetně ukázek zdrojového kódu v Kotlin.

5.2.1 CL modul

Modul příkazového řádku by měl spouštět příkazy, které dostane jako parametr a vrátit výsledek. Příklad implementace je uveden ve výpisu 5.1. Zde je definována funkce `runCommand` která přijímá příkaz ke spuštění jako `String` hodnotu a vrátí seznam zpráv, kde každá zpráva obsahuje jeden řádek výpisu z příkazového řádku. Nejprve se vytvoří seznam `content` pro ukládání výpisu příkazového řádku, následně probíhá inicializace nového procesu pomocí třídy `Runtime`, kde jako parametr se předává příkaz ke spuštění. Dále se načte výpis příkazového řádku, zapíše se do logu a zároveň do výsledného seznamu, který je pak vrácen jako výsledek funkce.

Výpis 5.1: Funkce pro spuštění příkazu

```
fun runCommand(command: String) : List<Message>{
    val content: MutableList<Message> = ArrayList()

    val runningService = Runtime
        .getRuntime()
        .exec(command)
    val lineReader = BufferedReader(InputStreamReader(
                        runningService.inputStream))
    val errorReader = BufferedReader(InputStreamReader(
                        runningService.errorStream))

    lineReader.lines().forEach { a: String? ->
        log.info(a)
        content.add(Message(a!!))}

    errorReader.lines().forEach { a: String? ->
        log.error(a)}

    return content
}
```

Funkce `runCommand` obsahuje dvě proměnné pro čtení výstupu z konzole, první je `lineReader` druhá – `errorReader`. Je to z toho důvodu, že konzolové okno vypisuje standardní a chybový výstup do zvláštních vláken, proto, aby neztratit chybovou hlášku, je potřeba definovat zvláštní hodnotu pro chybový vystup.

Nevýhoda výše uvedené funkce `runCommand` je v tom, že pokud příkaz ke spuštění způsobí stav, ve kterém příkazový řádek bude běžet delší dobu, tak funkce nebude pokračovat dál. Je potřeba přidat další funkci, pro příkazy které mohou běžet na pozadí aplikace. Implementace je v podstatě stejná jako funkce `runCommand` ale neobsahuje část pro čtení příkazového řádku.

Do modulu práce s příkazovým řádkem je vhodné přidat další užitečnou funkci. Jedná se o funkce pro zjištění stavu procesu podle čísla PID. Příklad implementace je uveden ve výpisu 5.2.

Výpis 5.2: Funkce pro zjištění stavu procesu

```
fun isProcessPidRunning(pid: Int): Boolean {
    val result = runCommand("ps␣-p␣$pid")
    return result.stream()
        .map { m -> m.text }
        .filter { s -> s.contains(pid.toString()) }
        .findAny()
        .isPresent
}
```

Výše uvedená funkce `isProcessPidRunning` přijímá jako parametr číslo PID a vrátí výsledek typu `Boolean`, který může nabývat hodnot `true` nebo `false`. Funkce zavolá příkaz pro zjištění jestli existuje v operačním systému proces, se stejným PID ze vstupu, pokud ano, funkce vrátí `true`, jinak `false`.

Modul příkazového řádku musí umět pracovat s instancemi Docker aplikací. Pro tyto účely jsou implementované zvláštní funkce. První z nich je funkce zjištění stavu Docker kontejneru `isDockerContainerRunning`, podle výpisu 5.3. Princip fungování je stejný jako u předchozí funkce ale implementace je přizpůsobena pro Docker kontejnery. Tak, pro ověření, zda kontejner běží, se použije příkaz `docker ps` který vypíše seznam všech aktuálně běžících kontejnerů. Následně tento příkaz je modifikován pomocí filtru `grep`, který umožňuje najít řádek, který obsahuje podstatnou informaci, v tomto případě, hodnotu `containerId`, která vstupuje do funkce jako parametr. Tato hodnota představuje unikátní identifikátor kontejneru, přes celý systém kde běží Docker. Následně tento příkaz se provolá přes příkazový řádek `bash`, pak podle výstupů lze zkontrolovat jestli `containerId` existuje a pokud ano, bude vrácena hodnota `True`, jinak `False`.

Výpis 5.3: Funkce pro zjištění stavu Docker kontejneru

```
fun isDockerContainerRunning(containerId:String):Boolean {
    val command = "docker␣ps␣|␣grep␣$containerId"
    val process = Runtime.getRuntime().exec(arrayOf(
        "/bin/sh", "-c", command))
    val reader = BufferedReader(InputStreamReader(
        process.inputStream))
    val errorReader = BufferedReader(InputStreamReader(
        process.errorStream))
    val output = reader.readLine()
    val errorOutput = errorReader.readLine()

    if (errorOutput != null) {
        log.error("Error␣output:␣$errorOutput")
    }
    return output != null && output.contains(containerId)
}
```

Další důležitou funkci je `getDockerContainerIdByName`. Tato funkce umožňuje zjistit hodnotu `containerId` pro známý název kontejneru. Zde se použije příkaz pro hledání identifikátoru kontejneru v seznamu podle výpisu 5.4, kde název kontejneru se předává jako parametr `containerName`. Následně z výstupu se vezme první hodnota, kterou je `containerId`.

Výpis 5.4: Příkaz pro výpis Docker kontejnerů podle názvu

```
val command = "docker␣ps␣|␣grep␣$containerName"
```

5.2.2 Úložiště procesu

Tato třída odpovídá za uložení čísla PID pro odpovídající proces nebo identifikátor Docker kontejnerů a měla by být pouze jedna instance této třídy. Jedinečnost instance třídy v Kotlin lze dosáhnout pomocí klíčového slova `object`. Tento modul obsahuje tři funkcí, první z nich je funkce pro zjištění již uloženého PID aplikace, příklad implementace je uveden ve výpisu 5.5

Výpis 5.5: Funkce pro získání uloženého PID

```
fun getProcessIdByAppName(appName: AppNameEnum): String? {
    return processIdMap[appName]
}
```

Funkce přijímá jeden parametr typu `AppNameEnum`, což je výčet názvů aplikací které podporuje backend. Následně je vrácen PID procesu aplikace nebo `containerId`, pokud takový existuje v mapě, jinak je vrácena `null` hodnota.

Další funkce `setProcessIdToApp` je určena pro zápis PID nebo `containerId` určité aplikace do mapy. Příklad implementace je ve výpisu 5.6.

Výpis 5.6: Funkce pro uložení PID

```
fun setProcessIdToApp(appName: AppNameEnum,
                      processId: String) {
    var oldProcessId = getProcessIdByAppName(appName)
    if (oldProcessId != null) {
        processIdMap.remove(appName)
    }

    processIdMap[appName] = processId
}
```

Funkce `setProcessIdToApp` přijímá dva parametry: název aplikace a PID procesu nebo `containerId`. Následně probíhá kontrola, jestli aplikace už má přidělen identifikátor, pokud ano, bude smazáno s uloží hodnota nového identifikátoru.

Poslední funkce v modulu, je funkce ke smazání existujícího PID nebo identifikátoru kontejnerů. Její implementace je obdobná funkci `setProcessIdToApp` ale bez uložení nové hodnoty identifikátoru. Funkce je užitečná pro případy, kdy je potřeba zastavit proces a tím pádem smazat jeho identifikátor z úložiště.

5.2.3 Log modul

Cílem logovacího modulu je odchytit příchozí požadavek na API a zaznamenat URL a IP adresu uživatele. Aby toho dosáhnout, je potřeba implementovat filter který je v frameworku Spring. Příklad definice třídy je uveden ve výpisu 5.7.

Výpis 5.7: Příklad vlastního filtru ve Spring framework

```
@Component
class RequestLoggingFilter : Filter {
    private val log = LoggerFactory
        .getLogger("RequestLoggingFilter")
}
```

Zde je potřeba uvést důležitou anotaci `@Component`, která informuje Spring framework o tom, že je potřeba automaticky vytvořit instanci této třídy, přičemž pouze

jedné instance. Tento filter musí implementovat funkci `doFilter`, příklad implementace je uveden ve výpisu 5.8.

Výpis 5.8: Příklad implementace filtru

```
override fun doFilter(
    servletRequest: ServletRequest,
    servletResponse: ServletResponse,
    filterChain: FilterChain
) {
    val request = servletRequest as HttpServletRequest
    val remoteHost = request.remoteAddr
    val path = request.requestURI
    val method = request.method
    val protocol = request.protocol

    log.info("Request␣from␣$remoteHost
␣␣␣␣␣␣␣␣=>␣$protocol␣$method␣$path")
    filterChain.doFilter(servletRequest, servletResponse)
}
```

Funkce `doFilter` přijímá příchozí požadavek, odpověď, která bude odeslána a samotný řetězec filtrů. V tomto kroku lze z příchozího požadavku vyčíst důležitou informaci: IP adresu uživatele, API metodu, URL parametry a použitý protokol. Díky tomu, že lze zjistit IP adresu uživatele, je možné implementovat funkce řízení přístupu ke API, různé metriky a navíc kontrolovat IP adresu jestli není v seznamu zakázaných uživatelů. Aby povolit příchozí požadavek ke zpracování, je potřeba zavolat funkce `filterChain.doFilter` čímž, požadavek bude pokračovat do dalšího filtru, pokud takový existuje.

5.2.4 Odchytávač chyb

Odchytávač chyb ve Spring se definuje jako třída s anotací `@ControllerAdvice`. Poté je možné implementovat funkci pro zachytávání výjimek v systému, příklad implementace je uveden ve výpisu 5.9.

Funkce přijímá příchozí požadavek a výjimku, kterou způsobil tento požadavek. Následně je možné z výjimky přečíst název parametru a hodnotu aby sestavit uživatelsky příjemnou hlášku, která bude vrácena spolu s odpovídajícím status kódem. Funkce odchytává pouze výjimky které souvisí se špatnou hodnotou parametru požadavku. Pro každý typ výjimek je potřeba definovat zvláštní funkci a uvádět typ

výjimky v anotaci `@ExceptionHandler` aby Spring framework mohl směrovat příchozí požadavek do správné funkce.

Výpis 5.9: Příklad implementace zachytávání výjimek

```
@ExceptionHandler(MethodArgumentException::class)
fun handleInvalidParamException(
    servletRequest: HttpServletRequest,
    exception: Exception):
        ResponseEntity<Message> {

val ex = exception as MethodArgumentTypeMismatchException
val name = ex.parameter.parameterName
val value = ex.value
val msg = Message("Invalid value for $name param: $value")
return ResponseEntity(msg, HttpStatus.BAD_REQUEST)
}
```

5.2.5 Service

Servisní třída obsahuje samotné řešení celého backendu, tedy spojuje jednotlivé moduly dohromady. Třída musí mít anotaci `@Service` aby informovat Spring o tom, že se jedna o servisní třídu. Jsou zde definované veřejné a soukromé funkce. Mezi veřejné funkce patří `executeOperation`, která odpovídá za spuštění určité operaci nad procesem a `getAppStateByName` která vrátí stav procesu.

Funkce `executeOperation` nejprve zjisti jestli tento backend server podporuje požadovanou aplikaci a pokud ne, vrátí chybovou hlášku pro uživatele, jinak zavolá soukromou funkce pro odpovídající aplikaci a předá typ operace, který se má provést. Systém podporuje tři typy operace: spustit, zastavit a restartovat proces. V následujícím kroku podle požadovaného typu operace se provolá modul příkazového řádku s odpovídajícím příkazem ke spuštění. Výsledek je pak přeformátován do uživatelské hlášky a vrácen v odpovědi. Funkce taky podporuje mezní stavy, kdy například uživatel zkusí spustit stejný proces několik krát.

Veřejná funkce `getAppStateByName` nedělá nic dalšího než zjistí stav aplikace podle PID nebo `containerId` v případě, že se jedná o Docker aplikací, pokud existuje, a vrátí odpovídající hlásku, zda aplikace běží nebo ne.

Servisní třída obsahuje další pomocní soukromé funkce. Příkladem může být funkce `startParser` pro spuštění programů Parser, podle výpisu 5.10. Funkce využívá modul příkazového řádku `CommandLineUtils` a předává příkaz ke spuštění

programu, který je uložen do proměnné `commandParserStart` a jeho hodnota je určena v konfiguračním souboru. V případě, že program se nepodaří spustit z důvodu jakékoliv chyby, bude vrácena chybová hláška.

Výpis 5.10: Příklad implementace spuštění programu

```
private fun startParser(): Int {
   CommandLineUtils.runCommandNoOutput(commandParserStart)
   return getParserPidFromSystem() ?:
        throw LinuxApiException(
            "Start␣parser␣exception:␣PID␣not␣found")
}
```

Mezi další užitečné privátní funkce patří `stopParser`, který využívá funkce s názvem `CommandLineUtils` k zastavení programu pomocí příkazu `kill`. V případě, že příkaz vrátí neočekávaný výstup, bude vrácena chybová hláška. Příklad implementace je uveden ve výpisu 5.11.

Výpis 5.11: Příklad implementace zastavení programu

```
private fun stopParser(processId: Int) {
    val messageList = CommandLineUtils
            .runCommand("kill␣$processId")
    if (messageList.isNotEmpty()) {
        throw LinuxApiException(
        "Kill␣command␣returns␣unexpected␣result:
␣␣␣␣␣␣␣␣␣␣␣␣␣$messageList")
    }
}
```

5.2.6 REST Kontrolér

Jedná se o třídu, která definuje REST API rozhraní. Třída musí mít anotací `@RestController` aby Spring framework mohl identifikovat třídu jako vstupní bod příchozích požadavků. Další důležitou anotací je `@RequestMapping("/manage")` který určuje mapovací URL, v tomto případě je to `/manage` protože se jedná o ovládání procesů.

Aby kontrolér byl schopen provolat veřejné funkce ze servisy, musí mít přístup do instancí servisní třídy. Ve Spring framewrok se toho lze dosáhnout pomocí automatického importu závislé třídy. Příklad importování servisy je uveden ve výpisu 5.12.

Výpis 5.12: Příklad implementace zastavení programu

```
@Autowired
private lateinit var service: ManageService
```

Zde je důležitá anotace `@Autowired` která říká Spring aby provedl auto import. Dále je potřeba definovat servisu jako `lateinit`, protože může se stát, že instance třídy kontroléru se vytvoří dřív, než servisy, pak dojde k chybě naběhnutí serveru.

Definice API v REST kontroléru je realizována pomocí funkce. Tak, funkce pro zjištění stavu programu, je uvedena ve výpisu 5.13.

Výpis 5.13: Definice REST API pro zjištění stavu aplikace

```
@GetMapping("/{APP_NAME}/state")
fun getAppStateByName(
    request: HttpServletRequest,
    @PathVariable APP_NAME: AppNameEnum):
        AppDetail {
    return service.getAppStateByName(APP_NAME)
}
```

Funkce podle výpisu výše definuje koncový bod API na URL adrese `/manage/APP_NAME/state` kde `APP_NAME` je parametr, který obsahuje název aplikace. Funkce volá servisní implementace a vrátí objekt typu `AppDetail` který je pak automaticky převeden do formátu JSON. Tento objekt obsahuje dva atributy: stav aplikace a textovou zprávu, která je volitelná.

Dalším koncovým bodem který definuje REST kontrolér je funkce pro spuštění operace nad procesem. Příklad takové funkce je uveden ve výpisu 5.14. Jedná se o metodu typu PUT, která přijímá dva parametry v URL: název aplikace a název operace, která se má provést.

Výpis 5.14: Definice REST API pro spuštění operace nad procesem

```
@PutMapping("/{APP_NAME}/{OPERATION}")
fun executeOperation(request: HttpServletRequest,
        @PathVariable APP_NAME: AppNameEnum,
        @PathVariable OPERATION: OperationEnum):
            AppDetail {
    return service.executeOperation(APP_NAME, OPERATION)
}
```

Všechny výše uvedené koncové body API, jsou zapsané do dokumentace podle OpenAPI specifikace ve Swagger souboru.

5.2.7 Bezpečností modul

Bezpečnost API je zajištěna na základě API klíče. Tento klíč je možné nastavit v konfiguračním souboru serveru a to tak, že každá instance backend serveru, může mít zvláštní API klíč. Princip fungování bezpečnostního modulu spočívá v kontrole hodnoty API klíče který musí být předán v hlavičce HTTPS požadavku. V případě, že požadavek neobsahuje API klíč nebo jeho hodnota není správná, bezpečnostní modul odmítá zpracování požadavku. Aby tento mechanismus fungoval bezpečně, je potřeba použít protokol HTTPS, díky čemu, obsah hlavičky a tedy i API klíče, bude zašifrován. Pro vývojové a testovací účely, stačí mít na straně serveru nakonfigurovány *self-signed* certifikát. Jedná se o certifikát, který lze vytvořit a podepsat samostatně, bez nutnosti kontaktovat certifikační autoritu.

Implementace bezpečnostního modulu se skládá z několika částí. Nejprve je potřeba nadefinovat konfigurační třídu `SecurityConfig`, příklad definice je uveden ve výpisu 5.15. Zde je uvedena anotace `@Configuration`, díky čemu, Spring framework může detekovat třídu jako konfigurační. Následně je potřeba informovat Spring framewrok, že se jedna o konfigurace bezpečnosti serveru, což zaručí anotace `@EnableWebSecurity`. Poslední součástí definice třídy je rozšíření rozhraní `WebSecurityConfigurerAdapter`, díky čemu, lze upravovat výchozí chování bezpečnostní konfigurace ve Spring framework.

Výpis 5.15: Definice bezpečnostní konfigurační třídy

```
@Configuration
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter()
```

Bezpečnostní modul, musí určitým způsobem kontrolovat hlavičky příchozích požadavků. Toho lze dosáhnout přepisem konfigurace podle výpisu 5.16.

Výpis 5.16: Konfigurace ověření příchozích požadavků

```
override fun configure(http: HttpSecurity) {
    http.authorizeRequests { authorize ->
        authorize.antMatchers("/api/**").authenticated()
    }
    .csrf { it.disable() }
    .sessionManagement { it.sessionCreationPolicy(
        SessionCreationPolicy.STATELESS) }
    .addFilterBefore(
        apiKeyAuthenticationFilter,
        UsernamePasswordAuthenticationFilter::class.java)
}
```

Výše uvedený blok kódu v jazyce Kotlin, upravuje výchozí chování bezpečnostní konfigurace v Spring framework tak, že nejprve nastaví atribut `authenticated` pro všechny příchozí požadavky na URL adresu `/api/**`, což vlastně znamená, jakoukoliv adresu na serveru. Další řádek kódu, vypíná *Cross-Site Request Forgery* (CSRF), což je v souladu s dokumentací Spring framework pro vývoj API, který není přímo určen pro uživatelské použití ve webovém prohlížeči. Následně se nastavuje typ relace jako *stateless*, což znamená že Spring nebude používat HTTP session. Jako poslední krok, je potřeba nastavit filtr, který bude provádět samotnou kontrolu API klíče. Filtr se nastavuje pomocí funkce `addFilterBefore`, což zajistí spuštění filtru před každým příchozím požadavkem.

Definice výše zmíněného bezpečnostního filtru je uvedena ve výpisu 5.17. Zde je použitá anotace `@Component` aby Spring mohl automaticky vytvořit instance této třídy. Následně filtr rozšiřuje rozhraní `OncePerRequestFilter`.

Výpis 5.17: Definice vlastního bezpečnostního filtru

```
@Component
class ApiKeyAuthenticationFilter : OncePerRequestFilter()
```

Samotná kontrola API klíče se pak provádí ve funkci `doFilterInternal` která je rozdělena do dvou výpisů. Definice funkce, včetně vstupních parametrů je uvedena ve výpisu 5.18. Funkce přijímá tři parametry, které se automaticky doplní pomocí frameworku Spring, jejich význam je následující:

- HttpServletRequest – instance příchozího HTTP požadavků, jedná se o objekt který obsahuje všechny atributy HTTP protokolu a ze kterého lze vyčíst vstupní data a záhlaví.
- HttpServletResponse – instance odchozího HTTP požadavků, objekt který bude po zpracování požadavku odeslán jako odpověď, pomocí Spring framework. Umožňuje upravit odpověď požadavků nebo nastavit specifické atributy.
- FilterChain – instance třídy, která reprezentuje řetězec filtru, je povinný parametr jakéhokoliv filtru v Spring.

Výpis 5.18: Definice funkce vlastního filtru

```
override fun doFilterInternal(
                        request: HttpServletRequest,
                        response: HttpServletResponse,
                        filterChain: FilterChain)
```

Implementace kontroly API klíče je uvedena ve výpisu 5.19. Na prvním řádku se provádí čtení hodnoty API klíče ze záhlaví HTTP požadavku. Hodnoty v hlavice požadavků se ukládají ve formátu klíč – hodnota. Pro API klíč byl zvolen klíč

v hlavičce jako „X-API-KEY“, kde „X“ znamená, že se jedná o vlastní klíč, který není v standardu protokolu HTTP uveden. Hodnota tohoto klíče se pak ukládá do proměnné `apiKey`. Následně probíhá kontrola, jestli hodnota API klíče je vůbec uvedena, pokud není, upraví se odpověď požadavků tak, že se nastaví HTTP kód 401, což znamená „SC_UNAUTHORIZED“ tedy chybějící autorizace a předá se odpovídající chybová hláška.

Výpis 5.19: Příklad kontroly API klíče v bezpečnostním filtru

```
val apiKey = request.getHeader("X-API-KEY")

if (apiKey == null || apiKey.isEmpty()) {
    response.sendError(
    HttpServletResponse.SC_UNAUTHORIZED,"Missing␣API␣Key")
    log.error("Missing␣API␣Key")
    return
} else if (apiKey != storedApiKey) {
    response.sendError(
    HttpServletResponse.SC_UNAUTHORIZED,"Invalid␣API␣Key")
    log.error("Invalid␣API␣Key")
    return
}

val authentication = UsernamePasswordAuthenticationToken(
    apiKey, apiKey)
SecurityContextHolder
    .getContext()
    .authentication = authentication
filterChain.doFilter(request, response)
```

Pokud hodnota API klíče byla předána v záhlaví požadavku, provede se kontrola jeho shody s klíčem, který byl uveden v konfiguračním souboru na serveru. V případě neshody, bude upravena odpověď požadavku na hodnotu 401 s odpovídající chybovou hláškou. Pokud dvě předchozí kontroly jsou v pořádku, provede se autorizace požadavků a výsledek se zapíše do kontextu `SecurityContextHolder`, díky čemu, v případě potřeby lze využít API klíč ze vstupu v jiné části serveru pomocí načtení kontextu.

Jako poslední krok, probíhá provolání řetězců filtrů, aby Spring framework mohl považovat tento filtr za splněný a měl povolení spustit další filtry, pokud takové existují.

5.2.8 Konfigurace backend serveru

Důležitou součástí backend serveru je možnost konfigurace dynamických parametrů. Jedná se o takové parametry, které se mohou lišit pro jednotlivé instance backend serverů na různých prostředí v reálném provozu. Tyto pramatery je potřeba předem uvést v konfiguračním souboru `application.properties`, který se pak předá jako parameter při spuštění backend serveru. Příklad nastavení vlastních hodnot v konfiguračním souboru je uveden ve výpisu 5.20.

Výpis 5.20: Příklad definice konfiguračních hodnot

```
command.hive.container.name=thehiveproject/thehive4
monitor.supported-apps=KAFKA,ZOOKEEPER,MONGODB
```

Princip je založen na typu klíč – hodnota, kde klíčem může být textový řetězec oddělený tečkou nebo symbolem minus pro přehlednost. Tyto parametry lze následně načít v kódu pomocí anotace `@Value`, kde je potřeba uvést celou hodnotu klíče z konfiguračního souboru, příklad použití je uveden ve výpisu 5.21. V případě, že konfigurační hodnota je jednoduchý textový řetězec, bude automaticky zapsána do proměnné typu String. Ale pokud se jedna o několik hodnot, oddělené čárkou, které je potřeba zpracovat jako seznam, musí být doplněn název klíče o funkci `split` s uvedením symbolu oddělení hodnot, v tomto případě čárkou.

Výpis 5.21: Příklad čtení konfiguračních hodnot

```
@Value("\${command.hive.container.name}")
private lateinit var theHiveContainerName: String
@Value("#{'\${monitor.supported-apps}'.split(',')}")
private lateinit var supportedApps: List<AppNameEnum>
```

5.2.9 OpenAPI dokumentace implementovaného API

Pro snadnější údržbu a případný budoucí vývoj, celé backend REST API rozhraní je zapsané formou OpenAPI dokumentace, která se skládá z jednoho souboru typu yaml. V hlavičce tohoto souboru je uvedena verze OpenAPI a případně další užitečné informace jako například název a popis projektu, odkazy do externí dokumentace nebo URL adresa serveru.

Jednotlivé REST API rozhraní jsou rozdělené do dvou typů pomocí tagů. Jedná se o REST API které jsou jenom pro čtení nebo takové API, které způsobují změnu stavu aplikace. Příklad definice tagu podle OpenAPI je uveden ve výpisu 5.22. Tag se skládá ze dvou polí: název a popis, přičemž název musí být unikátní přes všechny tagy v dokumentaci.

Výpis 5.22: Příklad definice tagu podle OpenAPI

```
tags:
  - name: read-only
    description: Read only operation.
  - name: app-control
    description: App control operation,
    this operation may affect the state of the application
```

Další součástí OpenAPI dokumentace je definice API rozhraní v rámci struktury `paths`. Příklad dokumentace REST API pro čtení stavu aplikace je uveden ve výpisu 5.23. První řádek uvádí definice URL adresy API, na druhém řádku je uveden typ požadavku, v tomto případě se jedná o GET požadavek. Pole `operationId` obsahuje unikátní identifikátor REST API koncového bodu.

Výpis 5.23: Příklad definice REST API rozhraní

```
/manage/{APP_NAME}/state:
get:
  description: Returns a state of application.
  operationId: getAppStateByName
  tags:
    - read-only
  parameters:
     - $ref: '#/components/parameters/APP_NAME'
  security:
    - ApiKeyAuth: []
  responses:
    '200':
      description: successful operation, returns state
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/AppDetail'
    '400':
      description: 'Invalid value for APP_NAME param'
    '401':
      description: 'UNAUTHORIZED'
    '404':
      description: 'Application {APP_NAME} not supported'
```

Pole `parameters` definuje seznam URL parametru které se očekávají při volání API, v tomto případě se jedná o název aplikace. Následně je uveden typ zabezpečení API a seznam návratových kódů, které může vrátit API, včetně popisů případů, ve kterých může vyskytnout odpovídající HTTP kód. Je důležité uvést, že v případě kódu 200, což znamená úspěšné zpracování požadavků, je vrácen objekt typu JSON který je definován podle OpenAPI jako schémata `AppDetail`, příklad definice je uveden v následujícím výpisu 5.24.

Výpis 5.24: Příklad definice JSON odpovědi podle OpenAPI

```
schemas:
    AppDetail:
      type: object
      properties:
        state:
          type: string
          description: Application state
          example: RUNNING
          enum:
            - RUNNING
            - STOPPED
            - ERROR
        message:
          type: string
```

Podle výše uvedeného výpisu je vidět definice JSON objektu s názvem `AppDetail`, který obsahuje dva parametry s názvy *state* a *message* přičemž, parameter *state* je typu `enum` s přesně definovaným seznamem možných hodnot.

5.3 Frontend implementace

Frontend aplikace se skládá ze dvou vrstev: Flask serveru a HTML vrstvy zobrazení. Flask server je aplikace, hlavním úkolem které je zpracování požadavku z vrtsvy zobrazení a komunikace s Kotlin backend serverem prostřednictvím REST API rozhraní. HTML vrstva odpovídá za zobrazení stavů monitorujících aplikací a celkově řeší jen reprezentační stránku frontend serveru, komunikuje jen s Flask serverem.

V další části kapitoly následuje podrobnější popis důležitých kroků v implementaci frontend serveru.

5.3.1 Flask aplikace

Flask je microframework v programovacím jazyce Python, který umožňuje jednoduše sestavit frontend server. Nejprve je potřeba vytvořit `init` soubor ve kterém probíhá vytvoření Flask aplikace a následně její konfigurace. Příklad inicializace a konfigurace aplikace je uveden ve výpisu 5.25.

Výpis 5.25: Příklad vytvoření Flask aplikace

```
app = Flask(__name__, static_url_path='/static')

app.config['SESSION_COOKIE_SECURE'] = True
app.config['DEBUG'] = True
```

Pro vytvoření Flask aplikace se používá konstruktor, kterému se předává název aplikace a cesta do složky, kde jsou umístěné statické soubory, jako například CSS. Následně je možné konfigurovat aplikace pomocí klíčů, ve výše uvedeném příkladě se nastavuje bezpečné použití cookie a debug mód.

Dalším krokem je potřeba vytvořit startovací Python skript, který spustí Flask server. Pro spuštění serveru lze využít knihovnu `socketio`, příklad použití je uveden ve výpisu 5.26.

Výpis 5.26: Příklad spuštění Flask aplikace

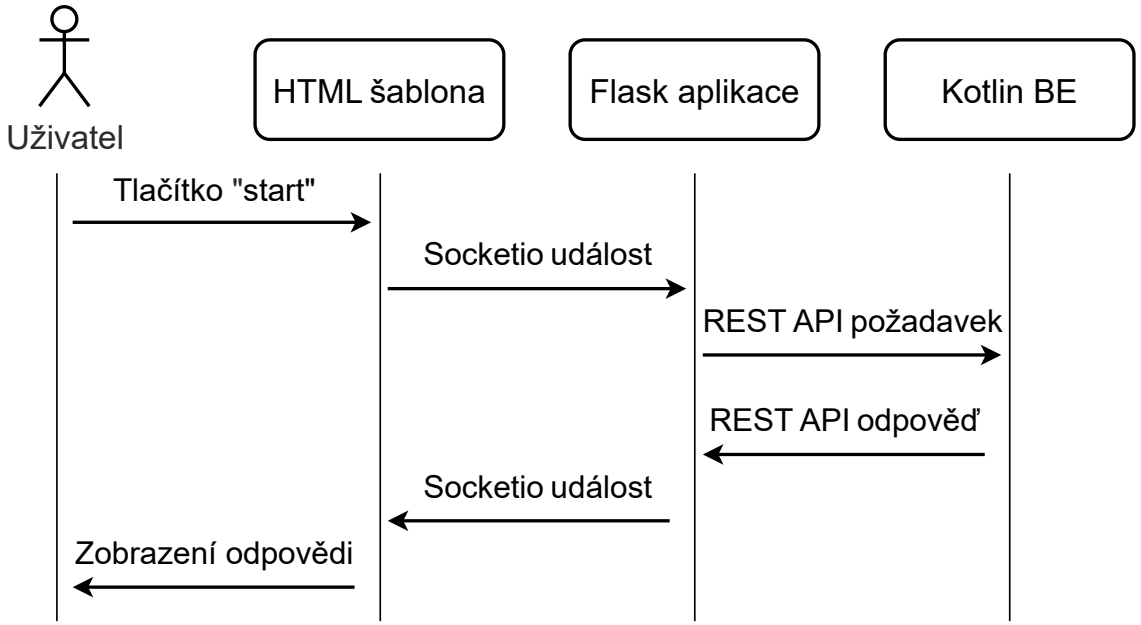
```
socketio.run(app, HOST, PORT,
    debug=True, use_reloader=True)
```

Po spuštění serveru, Flask hledá soubor `views.py`, ve které jsou popsané koncové body aplikace. Princip fungování spočívá v tom, že v souboru `views.py` jsou definované URL adresy které podporuje server a je uvedeno na kterou stránku vede URL adresa. Pro tyto účely se používá anotace `@app.route` která jako parametr přijímá typ požadavku a URL adresu, příklad definice domovské stránky je uveden ve výpisu 5.27

Výpis 5.27: Příklad spuštění Flask aplikace

```
@app.route('/home', methods=['POST', 'GET'])
def home():
    return render_template('index.html')
```

Takže podle výpisu uvedeného výše, při zavolání stránky s URL adresou `/home`, se provede zobrazení stránky podle HTML souboru `index.html`. Tohle zajistí funkce `render_template`, která může přijímat další volitelné parametry, které lze použít pro předávání dynamických dat do HTML šablony. Nevýhoda předávání dat tímto způsobem je v tom, že data se odesílají jen při načtení stránky. Aby udělat výměnu



Obr. 5.3: Princip fungování frontend serveru

dat mezi Flask aplikací a HTML vrstvou na pozadí, lze použít `socketio` knihovnu. Princip fungování této knihovny je založen a událostech, tedy na jedné straně se definuje funkce která naslouchá určitou událost, na druhé straně funkce pro odeslání události. Jakmile událost bude odeslána z HTML vrstvy, na straně Flask aplikace se spustí zpracování událostí, včetně přijatých dat jako volitelný parametr. Princip fungování je znázorněn na obrázku 5.3.

Na straně HTML vrstvy se použije JavaScript knihovna, pomocí které se odešle událost do Flask aplikace, příklad odeslání události na spuštění programů Parser, je uveden ve výpisu 5.28. Zde se použije funkce `emit` která přijímá jeden parametr – název události, v tomto případě se odesílá událost ke spuštění programu Parser s názvem události `startParser`.

Výpis 5.28: Příklad odelsani události

```
socket.emit('startParser');
```

Na druhé straně, na straně Flask aplikace, se odchytí tato událost pomocí funkce podle výpisu 5.29. Tato funkce má speciální anotaci `@socketio.on` která umožňuje knihovně `socketio` detekovat tuto funkce jako naslouchače určité události. Název události, kterou funkce naslouchá, se uvádí jako první argument anotace, v tomto případě se jedná o událost `startParser`. Jako druhý argument anotace se uvádí `namespace`, což znamená v podstatě skupinu, do které patří určitá událost. Pomoci `namespace` parametru lze se vyhnout kolizím, pokud existuje více naslouchačů se stejným názvem události, ale mají odlišnou skupinu.

Výpis 5.29: Příklad volání REST API

```
@socketio.on('startParser', namespace='/dashboard')
def startParser():
    headers = {'X-API-KEY': API_KEY}
    r = requests.put(PARSER_BASE_URL +
        '/api/manage/PARSER/START',
        headers=headers,
        verify=False)
    response = r.json()
    message = "Parser started: "
        + response['state'] + " "
        + response['message']
    data = {
        'appName' : 'Parser',
        'message': message }
    logging.info(data)
    socketio.emit('alert', data, namespace='/dashboard',
        broadcast=True)
```

Funkce `startParser` si nejprve sestaví hlavičku REST požadavků, kde pod klíčem `X-API-KEY` se uloží hodnota API klíče z konfigurace Flask serveru, hodnota tohoto klíče má být stejná jako na Kotlin serveru, kam bude požadavek odeslán. Následně probíhá sestavení HTTP požadavku typu PUT, pomocí konstruktoru z Python knihovny `requests`. Jako první parametr se uvádí IP adresa cílového serveru z konfigurace, včetně URL adresy, dále se nastaví hlavička požadavku a proběhne vypnutí kontroly certifikátů cílového Kotlin serveru, protože pro testovací účely se používá *self-signed* certifikát.

Následně se odešle REST požadavek na odpovídající backend server a přijatá odpověď je převedena do JSON formátu, díky knihovně `requests`. Z přijaté odpovědi se pak načte stav provedení operace, zda byl úspěšný nebo ne, a případně doplňující zpráva. Tato informace se uloží do objektu `data`, který je pak následně přeposlán na HTML vrstvu k zobrazení informační hlášky. Předání funguje stejným způsobem ale opačným směrem, tedy Flask aplikace posílá událost s názvem `alert`, tuto událost odchytí JavaScript funkce na straně HTML a může dál pracovat s daty na vstupu.

5.3.2 Asynchronní volání REST API

Frontend server musí monitorovat stav celé řady aplikací a umožnit uživatelovi zároveň ovládat určitou aplikaci bez nutnosti čekání ve frontě ostatních požadavků. Pro tyto účely, implementace monitorování stavu aplikaci byla navržena asynchronním

způsobem. Nejprve ve Flask serveru se nastaví globální boolean parametr s názvem `isMonitoring` podle kterého se bude zapínat a vypínat monitorování. Následně je potřeba vytvořit monitorovací funkce, která bude co nejvíc univerzální. Příklad této funkce je uveden ve výpisu 5.30.

Výpis 5.30: Příklad monitorovací funkce

```
def monitoringState(url, app):
    while isMonitoring:
        headers = {'X-API-KEY': API_KEY}
        rawResponse = requests.get(url,
            headers=headers,
            verify=False)
        jsonResponse = rawResponse.json()
        logging.info("%s␣STATE␣response:␣%s",
            app, jsonResponse)
        state = jsonResponse['state']
        message = jsonResponse['message']
        state_details = {
            'appName' : app,
            'state': state,
            'message': message
        }
        socketio.emit('statusUpdate', state_details,
            namespace='/dashboard', broadcast=True)
        socketio.sleep(3)
```

Výše uvedená funkce přijímá dva argumenty: URL adresu požadavků a název aplikace. Funkce spouští cyklus while, který bude běžet, pokud je nastaven globální parametr `isMonitoring` na hodnotu True. Tato funkce nedělá nic jiného, než sestaví HTTP GET požadavek podle URL adresy na vstupu, včetně API klíče a provolá tento požadavek. Následně přijatou JSON odpověď přepíše do nové datové struktury, kterou odešle do HTML vrstvy ke zobrazení. Důležitou poznámkou, je nastavení zpoždění 3 vteřiny po odeslání dat do HTML vrstvy, je to z důvodu aby nezahltit síť příliš častým voláním Kotlin serveru.

Samotné asynchronní volání funkce `monitoringState` je implementováno ve funkci `activate` podle výpisu 5.31. Tato funkce je definována jako naslouchač určité události, kterou odesílá HTML vrstva, v případě že uživatel zapnul monitorování. Uvnitř funkce se nejprve nastaví globální parametr `isMonitoring` na hodnotu True, následně probíhá příprava samostatného vlákna pro každou aplikaci. Jako vlákno se používá implementace z Python knihovny `Thread`, kde pomocí konstruktoru se

nastavuje funkce, která bude běžet ve vlákně a její argumenty, v tomto případě se jedná o URL adresu a název monitorující aplikace. Jako další krok, se nastaví příznak `deamon` pro každé vlákno. Jedná se o optimalizace serveru, protože `deamon` vlákno neblokuje běh celého serveru, v případě, že vyskytne nějaká chyba v hlavním vlákně Flask aplikace. Nakonec, každé vlákno se spustí příkazem `start`.

Tím je zajištěno, že každých 3 sekundy probíhá aktualizace stavu monitorujících aplikací. Pokud uživatel zvolí možnost zastavení monitorování, stačí provolat funkci, která nastaví globální parametr `isMonitoring` na hodnotu False, tím bude zajištěno korektní zastavení vláken.

Výpis 5.31: Příklad asynchronního volání funkce

```
@socketio.on('activate', namespace='/dashboard')
def activate():
    global isMonitoring
    isMonitoring = True

    t1 = Thread(target=monitoringState,
        args=(PARSER_BASE_URL +
            '/api/manage/PARSER/state','Parser'))
    t2 = Thread(target=monitoringState,
        args=(HIVE_BASE_URL +
            '/api/manage/HIVE/state','TheHIVE'))

    t1.daemon = True
    t2.daemon = True

    t1.start()
    t2.start()
```

5.3.3 Konfigurace Flask serveru

Microframework Flask podporuje dynamickou konfiguraci uživatelských parametrů. Aktuální implementace Flask serveru vyžaduje podporu konfigurování IP adress Kotlin serveru pro každou monitorující aplikací a jejich název a také API klíč. Pro tyto účely, je potřeba vytvořit `application.properties` soubor ve složce spolu se zdrojovými kódy aplikace. Uvnitř konfiguračního souboru, hodnoty se uvádí ve formátu klíč = hodnota podle výpisu 5.32. V případě, že je potřeba uvést více hodnot do jednoho klíče, lze tyto hodnoty oddělit například čárkou, a upravit způsob načtení parametru.

Výpis 5.32: Příklad konfiguračního souboru pro Flask server

```
APPLICATIONS = 'Parser, TheHIVE, Kafka'
PARSER_URL = 'https://10.0.0.90:8080'
```

Pro čtení konfiguračních hodnot uvnitř Flask serveru, je potřeba nejprve inicializovat konfigurační soubor příkazem podle výpisu 5.33.

Výpis 5.33: Inicializace konfigurace ve Flask

```
app.config.from_pyfile('application.properties')
```

Následně je možné načíst konfigurační parametr způsobem, uvedeným ve výpisu 5.34. Pro čtení jednoduchého parametru, stačí předat název parametru. V případě, že je potřeba načíst seznam hodnot, je potřeba předtím nastavit symbol oddělení hodnot, v tomto případě symbol čárky. Následně pomocí funkce `strip` odstranit mezery kolem hodnot.

Výpis 5.34: Inicializace konfigurace ve Flask

```
PARSER_BASE_URL = app.config['PARSER_URL']

app_list = app.config['APPLICATIONS'].split(',')
trimmed_app_list = [value.strip() for value in app_list]
```

5.3.4 Vrstva zobrazení

Vrstva zobrazení frontend aplikace se skládá z JavaScript kódu a HTML kódu spolu s CSS styly. Hlavním úkolem vrstvy je umožnit uživatelovi ovládat monitorování aplikací a reprezentovat aktuální stav aplikací. Mezi důležité body vrstvy zobrazení patří dynamická podpora seznamů monitorujících aplikací. Tedy při startu serveru, Flask předá do HTML vrstvy pomocí funkce `render_template` dva parametry: počet aplikací pro monitorování a seznam s názvy těchto aplikací. Následně pomocí cyklické operace v HTML se definuje tabulka se seznamem podporovaných aplikací a pro každou z nich se nastaví ovládací tlačítka s odpovídajícím identifikátorem. Díky čemu, není potřeba žádným způsobem upravovat vrstvu zobrazeni pokud bude potřeba přidat další aplikaci pro monitorování, je to celkem dynamické řešení.

Výpis 5.35: Příklad použití knihovny pro notifikace

```
socket.on('alert', function (data) {
    toastr.options.timeOut = 3000; // 3s
    toastr.info(data);
});
```

Mezi další důležité body, patří podpora notifikace. V JavaScript kódu je potřeba nadefinovat funkci, která bude přijímat data pro notifikace ze strany Flask serveru, příklad takové funkce je uveden ve výpisu 5.35. Funkce přijímá textovou zprávu `data`, která je pak zobrazena uživatelovi ve formátu notifikace. Pro tyto účely se používá JavaScript knihovna `toastr` která umožňuje navíc nastavit čas, jak dlouho se bude zobrazovat notifikace ve webovém prohlížeči.

Výpis 5.36: Aktualizace stavu

```
socket.on('statusUpdate', function (data) {
    document.getElementById(data.appName + "_state")
        .innerHTML = data.state;
    document.getElementById(data.appName + "_msg")
        .innerHTML = data.message;
});
```

Samotná aktualizace stavu monitorující aplikace je řešena pomocí funkce, která naslouchá událost typu `statusUpdate` podle výpisu 5.36. Funkce přijímá na vstupu objekt `data`, který přichází ze Flask serveru a následně obsah vstupních dat je vypsán do odpovídajícího HTML pole podle identifikátoru, který se skládá z názvu aplikace a druhu informace které je určeno pro tohle pole.

5.4 Testovací provoz

Kapitola se věnuje zkušebnímu provozu a ověření funkčnosti řešení. Jsou zde uvedené jednotlivé kroky které jsou potřebné ke spuštění Kotlin backend serverů a Flask frontend serveru. Další část kapitoly uvádí postup testování a výsledky ze zkušebního provozu, v neposlední řadě je uveden popis infrastruktury na které se provádělo testování.

5.4.1 Zprovoznění agentů

Agentem se rozumí běžící instance Kotlin backend serveru která provádí dohled nad jednou nebo více aplikací. Pro zprovoznění agenta který bude monitorovat aplikaci je potřeba spustit Kotlin backend server, na stejném zařízení, kde běží aplikace pro dohled. Pro úspěšné fungování celého řešení musí být splněny následující požadavky:

- na cílovém zařízení má být nainstalovaná Java verze 11
- pokud se jedná o dohled Docker aplikace, server musí běžet pod uživatelem s dostačujícím oprávněním aby mohl využívat služby Docker, nebo stačí spustit server jako sudo

- cílové zařízení musí mít správné směrování a funkční síťovou komunikaci s Flask serverem obousměrně

Pro spuštění agenta je potřeba mít připravený `jar` soubor, který se dá vytvořit sestavením projektu pomocí příkazu podle výpisu 5.37.

Výpis 5.37: Příkaz pro sestavení projektu

```
./gradlew clean build
```

Následně požadovaný `jar` soubor bude umístěn ve složce `build/libs` s názvem `linuxapi` a odpovídajícím číslem verze. Na cílovém zařízení, výše uvedený `jar` soubor se spouští příkazem podle výpisu 5.38. Zde jako argument se předává cesta do konfiguračního souboru pro určitou instancí agenta. Součástí projektu jsou předem připravené konfigurační soubory pro každou instanci agenta, které obsahují specifické parametry jako příkazy ke spuštění monitorující aplikace, API klíč, certifikát a další. Tyto konfigurace jsou umístěné ve složce `agent-configs`.

Výpis 5.38: Příklad spuštění backend serveru

```
java -jar linuxapi-0.0.1.jar --spring.config.location=file
://path/to/application.properties
```

Po úspěšném naběhnutí serveru, REST API jsou zveřejněné na portu 8080 který lze změnit v odpovídajícím konfiguračním souboru.

5.4.2 Zprovoznění Flask serveru

Dalším krokem je spuštění Flask serveru. Cílová stanice musí mít instalovaný Python verze 3 a taky instalované knihovny pro frontend server. Seznam knihoven je uveden v souboru `requirements.txt`, které lze jednoduše nainstalovat hromadně pomocí příkazu dle výpisu 5.39.

Výpis 5.39: Příklad instalace knihoven pro Flask server

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

Jako další krok je potřeba nastavit IP adresy agentů pro odpovídající dohledové aplikace. Parametry lze nastavit v konfiguračním souboru `application.properties` který se nachází v projektové složce `src`. Následně je možné spustit Flask server příkazem podle výpisu 5.40.

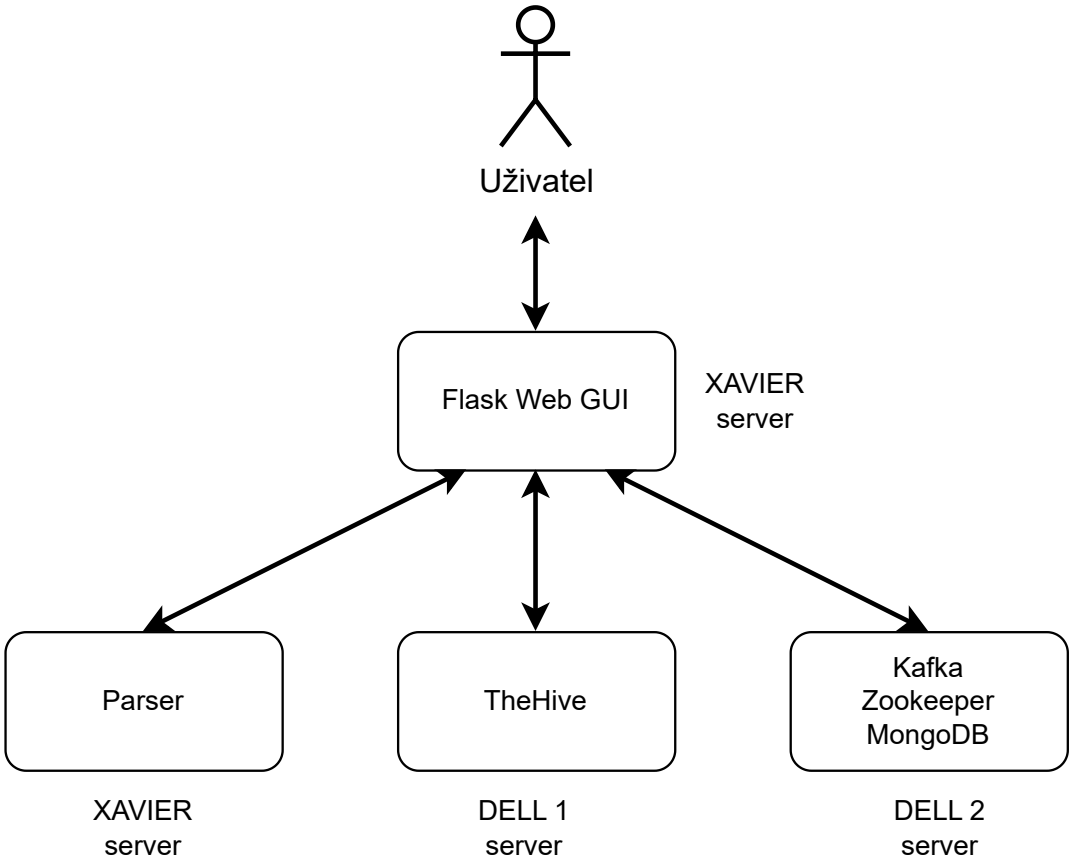
Výpis 5.40: Příklad spuštění Flask serveru

```
python3 app.py
```

Ve výchozím nastavení server běží na adrese `localhost` a portu 5000. Po otevření webové stránky se zobrazí hlavní stránka aplikace.

5.4.3 Testovací prostředí

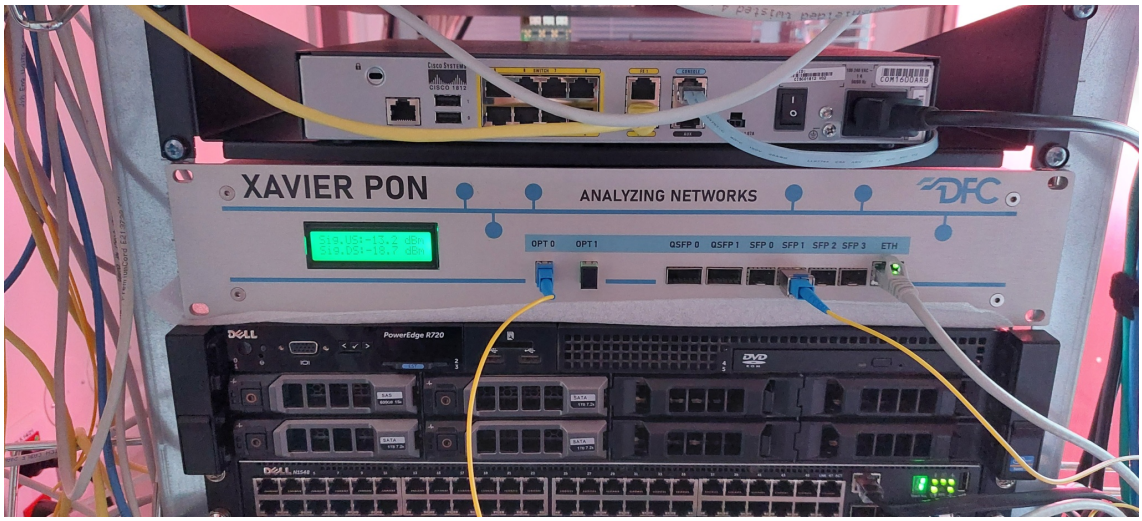
Implementované řešení provádí dohled pro celém 5 aplikací, několik z nich jsou nainstalované na různých zařízeních ale v rámci jedné sítě. Jedná se o následující aplikace: Parser, TheHive, Kafka, Zookeeper, MongoDB. Na schematickém obrázku 5.4 znázorněna testovací infrastruktura, která se skládá ze dvou Dell serverů a jednoho Xavier serveru, kde všechna zařízení jsou propojené do jedné společné sítě. Na obrázku 5.5 je pak vidět tyto zařízení zapojené do racku v serverovně. Každé zařízení používá jako operační systém Linux. Vzhledem k tomu, že se používají 3 zařízení, bylo potřeba nainstalovat 3 agenty a jeden Flask server, přičemž komunikace probíhá přes HTTPS protokol.



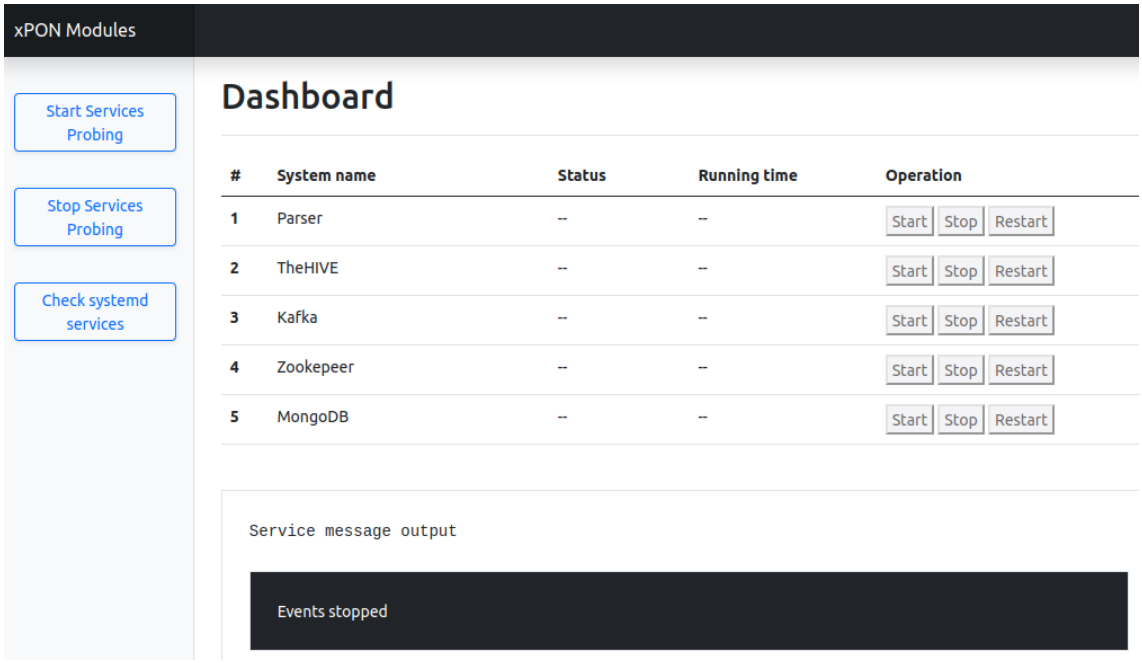
Obr. 5.4: Schéma testovacího prostředí

5.4.4 Přehled řešení a testování

Po načtení webové stránky `localhost:5000` zobrazí se úvodní stránka, kterou lze vidět na obrázku 5.6. Struktura stránky se skládá z ovládacího panelu vlevo, kde



Obr. 5.5: Testovací zařízení

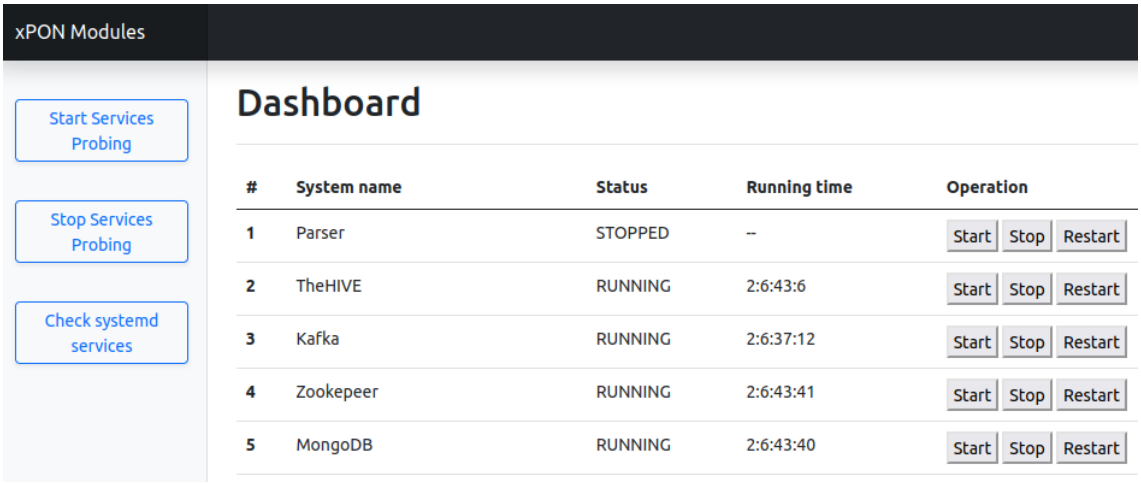


Obr. 5.6: Hlavní stránka aplikace

jsou umístěné tlačítka pro spuštění a zastavení monitorování všech aplikací a servisní tlačítko, které provádí kontrolu systémových služeb na zařízení, kde běží Flask server. Uprostřed je umístěna tabulka se seznamem podporovaných aplikací, kde se zobrazuje název aplikace, aktuální stav a čas, který určuje jak dlouho běží program. Navíc ke každé aplikace patří ovládací tlačítka ke spuštění, zastavení a restartování. Dole je uvedeno pole pro případný výpis systémových služeb nebo další funkciona-

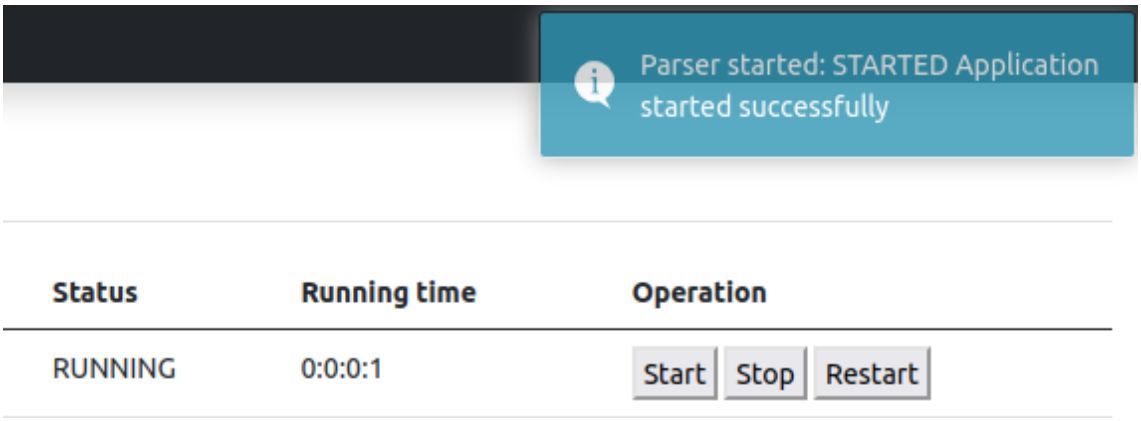
lity pro případný budoucí vývoj projektu, také vhodné pro výpis debug informací během vývoje.

Teď je možné spustit monitorování všech programů a hned je vidět stav monitorujících aplikací podle obrázku 5.7. Zároveň je vidět, že ovládácí tlačítka pro každou aplikaci jsou teď aktivní.



Obr. 5.7: Příklad běžícího monitorování

Jako první test, je možné spustit program Parser, pomocí tlačítka `Start`. Následně se zobrazí informační notifikace, o tom že spuštění programu proběhlo úspěšné a aktualizuje se stav programu v tabulce. Tato zpráva obsahuje text, který přišel v odpovědi na REST požadavek do backend serveru. Ukázku lze vidět na obrázku 5.8.



Obr. 5.8: Spuštění programu Parser

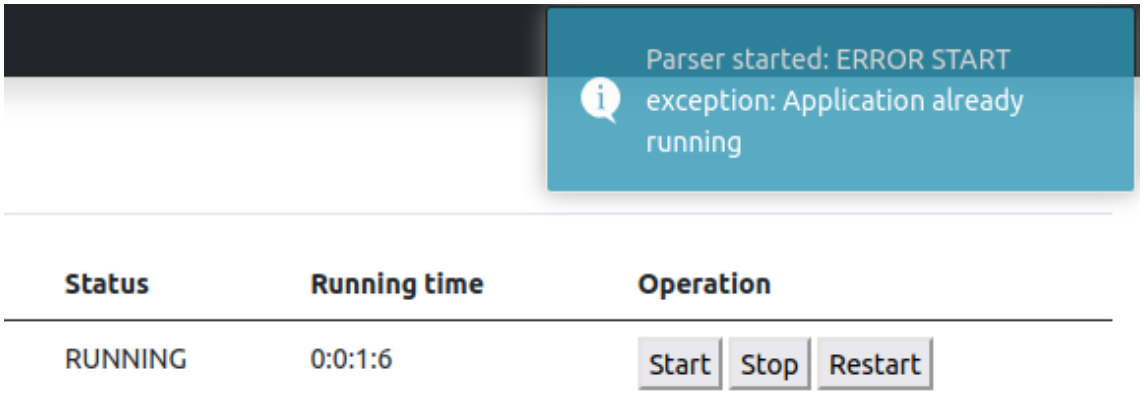
Následně, na straně backend serveru lze vidět podle logu příchozí požadavky na zjištění stavu programu Parser od určité IP adresy, což je adresa Flask serveru.

Příklad je znázorněn na obrázku 5.9. Zde je vidět, že program Parser aktuálně už běží a má přiděleno PID 1216689, zároveň je vidět i čas běhu programu, což je součást výpisů příkazu `ps`.

```
11:19:32  INFO 5233 - RequestLoggingFilter : Request from 10.0.0.27 => HTTP/1.1 GET /api/manage/PARSER/state
11:19:32  INFO 5233 - CommandLineUtils     : Execute command: ps -p 1216689
    PID TTY          TIME CMD
1216689 pts/1    00:00:00 XDMA_parser
11:19:35  INFO 5233 - RequestLoggingFilter : Request from 10.0.0.27 => HTTP/1.1 GET /api/manage/PARSER/state
11:19:35  INFO 5233 - CommandLineUtils     : Execute command: ps -p 1216689
    PID TTY          TIME CMD
1216689 pts/1    00:00:02 XDMA_parser
11:19:38  INFO 5233 - RequestLoggingFilter : Request from 10.0.0.27 => HTTP/1.1 GET /api/manage/PARSER/state
11:19:38  INFO 5233 - CommandLineUtils     : Execute command: ps -p 1216689
    PID TTY          TIME CMD
1216689 pts/1    00:00:04 XDMA_parser
11:19:41  INFO 5233 - RequestLoggingFilter : Request from 10.0.0.27 => HTTP/1.1 GET /api/manage/PARSER/state
11:19:41  INFO 5233 - CommandLineUtils     : Execute command: ps -p 1216689
    PID TTY          TIME CMD
1216689 pts/1    00:00:06 XDMA_parser
11:19:43  INFO 5233 - RequestLoggingFilter : Request from 10.0.0.27 => HTTP/1.1 PUT /api/manage/PARSER/STOP
11:19:43  INFO 5233 - CommandLineUtils     : Execute command: ps -p 1216689
    PID TTY          TIME CMD
1216689 pts/1    00:00:07 XDMA_parser
```

Obr. 5.9: Příklad logu na straně backend serveru

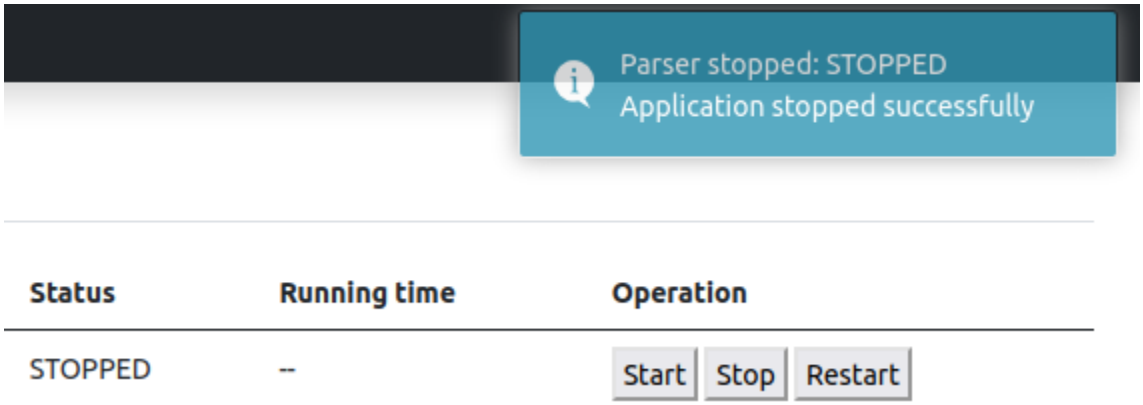
V případě, že uživatel zkusí spustit program, který již běží, backend server tento stav odchytí a odesílá uživateli odpovídající odpověď s chybovým stavem. Příklad odpovědi je znázorněn na obrázku 5.10.



Obr. 5.10: Spuštění programů Parser podruhé

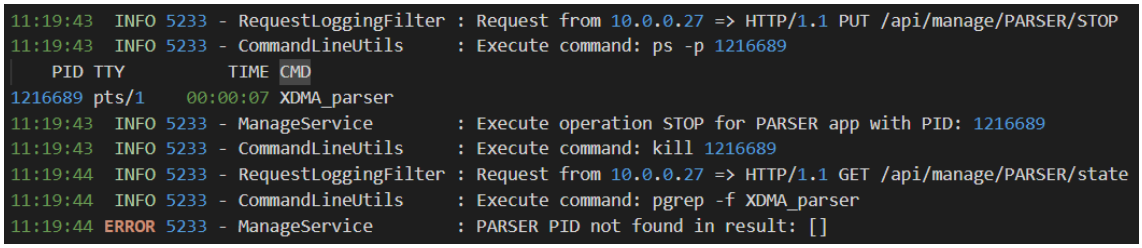
Po zastavení programu se zobrazí uživateli odpovídající informační hlášení, které je zobrazeno na obrázku 5.11. Následně v logu na backend serveru lze vidět příchozí API požadavek pro zastavení určené aplikace. Z logů lze také přečíst, že program byl zastaven pomocí příkazu `kill`. Později, pokud uživatel použije tlačítko pro zastavení programu podruhé, tak při dalším požadavku na zjištění stavu programu už PID pro

Parser nelze najít a aplikace na tento stav reaguje zobrazením hlášení, že program neběží. Příklad výpisu daného logu vztahujícího se na uvedený stav je znázorněn na obrázku 5.12. Tento výpis je však zobrazen pouze administrátorovi aplikace.



Obr. 5.11: Zastavení programu Parser

Aplikace Parser je klasický program v jazyce C, který se spouští pomocí příkazové řádky. Pro ověření funkcionality dohledu nad Docker aplikaci je možné zvolit jeden z progamu jako Kafka, Zookeeper nebo MongoDB které běží v Dockeru.



Obr. 5.12: Zastavení programů Parser na straně backendu

V případě, že uživatel zvolí Docker aplikaci, tak operace nad ní bude trvat o něco déle, proto na frontendu se zobrazí animace zpracování požadavků, jako na obrázku 5.13. V tomto případě se jedná o aplikaci Kafka. Výpis logu na straně backend serveru je uveden na obrázku 5.14.

Zde je vidět příchozí HTTP PUT požadavek k zastavení kontejneru Kafka. Následně podle identifikátoru kontejneru se zjišťuje, jestli program běží, podle logu je vidět, že Kafka běží už 2 dny, zároveň se to shoduje s časem běhu, který lze vidět v tabulce, kde čas běhu programu se uvádí ve formátu `dny:hodiny:minuty:vteřiny`. Pak se provede zastavení kontejneru příkazem `sudo docker stop kafka`. Při dalším požadavku na zjištění stavu aplikace, je vidět, že takový kontejner neexistuje, protože výsledek hledání je `null` hodnota.

#	System name	Status	Running time	Operation
1	Parser	STOPPED	--	Start Stop Restart
2	TheHIVE	RUNNING	2:6:46:58	Start Stop Restart
3	Kafka	RUNNING	2:6:41:2	Start Stop Restart
4	Zookepeer	RUNNING	2:6:47:34	Start Stop Restart
5	MongoDB	RUNNING	2:6:47:33	Start Stop Restart

Obr. 5.13: Zastavení programů Kafka

```
16:07:51  INFO 1308585 RequestLoggingFilter  : Request from 10.0.0.27 => HTTP/1.1 PUT /api/manage/KAFKA/STOP
16:07:51  INFO 1308585 CommandLineUtils      : Execute command: docker ps | grep 9136f26aa627
16:07:51  INFO 1308585 CommandLineUtils      : Output: 9136f26aa627   vutbr.cz/mvcr2/kafka:local "bin/kafka-server-st…"
    5 months ago   Up 2 days   0.0.0.0:9092-9093->9092-9093/tcp, :::9092-9093->9092-9093/tcp   kafka
16:07:51  INFO 1308585 ManageService          : Execute operation STOP for KAFKA app with Container ID: 9136f26aa627
16:07:51  INFO 1308585 CommandLineUtils      : Execute command: sudo docker stop kafka
16:07:54  INFO 1308585 CommandLineUtils      : Output: kafka
16:07:55  INFO 1308585 RequestLoggingFilter  : Request from 10.0.0.27 => HTTP/1.1 GET /api/manage/KAFKA/state
16:07:55  INFO 1308585 CommandLineUtils      : Execute command: docker ps | grep vutbr.cz/mvcr2/kafka
16:07:55  INFO 1308585 CommandLineUtils      : Output: null
```

Obr. 5.14: Zastavení programů Kafka na straně backendu

#	System name	Status	Running time	Operation
1	Parser	STOPPED	--	Start Stop Restart
2	TheHIVE	STOPPED	--	Start Stop Restart
3	Kafka	STOPPED	--	Start Stop Restart
4	Zookepeer	STOPPED	--	Start Stop Restart
5	MongoDB	STOPPED	--	Start Stop Restart

Obr. 5.15: Zastavení všech programů

Jako poslední testy je potřeba vyzkoušet zastavení všech programů a zastavení celého monitorování. Po zastavení dohledových aplikaci monitorovací tabulka vypadá jako na obrázku 5.15.

Následně pomocí tlačítek ke spuštění programu, lve ověřit, že všechny programy úspěšně se spustí a čas běhu programu se aktualizuje. Tento stav je znázorněn na obrázku 5.16.

#	System name	Status	Running time	Operation
1	Parser	RUNNING	0:0:0:3	Start Stop Restart
2	TheHIVE	RUNNING	0:0:0:31	Start Stop Restart
3	Kafka	RUNNING	0:0:0:26	Start Stop Restart
4	Zookepeer	RUNNING	0:0:0:26	Start Stop Restart
5	MongoDB	RUNNING	0:0:0:23	Start Stop Restart

Obr. 5.16: Spuštění všech programů

#	System name	Status	Running time	Operation
1	Parser	--	--	Start Stop Restart
2	TheHIVE	--	--	Start Stop Restart
3	Kafka	--	--	Start Stop Restart
4	Zookepeer	--	--	Start Stop Restart
5	MongoDB	--	--	Start Stop Restart

Obr. 5.17: Zastavení monitorování

Pomocí ovládacího tlačítka `Stop Services Probing`, je možné zastavit monitorování jako takové, to znamená, že Flask server už nebude odesílat žádné požadavky na agenty. Zároveň všechny záznamy v monitorovací tabulce se vrátí na výchozí hodnotu, což je vidět na obrázku 5.17.

Závěr

Diplomová práce se zabývá problematikou monitorování softwarových aplikací v operačním systému Linux. V rámci teoretické části jsou popsané zásady pro monitorování a ovládání procesů v operačním systému Linux. Následně jsou probrané frameworky Spring a Flask a jejích možností pro řešení zadání práce. V neposlední řadě jsou popsané způsoby dokumentace a specifikace API a také uvedené nejčastěji vyskytující bezpečnostní problémy při návrhu API.

V rámci praktické části bylo implementováno backend a frontend řešení. Na straně backendu byl použit programovací jazyk Kotlin spolu s frameworkem Spring. Backend se skládá z jednotlivých modulů které určitým způsobem mezi sebou spolupracují. Jsou zde implementované následující moduly: REST kontrolér, logovací modul, servisní modul, bezpečnosti modul, úložiště procesu, odchytávač chyb a modul pro práce s příkazovým řádkem. REST kontrolér odpovídá za definice API rozhraní a díky frameworku Spring dokáže přijímat příchozí požadavky a směřovat je do servisní třídy. V rámci servisní třídy probíhá samotné řešení problematiky, jsou zde zapojené další moduly a implementována řešení pro monitorování a ovládání procesu. Pro své účely servisní třída využívá úložiště procesů a modul pro práce s příkazovým řádkem (CL modul). Úložiště procesu funguje jako mapa, která je jedinečná v systému, kde klíčem je název aplikace a hodnota – identifikátor procesu PID nebo identifikátor Docker kontejneru. CL modul realizuje funkce pro spuštění příkazu do operačního systému jako nový proces, využívá systémovou třídy `Runtime`, díky čemu lze spustit prakticky jakýkoliv příkaz a zjistit identifikátor procesu. Další součástí backendu je logovací modul a odchytávač chyb. Logovací modul funguje na základě filtru framewroku Spring, což umožňuje odchytit příchozí požadavek a uložit užitečnou informaci jako IP adresa uživatele, typ požadavku, koncový bod API a použité parametry. Odchytávač chyb zajišťuje překlad systémových chyb do uživatelské zprávy, která je pak vrácena v odpovědi na požadavek, který způsobil chybu, tím se dá vyhnout odeslání citlivých údajů do uživatele, jako například stack trace chyby. V neposlední řadě je implementován bezpečnostní modul, který zajišťuje autorizaci příchozích požadavků na základě API klíče. Tento API klíč musí být uveden v záhlaví požadavku, který odesílá Flask server. Bezpečnostní modul povolí zpracování jen takového požadavku, který obsahuje platný API klíč, jinak je požadavek zamítnuty.

Frontend část je postavena na frameworku Flask a skládá se z Flask serveru v Python a vrstvy zobrazení která se skládá z JavaScript kódu a HTML spolu s CSS styly a je určena pro zobrazení výsledku. Pro předávání informace mezi serverem a vrstvou zobrazení je využita knihovna `socketio`. Tato knihovna funguje na základě událostí, tedy Flask server a HTML šablona mohou mezi sebou posílat události

a tím vyměňovat informace bez nutnosti načítat webovou stránku. Na základě těchto událostí, uživatel může pomocí tlačítka „start“ odeslat událost ke spuštění programu do Flask serveru. Tento server pak odešle REST API požadavek na odpovídající backend server který provede spuštění aplikace a vrátí zprávu se stavem provedení operace. Tato zpráva je následně upravena z Json formátu do textu a je zobrazena uživatelovi jako notifikace na webové stránce. Obdobným způsobem fungují i ostatní operace které jsou k dispozici u uživatele.

Poslední část práce se věnuje testovacímu provozu, zde byl uveden popis testovací infrastruktury který se skládá ze třech serverů a byla prakticky ověřena funkčnost řešení. Implementované řešení podporuje monitorování 5 aplikací a to jak systémových tak i Docker kontejnery.

Literatura

[1] BALL, Corey. *Hacking Apis : Breaking Web Application Programming Interfaces.* San Francisco: No Starch Press, 2022. ISBN 1718502443.

[2] COSMINA, Iuliana, Rob HARROP, Clarence HO a Chris SCHAEFER. *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools.* 5th ed. 2017. Imprint: Apress, 2017. ISBN 9781484228081.

[3] DUCKETT, Jon. *HTML and CSS: Design and Build Websites.* 2011. Nashville, TN: John Wiley & Sons. ISBN 9781118008188.

[4] GASPAR, Daniel, STOUFFER, Jack. *Mastering Flask Web Development : Build Enterprise-Grade Scalable Python Web Applications.* 2nd Edition. Birmingham: Packt Publishing, 2018. ISBN 1788995406.

[5] GRIFFITHS, Dawn, GRIFFITHS David. *Head First Kotlin : A Brain-Friendly Guide.* Sebastopol CA: O'Reilly; 2019. ISBN 1491996692.

[6] GRINBERG, Miguel. *Flask Web Development : Developing Web Applications with Python.* Second ed. Sebastopol CA: O'Reilly, 2018. ISBN 1491991739.

[7] HAVERBEKE, Marijn. *Eloquent JavaScript: A modern introduction to programming.* 3rd ed. 2018. San Francisco, CA: No Starch Press. ISBN 9781593279509.

[8] HECKLER, Mark. *Spring Boot: up and running : building cloud native Java and Kotlin applications.* Boston [MA]: O'Reilly, 2021. ISBN 9781492076988.

[9] JEMEROV, Dmitry, ISAKOVA, Svetlana. *Kotlin in Action.* Shelter Island NY: Manning Publications; 2017. ISBN 9781617293290.

[10] KANE, Sean, MATTHIAS, Karl. *Docker : Up & Running.* 2nd ed. O'Reilly Media, 2018. ISBN 9781491917572.

[11] LATHKAR, Malhar. *Building web apps with python and flask: Learn to develop and deploy responsive RESTful web applications using flask framework.* 2021. New Delhi, India: BPB Publications. ISBN 9789389898835.

[12] LUTZ, Mark. *Learning Python.* 2013. Sebastopol, CA: O'Reilly Media. ISBN 9781449355739.

[13] MADDEN, Neil. *Api Security in Action.* 1st ed. Erscheinungsort nicht ermittelbar: Manning Publications, 2021. ISBN 1617296023.

[14] MASSE, Mark. *Rest Api Design Rulebook.* Sebastopol CA: O'Reilly, 2012. ISBN 1449310508.

[15] MAUERER, Wolfgang. *Professional Linux Kernel Architecture.* Indianapolis IN: Wiley Pub; 2008. ISBN: 9780470343432.

[16] MCDONALD, Malcolm. *Web security for developers: Real threats, practical defense.* 2020. San Francisco, CA: No Starch Press. ISBN 9781593279943.

[17] NICKOLOFF, Jeff, KUENZLI, Stephen and FISHER, Bret. *Docker in Action.* 2022. New York, NY: Manning Publications. ISBN 9781617294761.

[18] *OWASP API Security Project*, [online], dostupné z: https://owasp.org/www-project-api-security/

[19] RELAN, Kunal. *Building Rest Apis with Flask : Create Python Web Services with Mysql.* New York: Apress, 2019. ISBN 1484250214.

[20] RICHARDSON, Leonard, AMUNDSEN, Mike. *Restful Web Apis.* First edition second release ed. Beijing: O'Reilly, 2015. ISBN 1449358063.

[21] SHOTTS, William E. *The Linux command line: a complete introduction.* San Francisco: No Starch Press, 2012. ISBN 9781593273897.

[22] SIRIWARDENA, Prabath. *Advanced Api Security : Oauth 2. 0 and Beyond.* 2nd ed. Berkeley CA: Apress L.P, 2020. ISBN 1484220498.

[23] TANENBAUM, Andrew S. *Modern operating systems.* 2nd ed. Upper Saddle River: Prentice Hall, 2001. ISBN 0130313580.

[24] TURNBULL, James. *The Docker Book: Containerization is the new virtualization* 1st ed. Turnbull Press, 2014. ISBN 9780988820203.

[25] WALLS, Craig. *Spring Boot in action.* Shelter Island, NY: Manning Publications, 2016. ISBN 1617292540.

[26] WARD, Brian. *How Linux works: what every superuser should know.* 2nd edtion. San Francisco, CA: No Starch, 2015. ISBN 9781593275679.

Seznam symbolů a zkratek

ACL Access control list
API Application Programming Interface
CORS Cross Origin Resource Sharing
CSRF Cross-Site Request Forgery
CSS Cascading Style Sheets
CWD Current Working Directory
DoS Denial-of-Service
FD File Descriptor
HTML HyperText Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
HW Hardware
IPC Interprocess Communication
JDBC Java Database Connectivity
JSP Jakarta Server Pages
JVM Java Virtual Machine
MITM Man In The Middle
OAS Open API Specification
OWASP Open Web Application Security Project
PID Process Identification
POSIX Portable Operating System Interface
PPID Parent Process Identification
ReDoS Regular expression Denial of Service
REST Representational State Transfer
RPC Remote Procedure Call
SH Signal Handler
SIEM Security Information and Event Management
SOAP Simple Object Access Protocol
SQL Structured Query Language
TCP Transmission Control Protocol
UDP User Datagram Protocol
URI Uniform Resource Identifier
URL Uniform Resource Locator
UUID Universally Unique Identifier
XML Extensible Markup Language