FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

# Zadání bakalářské práce

| | |
|---|---|
| **Název:** | LightLog software pro zpracovaní logů aplikací |
| **Student:** | Dominik Dosoudil |
| **Vedoucí:** | Ing. Filip Glazar |
| **Studijní program:** | Informatika |
| **Obor / specializace:** | Webové a softwarové inženýrství, zaměření Softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | do konce letního semestru 2022/2023 |

## Pokyny pro vypracování

Cílem této bakalářské práce je navhrnout a implementovat software pro zpracování a následnou prezentaci logů aplikací běžících na linuxovém serveru. Aplikace umožní ostatnícm aplikacím na serveru unifikované zpracování a uchovávání zpráv. Zároveň usnadní správcům serveru nahlížení do sledovaných logů.

Software bude navrhnut jako mikro služba, která nabízí rozhraní pro zápis zprávy pomocí různých protokolů. Zaměřte se především na NATS a případně na obecnější protokoly jako je například http.

Pro implementaci vyberte vhodné technologie a konkrétní programovací jazyk, který je vhodný pro tento případ použití.

1. Specifikujte požadavky na software
2. Na základě požadavků zvolte vhodné technologie pro implementaci
3. Proveďte softwarový návrh
4. Implementujte prototyp aplikace
5. Softwarové řešení podrobte vhodným testům např. Unit testy, zátěžové testy atd.
6. Prototyp aplikace nasaďte do produkčního prostředí a zhodnoťte výsledky vaší práce

---

*Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 15. února 2021 v Praze.*

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

# LightLog software pro zpracování logů aplikací

***Dominik Dosoudil***

Katedra softwarového inženýrství
Vedoucí práce: Ing. Filip Glazar

13. května 2021

# Poděkování

V první řadě bych chtěl poděkovat vedoucímu mé bakalářské práce Ing. Filipu Glazarovi, který se mnou diskutoval a pomohl s klíčovými částmi praktické i teoretické části. Dále bych chtěl poděkovat firmě INSOFT s.r.o. za umožnění psát tuto aplikaci jako bakalářskou práci, volné ruce ve volbě technologií a konzultace některých problematik. Nakonec děkuji své přítelkyni a svým spolužákům za rady a názory zejména ohledně textu práce a své rodině za podporu.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na jejímž základě se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ustanovení § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisu.

V Praze dne 13. května 2021 . . . . . .

### Odkaz na tuto práci

Dosoudil, Dominik. *LightLog software pro zpracování logů aplikací.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

# Abstrakt

Cílem bakalářské práce je navržení a implementace paměťově a procesorově efektivního agregátoru log zpráv aplikací systému.

Nejprve je obecně popsán význam logování a jeho stav v operačním systému Linux, následně je popsána konkurence, poté analýza a výběr vhodných technologií a programovacího jazyka. Nakonec je vysvětlena architektura a samotná realizace a implementace aplikace.

Výsledkem práce je funkční prototyp uspokojující zadané požadavky. Přináší zpřehlednění monitorování stavu aplikací administrátorům.

**Klíčová slova** Rust, asynchronní programování, logování aplikací, monitorování aplikací, agregace logů, modulární systém, hexagon architektura

# Abstract

The aim of this thesis is to design and implement memory and performance efficient log agregator for applications of software stack.

Logging theory is described at the beggining of thesis. Next section compares existing solutions for similar problems. Then technological analysis and selection of appropriate programming language follows. Explanation of architecture and implementation of application is the last.

Result of this thesis is a working prototype that satisfies requirements. It brings better monitoring and clearer overview of application logs for administrators.

**Keywords** Rust, asynchronous programming, app logs, app monitoring, log aggregation, modular system, hexagon architecture

# Obsah

# Seznam obrázků

# Seznam tabulek

# Úvod

Logování aplikace je dnes takřka nutné, pokud chce její správce vědět o ději, stavu a chybách, které v aplikaci případně nastanou. Zároveň tak administrátoři mohou zjistit, co se v aplikaci dělo a co k chybě mohlo vést. Jedna z prvních činností, které administrátor provede po nahlášení problému, je právě otevření a přečtení log souboru. To platí pro malé programy i větší systémy sestávající z několika spolupracujících aplikací.

Dokud administrátoři spravují jednu aplikaci, obvykle si vystačí s textovým souborem a textovými nástroji, které poskytuje většina Linux distribucí. Ve chvíli, kdy ovšem spravují větší systém a každá aplikace má svůj vlastní log soubor a navíc v různých formátech, hledání chyb začne být obtížné a nepřehledné. Výsledek této práce vyřeší tento problém sjednocením záznamů a usnadněním jejich prohledávání. Vzhledem k cílení na jednoduchost a rychlost je LightLog určen zejména pro správce středně velkých systémů komunikujících aplikací.

Téma jsem si zvolil, protože se zmíněnými obtížemi jsem se setkal v práci. Často hledáme souvislosti mezi děním v několika různých aplikací, což obnáší neustálé kontrolování času zapsání řádků v několika souborech a porovnávání časových známek, abychom si rozmysleli, která událost nastala dříve, nebo později. Dále je obvykle obtížné z textu automatizovaně vyčíst nějaká data, protože logování do textového souboru neumožňuje ke zprávě přidat metadata. Z těchto důvodů jsem se rozhodl vytvořit řešení zmíněných problémů. Kromě toho navíc nebude nutné se přihlásit přímo na vzdálený server, protože vše bude dostupné přes webovou aplikaci.

Bakalářská práce obsahuje představení a srovnání konkurenčních aplikací, výběr vhodných technologií pro implementaci prototypu vlastního řešení a vývoj aplikace, která odstraní komplikace spojené s čtením log zpráv jiných aplikací. Dále obsahuje vysvětlení technického návrhu architektury aplikace a výsledky z testování v simulaci reálného provozu.

Kapitola 1

# Cíl práce

Cílem této bakalářské práce je vytvoření paměťově a procesorově nenáročné služby pro zpracovávání, uchovávání a prezentaci logů systémových aplikací běžících na linuxovém serveru.

Cílem teoretické části je rešerše odpovídajících technologií. Zejména výběr správného programovací jazyka, který je dostatečně efektivní a umožňuje bez větších komplikací implementaci jednotlivých komponent služby. To zahrnuje vyhledání knihoven, které mohou být použity namísto psaní funkcionalit od začátku.

Cíl praktické části lze rozdělit na serverovou část (dále jen backend) a klientskou část (dále jen frontend). Backend plně umožní zapisování logovaných zpráv skrz službu NATS (5.9), ale bude připravený na případné rozšíření. Zprávy bude zpětně poskytovat pomocí veřejného API (5.6) používajícího protokol HTTP (5.7), které umožní filtrování a stránkování. Dále také bude přenášet nové zprávy klientům připojeným přes WebSocket (5.8).

Frontend umožní přehledné zobrazování zpráv s možností použití filtrování a automatické načítání zpráv přicházejících ze serveru skrz zmíněný WebSocket.

Kapitola 2

# Motivace

Požadavek na implementaci aplikace LightLog vznikl ve firmě INSOFT s.r.o., která se zabývá vývojem a provozem systémů call center, které slouží pro telefonickou komunikaci mezi zákazníky a zaměstnanci nějaké firmy. Systém pro obsluhu takového call centra zahrnuje realizaci hovorů a jejich distribuci na agenty, prezentaci statistik v reálném čase, vytváření reportů nebo třeba ovládání ústředny pomocí operátorské aplikace.

Celý systém sestává z několika služeb, které spolu kooperují pomocí různých protokolů. Hlavní částí systému je jádro aplikace, jež se stará o zprostředkování hovorů, jejich směrování mezi částmi ústředny a operátory, udržování a ukládání stavu aplikace a vytváření událostí. Dalšími důležitými částmi jsou klientské aplikace běžící v prohlížečích agentů a tzv. *api proxy* překládající komunikační protokol jádra na protokol, jemuž rozumí klient.

Pro rychlost a někdy i kvůli zákaznické bezpečnostní politice se realizace tohoto systému obvykle provádí na fyzickém serveru umístěném přímo v geografické lokaci zákazníka. Díky tomu není potřeba upravovat firewall firmy a umožňovat systému ústředny speciální přístup do veřejné sítě. Tyto požadavky vedou vývoj ve firmě ke snaze minimalizovat nároky aplikací a potřebu škálovat.

Vzhledem k tomu, že fungování systému spoléhá na častou komunikaci mezi jednotlivými částmi, je potřeba zjišťovat dění a proběhlé zprávy mezi nimi, pokud nastane nějaká chyba. Z toho důvodu vznikla potřeba sdružovat zápisy dění těchto aplikací (logy) do jednoho místa kvůli přehlednosti. Z předchozího odstavce však plyne, že je potřeba použít maximálně efektivní řešení.

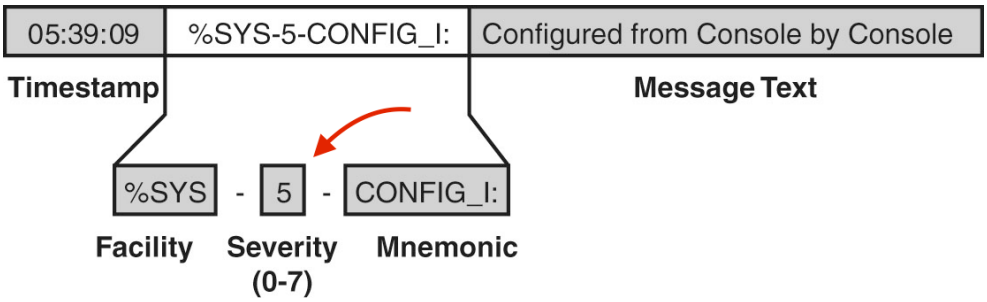
Kapitola 3

# Logování obecně

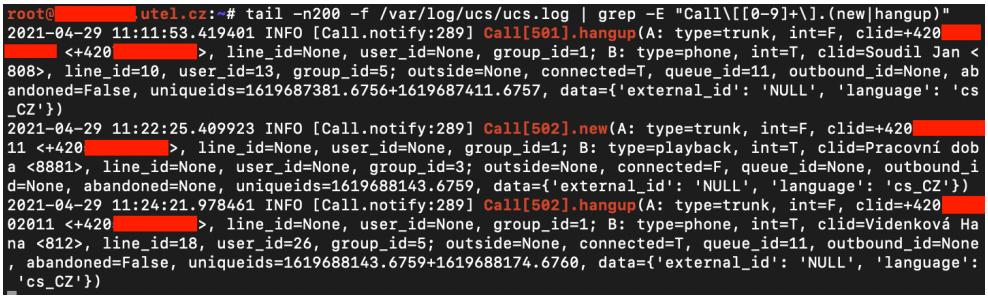
Logování aplikací je používané velmi často i v samotném operačním systému Linux, pro který je určena aplikace této bakalářské práce. Linux má proto připravené umístění pro log soubory. Toto místo je složka `/var/log`. [8]

Jedním z nejdůležitějších souborů je `/var/log/syslog`, který obsahuje zprávy téměř z celého systému včetně samotného operačního systému. Má textový formát a každý řádek je jeden záznam. Nicméně najít v něm nějaké informace může trvat dlouho, protože je obvykle velký. [9]

Formát syslog zpráv je znázorněn v obrázku 3.1. První položka *Timestamp* je časový údaj. Tento údaj je velmi důležitý pro zjištění, kdy se událost, kterou zpráva popisuje, udála. Další položka *Facility* představuje zdroj této zprávy. Může to být například jádro operačního systému, e-mailový server nebo síť. Dále následuje *Severity*, což je hodnota znázorňující důležitost zprávy. Zprávy mohou být od informativních po chybové. Předposlední položka *Mnemonic* je nepovinná a popisuje zprávu. Poslední *Message Text* je samotná zpráva popisující stav nebo dění v aplikaci. Tato poslední položka je nejdůležitější, protože dle ní můžeme zjistit, co se v systému odehrálo. [1]




Obrázek 3.1: Syslog formát zprávy [1]




Obrázek 3.2: Ukázka *tail* a *grep*

## 3.1 Nástroje

Jedny z nejužitečnějších nástrojů na prohlížení textových log souborů jsou `tail` a `grep`.

Nástroj `tail` nám umožňuje vypsat posledních několik řádků ze souboru. Pomocí přepínače `-f` (`--follow`) ale dokáže vypisovat i nově zapsané řádky. Můžeme tedy v reálném čase sledovat nové záznamy.

Program `grep` pak velmi dobře funguje na filtrování log zpráv. Jako parametr přijímá vyhledávaný textový řetězec. Použít lze například pomocí roury v kombinaci s `tail`.

Například příkaz `tail -f /var/log/syslog | grep error` vypíše z posledních 10 řádků záznamy, které obsahují slovo „error“, a následně bude vypisovat nové záznamy, které také obsahují „error“.

Ukázku těchto nástrojů můžeme vidět v obrázku 3.2, kde pomocí *tail* vypíšeme posledních 200 záznamů a všechny nově přibývající. Tyto vypsané záznamy vyfiltrujeme pomocí *grep* a regulárního výrazu. Grep nám tmavě červeně zvýrazní nalezenou shodu s regulárním výrazem.

Kapitola 4

# Analýza konkurence

Protože velkých distribuovaných systémů je v dnešní době nepřeberné množství a psaní mikro služeb je oblíbená architektura, která umožňuje velkým firmám rozdělit jejich systém na několik menších aplikací a získat více agilnosti [10], s problémem zpracování log zpráv a monitorováním systémů se již setkalo mnoho administrátorů a vývojářů. Proto lze také nalézt služby, které se problémem zabývají.

## 4.1 ELK Stack

ELK software stack sestává z komponent Elasticsearch, Logstash, a Kibana.

Zaměřuje se na agregaci logů a kromě toho nabízí i monitorování a vizualizaci jiných metrik. [11]

Nicméně Elasticsearch je velmi náročný na paměť RAM, protože je spuštěný v *JVM (Java virtual machine)*, což je běhové prostředí pro programy napsané v jazyku Java. Téměř polovina dostupné paměti by měla být přidělena JVM. [12]

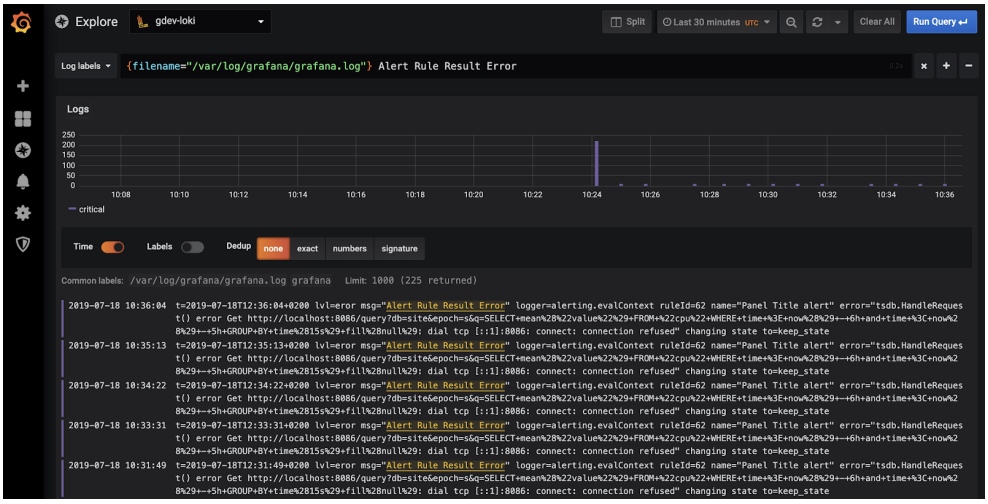
Proto se ELK hodí spíše pro větší systémy, které běží na více spolupracujících serverech a není problém mít dedikované servery přímo pro ELK.

## 4.2 Grafana Loki

Grafana je monitorovací systém umožňující zobrazování grafů znázorňujících stav systémů v čase (např. zátěž procesoru a paměti, počet dotazů a jiné metriky). Zároveň umožňuje notifikování administrátora. [13]

*„Grafana Loki je soubor komponent, ze kterého může být složen plnohodnotný log software stack.“* ([14] překlad autora)

Na obrázku 4.1 můžeme vidět příklad zobrazení historie logů. Kromě zpráv samotných můžeme vidět i políčko pro filtrování a sloupcový diagram počtu logů za určitý čas.




Obrázek 4.1: Grafana Loki Log [2]

Existuje několik předpřipravených klientů pro specifické případy použití (například monitorování docker kontejnerů nebo AWS Lambda serverů, což je služba od firmy Amazon Web Services umožňující spouštění kódu bez potřeby administrace serveru), nicméně pouze HTTP API je univerzální. [15]

Z toho důvodu není možné použít NATS pro zasílání log zpráv.

## 4.3 TICK-stack

Podobně jako Grafana umožňuje *TICK-stack* ukládání metrik, nahlížení do nich a notifikování. [16] Jen v tomto případě je každá část rozdělená do různých částí a spolu vytváří tzv. *software stack*.

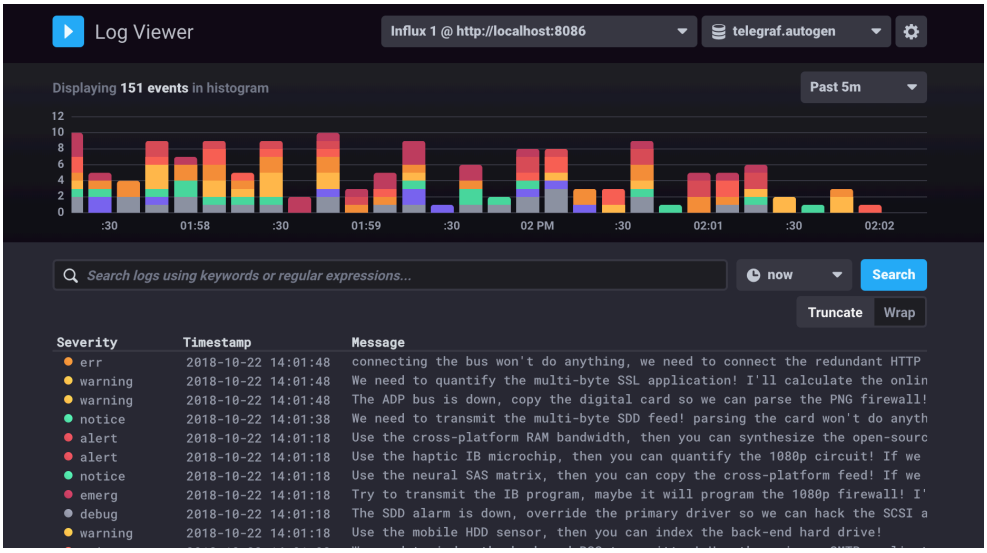
Obrázek 4.2 ukazuje *Chronograf Log Viewer* a jeho funkce zobrazování log zpráv. Log sestává pouze ze 3 částí: vážnost zprávy, čas vzniku a samotná zpráva.

Filtrování nabízí zobrazení zpráv pouze z nějakého času, zpráv odpovídajících regulárnímu výrazu nebo zpráv odpovídajících určitému klíčovému slovu. Klíčové slovo může být *severity*, *facility* nebo jiný štítek přiložený k log zprávě. [17]

TICK se ovšem zaměřuje primárně na ukládání a vizualizaci číselných hodnot v čase, které lze zobrazit v grafech. Příkladem může být cena akcie. [18]

## 4.4 Závěr analýzy

Ze zmíněných poznatků vyplývá, že ani jedna z probraných technologií není plně vhodná pro dané požadavky zmíněné v kapitole 1.




Obrázek 4.2: Chronograf Log View [3]

Kapitola 5

# Analýza technologií

Základní požadavky na aplikaci LightLog jsou shromažďování, ukládání a prezentování log zpráv. Od toho se odvíjí i analýza technologií.

## 5.1 Výběr programovacího jazyka pro backend

Backend je serverová část aplikace a v tomto případě zároveň hlavním jádrem projektu. Její zodpovědností je umožnit skrze libovolné podporované rozhraní (NATS, HTTP, ...) zapisovat log zprávy. Dalším úkolem této služby je ukládání přijatých log zpráv do úložiště, které může být také vybrané libovolně z podporovaných možností. A nakonec je třeba tyto zprávy prezentovat uživateli. Proto je třeba umožnit zpětné získání uložených zpráv a také okamžité obdržení zpráv nových.

Jelikož se očekává, že aplikace nebude náročná, je potřeba vybrat vhodný jazyk, který bude rychlý a bude mít nízké nároky na operační paměť. Zároveň je ale potřeba vybrat jazyk, ve kterém se bude dobře pracovat a aplikace v něm napsaná bude udržitelná.

## 5.2 Rychlost

Z dat v tabulce 5.1, která srovnává výkon programovacích jazyků C++, Java a Rust, můžeme vyčíst, že v rychlosti běhu programu je srovnatelně stejně efektivní jazyk C++ a Rust. O potřebné paměti platí totéž. Oproti tomu jazyk Java má občas podobný čas běhu a jindy znatelně vyšší jak můžeme vidět v prvních třech testech. Spotřebovaná paměť jazyka Java je ovšem mnohem vyšší, zejména u posledních čtyř testů.

To je nepřívětivé k požadavku na nízkou náročnost aplikace, a proto jazyk Java není v tomto ohledu vhodný pro implementaci aplikace LightLog.

Tabulka 5.1: Srovnání rychlostí a paměťové náročnosti programovacích jazyků C++, Java a Rust [6], [7]

| **Test** | **Jazyk** | **Čas** | **Paměť** |
|---|---|---|---|
| regex-redux | C++ | 1,08 | 203 816 |
| | Java | 5,58 | 985 696 |
| | Rust | **0,79** | **143 672** |
| mandelbrot | C++ | **0,84** | 34 604 |
| | Java | 4,15 | 69 136 |
| | Rust | 0,92 | **32 588** |
| k-nucleotide | C++ | **1,95** | **156 372** |
| | Java | 4,98 | 356 904 |
| | Rust | 2,75 | 159 068 |
| reverse-complement | C++ | 0,63 | 499 704 |
| | Java | 1,54 | 670 924 |
| | Rust | **0,45** | **499 100** |
| binary-trees | C++ | **1,04** | **154 824** |
| | Java | 2,48 | 1 722 848 |
| | Rust | **1,04** | 198 844 |
| spectral-norm | C++ | 0,72 | **1 216** |
| | Java | 1,63 | 39 304 |
| | Rust | **0,71** | 2 668 |
| fannkuch-redux | C++ | **4,91** | 1 912 |
| | Java | 10,64 | 35 192 |
| | Rust | 7,13 | **1 176** |
| n-body | C++ | 4,09 | 1 740 |
| | Java | 6,74 | 35 844 |
| | Rust | **3,32** | **844** |
| fasta | C++ | 0,78 | 2 468 |
| | Java | 1,21 | 44 620 |
| | Rust | **0,77** | **1 884** |
| pidigits | C++ | **0,60** | 4 944 |
| | Java | 0,93 | 36 552 |
| | Rust | 0,88 | **2 980** |

## 5.3 Stack a Heap

Pro lepší porozumění následující kapitoly si vysvětlíme pojmy *stack* (zásobník) a *heap* (halda). Tyto pojmy můžeme vysvětlovat z pohledu ukládání hodnot v paměti a z pohledu datových struktur. kapitole nás bude zajímat pohled ukládání dat v paměti.

*Stack* i *heap* se nachází v paměti RAM. [19]

### 5.3.1 Stack

*Stack* je rychlejší, ale velikostně omezená paměť programu. Slouží pouze pro uchování statických proměnných. [19]

Pořadí alokovaných dat je ve tvaru *LIFO*. To znamená, že poslední alokovaná data budou první uvolněná. [19]

Velikost této paměti je známá při kompilaci. Alokace probíhá při zavolání funkce. V této chvíli se alokují proměnné této funkce a v okamžiku, kdy je funkce dokončena, se paměť uvolní. [20]

### 5.3.2 Heap

Velikost této paměti je omezená pouze velikostí paměti RAM. Naopak je ale trochu pomalejší. Data na *heap* se alokují dynamicky za běhu programu. Můžeme je alokovat a uvolnit kdykoliv za život programu. [19] Práce s *heap* je obecně považována za méně bezpečnou, protože alokaci a uvolňování může více či méně (záleží na programovacím jazyku viz následující kapitoly) ovlivnit programátor. To může vést k únikům paměti, pokud není správně uvolněna. [20]

## 5.4 Správa paměti

Jeden z dalších aspektů, které různé jazyky řeší různými způsoby, je práce s pamětí.

*„Počítačové programy potřebují alokovat paměť pro uložení hodnot a datových struktur.“* [21] Může se stát, že program nezachází s alokovanou pamětí správně a neuvolňuje ji ve chvíli, kdy již není potřebná. Pokud tento problém nastane v části programu, která se za běhu opakuje, znamená to, že program kumuluje stále více paměti. V nejhorším případě tak program zabere veškerou dostupnou paměť počítače. [21] To je problém nejen pro proces samotný, ale i ostatní souběžně běžící procesy, jelikož nemohou dále pokračovat ve své práci, pokud náhle začnou potřebovat paměť, nebo dokonce spadnou, protože neočekávají, že by paměť nebyla dostupná.

Problém alokace a hlavně správného uvolnění paměti některé jazyky řeší samy (automatická správa [21]) a u některých tato zodpovědnost zůstává na programátorovi (manuální správa [21]).

### 5.4.1 Manuální

Vhodným příkladem manuální správy paměti je jazyk C++, kde programátor zodpovídá za to, že programu přidělená paměť operačním systémem bude uvolněna. K alokaci se používá klíčové slovo *new* a k uvolnění klíčové slovo *delete*.

V ukázce kódu 5.1 můžeme vidět dynamickou alokaci datového typu *int*. Na prvním řádku pouze inicializujeme proměnnou typu ukazatel na *int*. Tento ukazatel ukazuje na alokovaný prostor v paměti. Na dalším řádku pomocí klíčového slova *new* alokujeme prostor v paměti o velikosti datového typu *int* (4 B [22]). Na třetím řádku pomocí hvězdičky na alokované místo, na které ukazuje ukazatel, umístíme hodnotu 10. Na posledním řádku toto alokované místo uvolníme a tím operační systém informujeme o tom, že toto místo může přidělit jinému procesu a náš program k němu již nebude přistupovat.

```
1 int* numberPointer;
2 numberPointer = new int;
3 *numberPointer = 10;
4 delete numberPointer;
```

Ukázka kódu 5.1: C++ dynamická alokace datového typu int

Pokud bychom na konci zapomněli uvolnit paměť a náš program by nic jiného nedělal, není to až takový problém, protože operační systém si přidělenou paměť znovu vezme zpět po skončení programu.

Pokud následující funkci *a* v kódu 5.2 může program spouštět mnohokrát, dostáváme se do situace, kde alokujeme paměť, následně s ní pracujeme a na konci funkce *a* ji neuvolníme a dokonce ztratíme ukazatel na tuto alokovanou paměť, protože tento ukazatel je uložený v proměnné *numberPoint*, která přestane existovat právě na konci funkce *a*. Takže už ani nikdy nemůžeme paměť uvolnit, protože jsme o ní úplně ztratili jakékoliv informace.

```
1 int a() {
2     int* numberPointer;
3     numberPointer = new int;
4     *numberPointer = 10;
5 }
```

Ukázka kódu 5.2: C++ dynamická alokace datového typu int bez uvolňování

Zmíněné ukázky kódu jsou krátké a je nepravděpodobné, že by zkušený programátor zapomněl v těchto případech paměť uvolnit, nicméně programy obvykle nebývají takto přehledné a ne vždy je jednoduché si uvědomit všechny případy, kdy už je správná chvíle pro uvolnění paměti. Proto je tento způsob správy paměti náročný pro programátora a snadno v něm můžeme udělat chybu, která může mít fatální následky.

Tyto problémy lze řešit používáním nástrojů knihovny *memory*, které se postarají o alokaci i uvolnění paměti pomocí počítání referencí na alokovaný

objekt. Ve chvíli, kdy počet referencí klesne na 0, a tedy již žádná část programu nepoužívá tento objekt, může být uvolněn. [23]

### 5.4.2 Automatická

Mnoho jazyků, jako například Java, využívá k uvolňování pamětí tzv. *garbage collector*. [21]

Je mnoho implementací, ale často fungují ve dvou fázích. Nejprve proběhne identifikace a označení objektů, na které již neodkazuje žádná reference. V dalším kroku jsou tyto označené objekty smazány.

Velkou výhodou tohoto způsobu správy paměti je to, že programátor nemusí vůbec přemýšlet o uvolňování paměti a o případných problémech tím způsobených. *Garbage collector* je například v běhovém prostředí jazyku Java již vestavěný. Na druhou stranu tento proces potřebuje trochu výkonu ve chvíli, kdy je spuštěn, a ačkoliv je několik implementací, které jsou méně či více efektivní, je takřka vždy potřeba zastavit nebo zpomalit ostatní části aplikace, aby *garbage collector* mohl provést svou práci. Proto tento přístup nebude nikdy tak efektivní jako manuální správa paměti.[24]

### 5.4.3 Správa paměti v jazyku Rust

Jazyk Rust má velmi unikátní způsob správy paměti pojmenovaný *ownership*. Správa paměti je plně vyhodnocena a zajištěna kompilátorem při kompilaci. Při běhu je paměť uvolňována přímo po jejím využití podobně jako v C++. Tedy není potřeba žádný výkon navíc při běhu programu jako u zmíněného způsobu *garbage collector* ani není potřeba si dávat pozor na to, zda někde nenecháváme neuvolněnou paměť. Kompilátor nám to neumožní.

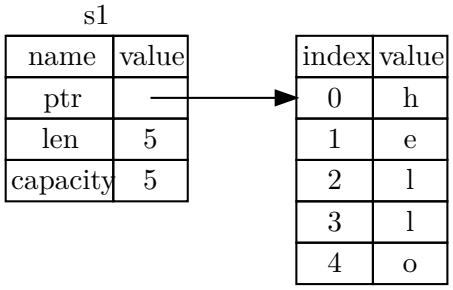
Aby byl kompilátor schopný správně zajistit uvolnění paměti, je potřeba dodržovat několik pravidel, které kompilátor kontroluje a nedovolí nám některé operace, které by v jiných jazycích bylo možné jednoduše provést.

Tímto získáváme výhody z obou způsobů uvolňování paměti za malou cenu komplexnějšího kódu. Tvůrci Rust kompilátoru ovšem vývojářům pomáhají kvalitními chybovými hláškami, které často dobře vysvětlí problém a s psaním kódu pomohou.

Pravidla systému *ownership* podle ([4] překlad autora):

1. *„Každá hodnota v programu má proměnnou, která se nazývá* `vlastník`*.“*
2. *„Může existovat vždy pouze jeden* `vlastník` *v daném čase.“*
3. *„Když se* `vlastník` *dostane mimo oblast platnosti*[1]*, hodnota je smazána.“*

[1] Oblastí platnosti je myšlený oborově známý anglický výraz „scope“.




Obrázek 5.1: Alokovaný *String* [4]

Tato pravidla nelze vysvětlit postupně a nezávisle na sobě, ale v následujících odstavcích budou zmíněny situace, ze kterých nakonec vyplyne význam všech pravidel.

K zajištění pravidla, že každá hodnota má pouze jednoho vlastníka, se v jazyce Rust využívá konceptu zvaného *borrowing*.

V ukázce kódu 5.3 si vysvětlíme rozdíl mezi mělkou kopií dat a přemístěním.

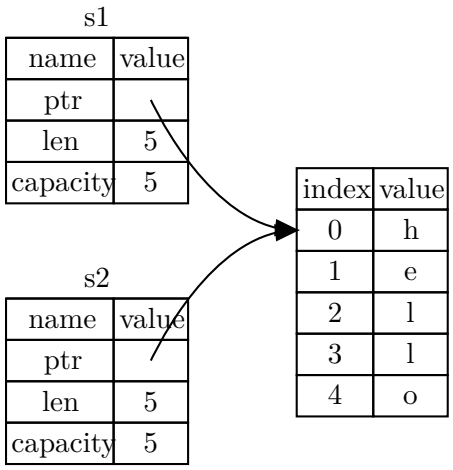
```
1 let s1 = String::from("hello");
2 let s2 = s1;
```

Ukázka kódu 5.3: Rust – ukázka inicializace dynamicky alokovaných dat a jejich přiřazení do další proměnné [4]

Datový typ *String* v jazyku Rust nemá známou velikost již při kompilaci stejně jako u většiny často používaných jazyků, a tak je potřeba data alokovat dynamicky za běhu programu. Grafické znázornění alokovaných dat můžeme vidět na obrázku 5.1. Jelikož *ptr* (ukazatel na dynamicky alokovaná data), *len* (délka textu) a *capacity* (kapacita alokovaného prostoru) jsou primitivní datové typy, jejich velikost je známá již za kompilace a je možné je uložit na *stack*, ale délka uloženého textu se může za běhu měnit, a proto musí být uložený na *heap*. Na *stack* společně s délkou textu a kapacitou uložíme pouze ukazatel na místo uložení textu.

Na druhém řádku v ukázce kódu 5.3 následně provedeme přiřazení dat z proměnné `s1` do proměnné `s2`. Nyní si vysvětleme dříve zmíněný pojem mělká kopie. Jelikož uložený text (nebo jiný typ dat, který nemá známou velikost při kompilaci) může být libovolně dlouhý, jeho kopírování může trvat velmi dlouho, provedeme tzv. mělkou kopii. Ta nám vytvoří kopii dat *ptr*, *len* a *capacity*. Hodnota *ptr* je ovšem stejná a ukazuje tedy na stejná data. Grafické znázornění můžeme vidět na obrázku 5.2. Oproti tomu hluboká kopie by zkopírovala i data uložená na *heap*.

Tento scénář však porušuje pravidlo 2, a proto ho Rust nedovoluje. Místo toho provede přemístění hodnoty do proměnné `s2` a ta se stane *vlastníkem*.




Obrázek 5.2: Mělká kopie [4]

Kdyby bylo možné vytvořit mělkou kopii, a tím více vlastníků hodnoty, nebylo by možné mazat hodnoty podle pravidla 3. Po přemístění hodnoty do proměnné *s2* je proměnná *s1* zneplatněna a nemůže být dále použita viz 5.5. [4]

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Ukázka kódu 5.4: Rust – kód, který nelze zkompilovat kvůli použití již nevalidní proměnné [4]

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which
    does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move

error: aborting due to previous error
```

Ukázka kódu 5.5: Výsledek kompilace 5.4 [4]

*„Rust nikdy implicitně nevytvoří hlubokou kopii dat.“* ([4] překlad autora) Pokud bychom chtěli vytvořit hlubokou kopii datového typu *String*, explicitně

zavoláme metodu *clone* jako v ukázce 5.6. Tento kód lze v pořádku zkompilovat. [4]

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Ukázka kódu 5.6: Rust – hluboká kopie [4]

Následně je potřeba zmínit možnost implicitního kopírování dat, u kterých známe velikost při kompilaci. Pokud máme v proměnné `x` hodnotu 5 datového typu `u32`, který potřebuje vždy 32 bitů, při přiřazení této hodnoty do jiné proměnné se provede zkopírování hodnoty, a tedy je možné původní proměnnou `x` znovu použít. Existují 2 hodnoty a každá má jiného `vlastníka`. Viz kód 5.7.

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

Ukázka kódu 5.7: Rust – implicitní kopie dat [4]

Pokud potřebujeme dynamicky alokovanou hodnotu předat do funkce, máme několik možností. V prvním případě se provede přemístění stejně jako u přiřazení. Tento případ je znázorněn v kódu 5.8. Program začíná na řádku 6, kde se alokuje a inicializuje proměnná `s1`. Následně je tato proměnná předána jako parametr do funkce `debug_len`. Protože hodnota `s1` nemá známou velikost při kompilaci a je alokovaná na *heap*, je provedeno přemístění do `s` a `s` se stává `vlastníkem`. Následně se vypíše délka textu a končí oblast platnosti proměnné `s`. Z toho důvodu je podle pravidla 3 její hodnota smazána. Pokud bychom následně chtěli použít proměnnou `s1`, není to již možné, protože proměnná po přemístění není platná.

Pokud bychom hodnotu potřebovali znovu použít, můžeme ji z funkce vrátit a znovu by se provedlo přemístění. Tento způsob však není moc přívětivý. Jak tento problém vyřešit lépe si ukážeme následně.

```
1 fn debug_len(s: String) {
2     println!("{}", s.len());
3 }
4
5 fn main() {
6     let s1 = String::from("hello world");
7     debug_len(s1);
8 }
```

Ukázka kódu 5.8: Rust – přemístění hodnoty do funkce

V kódu 5.9 si vysvětlíme, jak zařídit, aby proměnná nebyla po předání do funkce zneplatněna. Aby to bylo možné, nesmí se provést přemístění. Místo toho funkci předáme pouze referenci, čemuž se říká „půjčení“. Ve chvíli, kdy

proměnná `s` opustí oblast platnosti, hodnota není smazána, protože `s` nevlastní hodnotu, a tak je možné znovu použít `s1` viz řádek číslo 6.

Reference je označena znakem „&“ a toto označení musí být jak v hlavičce funkce u parametru, tak při jejím volání u proměnné.

```
1 fn main() {
2     let s1 = String::from("hello");
3
4     let len = calculate_length(&s1);
5
6     println!("The length of '{}' is {}.", s1, len);
7 }
8
9 fn calculate_length(s: &String) -> usize {
10     s.len()
11 }
```

Ukázka kódu 5.9: Rust – půjčení hodnoty [25]

Pokud bychom chtěli hodnotu uvnitř funkce upravit, musíme explicitně specifikovat, že funkce může upravovat hodnotu. To zařídíme přidáním klíčového slova `mut` viz ukázka kódu 5.10. Toto klíčové slovo však musí být i u proměnné `s`, jinak je implicitně hodnota neměnná a ani není možné vytvořit její proměnlivou referenci.

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     change(&mut s);
5 }
6
7 fn change(some_string: &mut String) {
8     some_string.push_str(", world");
9 }
```

Ukázka kódu 5.10: Rust – proměnlivé půjčení [25]

Pravidla referencí podle ([25] překlad autora):

1. *„Můžete mít pouze jednu proměnlivou referenci nebo jakýkoliv počet neměnných referencí v daný čas.“*
2. *„Reference musí být platná.“*

Pravidlo 1 nám zaručuje, že se do hodnoty nebude zároveň zapisovat z více míst nebo zapisovat a číst. Díky tomu je hodnota vždy v konzistentním stavu. Příklad porušení tohoto pravidla můžeme vidět v ukázce kódu 5.11, kde na řádce číslo 5 vytváříme proměnlivou referenci, ale přitom stále existují platné neměnné reference na stejnou hodnotu. Platné jsou z důvodu, že se na řádce 7 (poté, co je vytvořena proměnlivá reference) používají. Kdyby již po vytvoření proměnlivé reference `r3` nebyla použita `r1` ani `r2`, je vše v pořádku.

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

Ukázka kódu 5.11: Porušení pravidla 1 [25]

Pravidlo 2 nám zaručuje, že reference ukazuje na hodnotu, která existuje. Pokud bychom měli referenci, která ukazuje na neplatnou proměnnou, hodnota je již smazána. Příklad porušení tohoto pravidla můžeme vidět v ukázce kódu 5.12, kde na řádce číslo 4 vracíme z funkce referenci na hodnotu, která bude v ten samý moment smazána, protože oblast platnosti jejího `vlastníka` je pouze v rámci této funkce.

```
fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Ukázka kódu 5.12: Porušení pravidla 2 [25]

Díky zmíněným a kompilátorem kontrolovaným pravidlům je možné při kompilaci bezpečně vědět, kdy je možné smazat dynamicky alokovanou hodnotu. Navíc se tímto způsobem předchází jevu zvanému *data race* – přístup do stejných dat z více míst v programu, kde alespoň jeden z přístupů upravuje hodnotu. [25]

Tento systém pomáhá předcházet určitým typům chyb, které se projevují nahodile až za běhu programu a špatně se dohledávají.

Z těchto důvodů je jazyk Rust nejen efektivní, ale i bezpečnější, a proto bude použit pro implementaci backend části aplikace.

## 5.5 Další výhody jazyka Rust

Rust je považován za moderní jazyk a snaží se co nejvíce usnadnit práci programátorovi a předejít chybám.

### 5.5.1 Hlášení chyb v kompilátoru

Pokud kompilátor *rustc* objeví chybu v kódu, obvykle vypíše velmi čitelné vysvětlení a často dokáže navrhnout úpravy, které mohou chybu opravit. Příklad chybové hlášky je 5.5.

### 5.5.2 Typový systém

Rust používá statické typování. To znamená, že každá proměnná má vždy jasně daný datový typ, může obsahovat pouze určitou množinu dat a mohou s ní být prováděny pouze určité operace. [26]

Statické typování je kontrolováno při kompilaci, a tak se předchází chybám za běhu programu způsobeným například voláním funkce s nesprávnými hodnotami.

### 5.5.3 Správce balíčků cargo

Většina programátorů pro psaní programů využívá knihovny, aby nemuseli implementovat to, co již někdo vyřešil před nimi. O stažení těchto knihoven k Rust projektu se stará správce balíčků *cargo*. Názvy a verze knihoven jsou zapsané v souboru *Cargo.toml* viz 5.13.

Kromě stahování balíčků nám *cargo* umožňuje i spouštění programu při vývoji pomocí příkazu `cargo run` a kompilaci pomocí příkazu `cargo build` za použití kompilátoru *rustc*. Pokud chceme produkční kompilaci s optimalizacemi, přidáme parametr `--release`.

Dále nám cargo umožňuje inicializovat celý projekt pomocí `cargo init`, což nám předvytvoří *Cargo.toml* a *src/main.rs*, který obsahuje fungující „hello world“ program.

```
1  [package]
2  name = "light-log"
3  version = "0.6.0"
4  authors = ["Dominik Dosoudil <mail@example.com>"]
5  edition = "2018"
6
7  [[example]]
8  name = "nats-publish-log-request"
9
10 [dependencies]
11 async-nats = "0.9.1" # client for message broker NATS
12 serde = { version = "1.0", features = ["derive"] }
13
14 [dev-dependencies]
15 serial_test = "0.5.1"
```

Ukázka kódu 5.13: Příklad souboru Cargo.toml

### 5.5.4 Vlastnost (*trait*)

Rust nenabízí programátorům třídy ani rozhraní ani dědičnost. Namísto toho představuje mnohem jednodušší a plnohodnotný systém „vlastností“ objektů. Díky těmto vlastnostem mohou mít různé objekty sdílené chování. Každý objekt může mít neomezeně různých vlastností a implementace této vlastnosti může být specifická i výchozí.

Definici vlastnosti bez výchozí implementace i její implementaci konkrétní struktuře můžeme vidět v ukázce kódu 5.14.

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct Article {
    pub headline: String,
    pub author: String,
    pub content: String,
}

impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{}, by {}", self.headline, self.author)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Ukázka kódu 5.14: Příklad definice vlastnosti [27]

## 5.6 API

API, tedy *Application Programming Interface*, je obecné rozhraní, které nám umožňuje interagovat s jinou aplikací.

Tato aplikace může být například knihovna, která nabízí funkce pro její použití, objekt mající veřejné metody pro jeho manipulaci nebo třeba serverová služba umožňující komunikaci přes internet pomocí nějakého protokolu. [28]

API nám znemožňuje manipulovat s aplikací libovolně a omezuje nás pouze na autorem očekávané zásahy. To zaručuje správné chování aplikace. Pokud bychom například mohli sahat přímo na data aplikace, mohli bychom je změnit do aplikací neočekávaného stavu a způsobit tím fatální chybu.

Na druhou stranu se díky API nemusí uživatel starat o konkrétní implementaci, což mu zjednodušuje práci. [29]

## 5.7 HTTP

*„Hypertext Transfer Protocol je protokol na aplikační vrstvě pro posílání dokumentů jako například HTML. Byl navržen pro komunikaci mezi webovými prohlížeči a webovými servery, ale může být použit i pro jiné účely. HTTP používá klasický klient server model, kde klient vytváří spojení pro poslání požadavku a následně čeká na odpověď.“* ([30] překlad autora)

HTTP požadavek posílaný klientem serveru se skládá ze několika částí. První částí je metoda, která určuje, co by se se zdrojem mělo dít.

Metody používané pro manipulaci zdrojů:

**GET** metoda označuje získání specifikovaného zdroje, stav nebude upraven a klient očekává požadovaný zdroj v odpovědi

**POST** je používána pro poslání nového subjektu do zdroje, obvykle způsobuje změnu stavu

**PUT** nahradí stávající reprezentaci cílového zdroje obsahem požadavku

**DELETE** smaže specifikovaný zdroj

**PATCH** se používá pro částečnou úpravu zdroje dat

Dále exitují méně často používáne metody *OPTIONS*, *HEAD*, *TRACE* a *CONNECT*. [31]

Další částí je cíl, což je obvykle URL adresa. Třetí částí je verze HTTP označující formát zbytku zprávy. Následně je možné přidat hlavičky a tělo zprávy.

### 5.7.1 HTTP API

Pomocí HTTP protokolu lze implementovat dříve zmíněné API. Pokud potřebujeme aplikaci umožnit vnější zásahy jiných (klientských) aplikací, můžeme implementovat podporu přijímání HTTP požadavků. Na základě těchto HTTP požadavků můžeme buď ovládat aplikaci, nebo z ní získávat data.

Pro tento typ API existují standardy jako například *REST* a *SOAP*, ale vzhledem k požadavkům na aplikaci není potřeba těchto standardů využívat.

## 5.8 WebSocket

Protože je požadavek, aby frontend část aplikace LightLog zobrazovala nové zprávy okamžitě po přidání, je potřeba mít technologii, která nám umožní tyto zprávy posílat ze serveru klientovi.

Dříve nebyla jiná možnost než využít tzv. *polling*, což znamená, že klient se periodicky dotazoval serveru o nová data pomocí HTTP protokolu. Tato metoda je ale neefektivní, protože data byla získána o nějaký čas později

než vznikla, protože klient si o ně řekl například každých 5 vteřin. Druhý problém tohoto řešení je posílání zbytečných požadavků, pokud žádná data od předchozího požadavku nevznikla.

Řešením tohoto problému je právě WebSocket. Ten umožňuje klientovi navázat se serverem udržované spojení, skrz které mohou obě strany posílat data. Tato data přijdou druhé straně okamžitě. [32]

## 5.9 NATS

*„NATS.io je jednoduchý, bezpečný a výkonný open source systém pro posílání zpráv v cloud aplikacích, IoT nebo architektuře mikro služeb.“* ([33] překlad autora)

Aplikaci NATS se obecně říká *message broker*, což je systém umožňující posílání a přijímání zpráv mezi 2 a více stranami. Tedy zprostředkovává komunikaci mezi různými službami.

NATS pro komunikaci více služeb používá metodu publikovat/odebírat. Každé zprávě je při publikování přiřazený předmět, který se liší podle typu zprávy. Ostatní služby mohou tento předmět odebírat, pokud je daný typ zprávy zajímá. NATS se postará o to, aby daným službám chodily jen zprávy, jejichž předmět služba odebírá. [34]

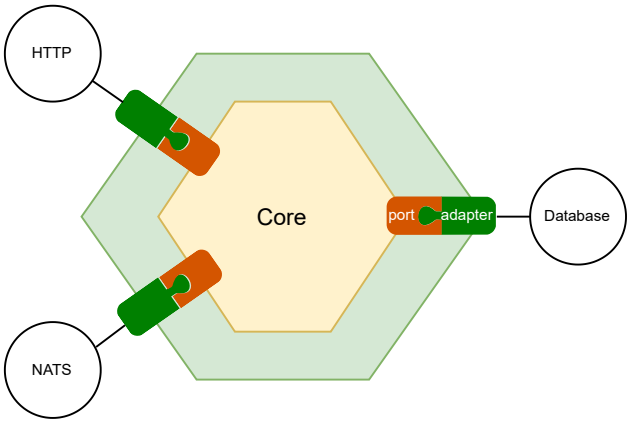
Další možností komunikace je požadavek/odpověď. Pokud služba A od služby B něco vyžaduje, je tento způsob vhodný. NATS tuto možnost umožňuje pomocí vytvoření předmětu, který bude použit při odeslání odpovědi na tento požadavek. Služba A vygeneruje náhodný předmět. Tento předmět začne odebírat. Následně odešle požadavek, ke kterému přidá předmět označující typ požadavku a vygenerovaný předmět, na který má služba odeslat odpověď. Pokud existuje služba, která umí zpracovat daný typ požadavku, odebere zprávu, zpracuje ji a odešle odpověď na původně vygenerovaný předmět. Strana A obdrží odpověď a přestane odebírat vygenerovaný předmět. [35]

### 5.9.1 Rust a NATS

Pro práci s NATS existuje Rust knihovna nazvaná *async_nats*. Tato knihovna využívá asynchronního programování (5.11). [36] Umožňuje nám připojit se k NATS serveru, přihlásit se k odběru zpráv s určitým předmětem i odesílat zprávy pomocí struktury *Connection* a přidružených metod.

## 5.10 Hexagon architektura

Hexagon architektura, které se také říká „Architektura Portů a Adaptérů“, rozděluje aplikaci do několika částí. Hlavní částí je doménová logika, která by měla fungovat beze změn, pokud se rozhodneme například změnit komunikační protokol. Proto musí být slabě vázaná na ostatní komponenty aplikace.




Obrázek 5.3: Hexagon architektura

K tomu slouží tzv. porty (oranžové na obrázku 5.3). K těmto portům pak existují adaptéry (zelené na obrázku 5.3) umožňující s ostatními komponentami komunikovat tak, jak to vyžaduje port.

To velmi usnadňuje případné nahrazení jednotlivých komponent, protože není potřeba upravovat jádro. Stačí jen vytvořit vhodný adaptér.

## 5.11 Asynchronní programování

Asynchronní programování je typ paralelního programování, které nám umožňuje spustit více souběžných úloh na malém množství vláken. [37]

Paralelní programování je užitečné, pokud chceme urychlit běh programu vypočítáním určitých podproblémů na jiných vláknech procesoru nebo pokud chceme zajistit, aby například přijetí nových požadavků od klienta nebylo blokované zpracováváním požadavků předchozích.

Asynchronní programování se vyznačuje přirozenější sémantikou kódu na rozdíl od vícevláknového programování. Tomu napomáhají klíčová slova *async* a *await*. Dále asynchronní programování vyžaduje méně výkonu na samotnou paralelizaci. [37]

## 5.12 Výběr technologií pro klientskou aplikaci

Je potřeba administrátorovi umožnit přehledné nahlížení do databáze log zpráv. Získání dat bude probíhat skrz backend část. Pro dobrou přístupnost bude klientská aplikace dostupná z webového prohlížeče. Musí umožňovat prohlížet historii zpráv a zároveň zobrazovat nově příchozí zprávy v reálném čase.

## 5.13 TypeScript

TypeScript je nadstavbou velmi populárního jazyka JavaScript. TypeScript přidává několik rozšíření, z nichž nejzásadnější je statické typování. Nijak nemění již stávající syntax jazyka JavaScript, ale pouze přidává nové věci. Tedy není potřeba kompletně přepisovat již stávající kód. Pouze stačí postupně přidávat typy. Zároveň TypeScript podporuje odvozování typů, takže není nutné přidávat typ úplně všude a získáme ekvivalentní bezpečnost. Příklad odvozování můžeme vidět v ukázce kódu 5.15. [38]

Výhody statického typování jsou obdobné jako u již zmíněného typového systému jazyka Rust (5.5.2). Předchází chybám za běhu programu spojeným s datovými typy.

TypeScript kód je po provedení kontroly typů a validity kódu včetně přidaných vlastností (např. datová struktura *enum*) zkompilován do JavaScript kódu a můžeme jej tedy spustit kdekoliv, kde již lze spustit JavaScript. Nejběžnější je prohlížeč nebo například populární běhové prostředí *node.js*.

```
1 let x = 3;
2 //  ^ = let x: number
3
4 let y = [0, 1, null];
5 //  ^ = let y: (number | null)[]
```

Ukázka kódu 5.15: Příklady odvozování typů [39]

## 5.14 React

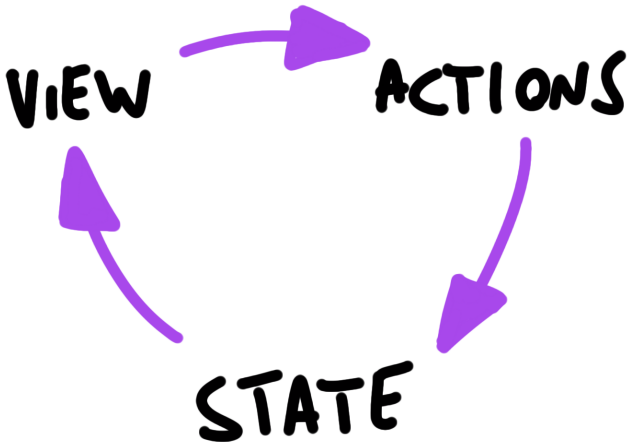
React je knihovna vytvořená firmou Facebook určená pro tvorbu uživatelských rozhraní primárně pro webový prohlížeč. Nicméně je možné použít React i pro mobilní zařízení.

React používá k tvorbě grafického rozhraní komponenty, které je možné psát hierarchicky podobně jako HTML. Oproti HTML ovšem dokáží udržovat vlastní vnitřní stav. [40]

Jedním ze základních principů fungování knihovny React je jednosměrný tok dat znázorněný na obrázku 5.4. Díky tomu, že data lze upravovat vždy jen jedním směrem, dokáže React překreslovat grafické rozhraní jen tehdy, pokud se data změnila. Je ovšem potřeba dodržovat pravidlo úpravy dat pouze tak, aby se vždy změnila i reference. [41]

## 5.15 PostgreSQL

Nakonec je potřeba vybrat, kam bude aplikace ukládat historii log zpráv. Jelikož bude backend navržený tak, aby bylo při potřebě možné vrstvu pro ukládání dat vyměnit za jinou za pomocí implementace odpovídajícího rozhraní, je možné v budoucnu tuto část nahradit.



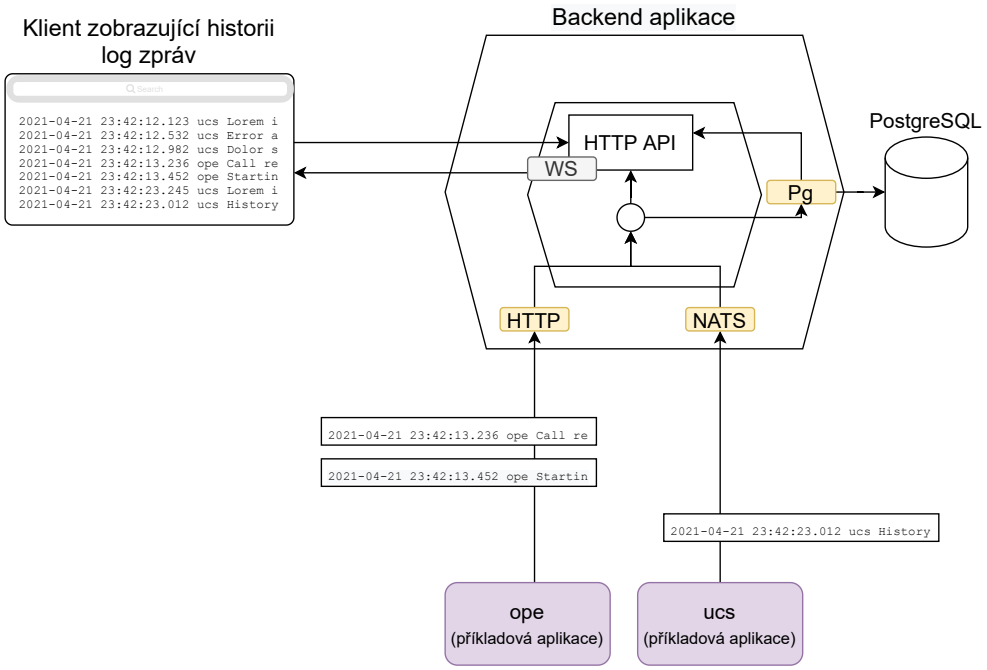

Obrázek 5.4: React - jednosměrný tok dat [5]

*„PostgreSQL je efektivní objektově relační databázový systém s 30 lety aktivního vývoje, který si zasloužil silnou reputaci za spolehlivost, robustnost funkcionalit a výkon.“* ([42] překlad autora)

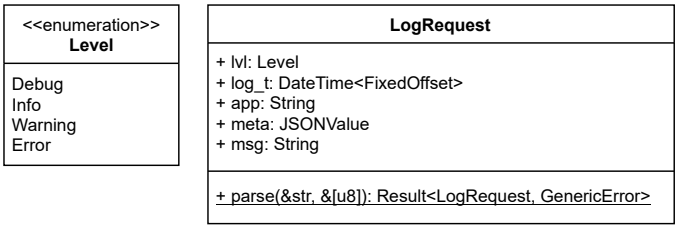
Kapitola 6

# Realizace

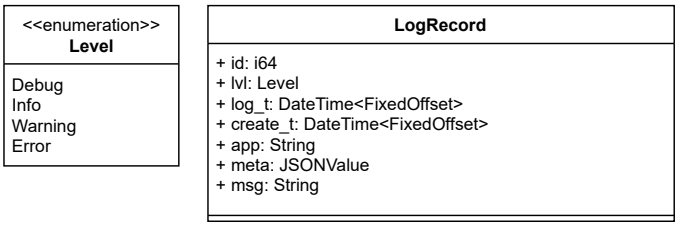
Tato kapitola pojednává o konkrétních klíčových krocích při implementaci aplikace. Nejprve si ukážeme, jak spolu komunikuje databáze, backend aplikace, klientská část a jiné programy logující pomocí této služby. Následně si vysvětlíme, jak funguje backend část, a nakonec klientskou aplikaci.




Obrázek 6.1: Architektura komunikace LightLog




Obrázek 6.2: UML diagram struktury LogRequest




Obrázek 6.3: UML diagram struktury LogRecord

## 6.1 Architektura částí a komunikace

Na obrázku 6.1 můžeme vidět znázornění všech částí aplikace a fialově podbarvených příkladů jiných programů. Zároveň vidíme komunikaci mezi nimi.

Backend aplikace je navržen podle hexagon architektury. To znamená, že jádro aplikace pracuje pouze s rozhraními komunikačních služeb a konkrétní implementace je od jádra odstíněna viz žlutě podbarvené obdélníky.

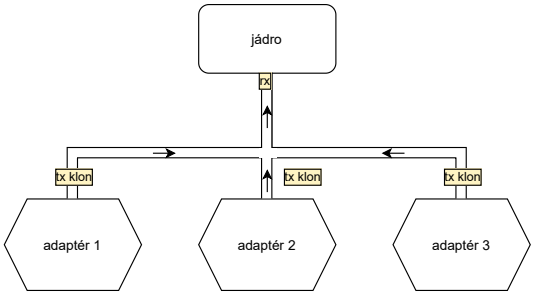
Díky tomu je možné jednotlivé části flexibilně vyměnit bez zásahu do jádra aplikace.

Adaptérů umožňujících zasílání požadavků o zaznamenání zprávy (na obrázku *HTTP* a *NATS*) může být navíc libovolně mnoho. Každý z těchto adaptérů ověří správnost přijatých dat a zpracuje je do struktury LogRequest znázorněné na obrázku 6.2, které rozumí jádro aplikace. Následně takto strukturovaná data odešle do kanálu, kterým se požadavek dostane do jádra aplikace.

Jádro tento požadavek uloží do databáze opět pomocí rozhraní propojující konkrétní databázi a jádro tak, aby mohla být použita jiná databáze. Pokud se podaří uložení a vygeneruje se k požadavku čas zapsání, převede se tento požadavek do struktury LogRecord, která má navíc `id` a `create_t` viz 6.3.

Tato struktura je následně předána části jádra, která pomocí WebSocket okamžitě distribuuje tento záznam aktivním klientům.

Poslední nezmíněný druh komunikace je v případě získávání historie log zpráv. To je umožněno pomocí HTTP API, které na požadavek s určitými parametry odpoví stránkou log záznamů.




Obrázek 6.4: Kanál mezi adaptéry přijímání požadavků a jádrem

## 6.2 Backend

### 6.2.1 Synchronizace dat v adaptérech přijímání požadavků

Jedním z nejdůležitějších implementačních bodů je rozšiřitelnost a modularita komunikačních kanálů pro zasílání požadavků o zaznamenání log zprávy. Nejprve si vysvětlíme, jak adaptéry předávají data jádru. Toho dosáhneme použitím tzv. kanálu z knihovny *async_channel*. Ta nám při vytvoření instance kanálu předá jeho strany – odesílací (*request_tx*) a přijímací (*request_rx*). Vytvoření kanálu můžeme vidět v ukázce kódu 6.1.

```
let (request_tx, request_rx) = channel::unbounded::<LogRequest>();
```

Ukázka kódu 6.1: Vytvoření asynchronního jednosměrného kanálu pro objekty struktury LogRequest

Obě strany lze navíc klonovat (vytvářet hluboké kopie). [43] Toho využijeme při vytváření adaptérů pro přijímání požadavků o zaznamenání log zpráv (v obrázku 6.1 HTTP a NATS).

Každému adaptéru tedy předáme klon odesílací strany viz kód 6.2 a přijímací stranu dáme jádru. Tímto způsobem budou jednotlivé adaptéry předávat data jádru a jádro nepotřebuje vědět, kolik a jaké adaptéry existují. Jednoduše odebírá data z kanálu. Tento návrh je znázorněný na obrázku 6.4.

```
for adapter in adapters.into_iter() {
    let request_tx_clone = request_tx.clone();
    task::spawn(async move {
        match adapter.start(request_tx_clone).await {
            Err(err) => { error!("{}", err); }
            _ => {}
        }
    });
}
```

Ukázka kódu 6.2: Zapnutí adaptérů a předání klonu tx

```rust
#[async_trait]
pub trait LogRequestPort {
    async fn start(&self, q: Sender<LogRequest>) -> Result<(), GenericError>;
}

#[async_trait]
impl LogRequestPort for HttpAdapter {
    async fn start(&self, log_request_tx: Sender<LogRequest>) -> Result<(), GenericError> {
        ...
        Ok(())
    }
}
```

Ukázka kódu 6.3: Ukázka implementace vlastnosti

### 6.2.2 Modularita adaptérů

Následně si ukážeme, jak je vyřešené samotné odstínění adaptérů od jádra. Když jádro aplikace zapíná adaptéry a předává jim `request_tx` na řádku 4 v kódu 6.2, pracuje s *trait* `LogRequestPort`, což je pouze obecné rozhraní, které musí adaptér implementovat, aby mohl být použit. Díky tomu jádro ví, že na každém adaptéru může volat funkci `start` s právě jedním parametrem *Sender<LogRequest>*, což je odesílací strana kanálu zmíněného dříve v této kapitole.

Tato vlastnost je adaptéru přidána pomocí klíčového slova `impl` viz ukázka kódu 6.3. Výhodou je naprostá volnost v konkrétní implementaci adaptéru. Jádro ho pouze spustí a sdělí, kam má posílat data. Dále již není třeba, aby se staralo o běh tohoto adaptéru.

Adaptéry mající potřebnou vlastnost se následně zaregistrují na unifikovaném místě, jako tomu je v kódu 6.4. Datový typ prvků tohoto vektoru je specifikovaný jako `Box<dyn LogRequestPort + Send + Sync>`. Datový typ *Box* se v jazyce Rust používá pro ukazatel na data uložená v paměti *heap*. Je potřeba *Box* použít, protože adaptéry spolu sdílejí pouze vlastnost *LogRequestPort*, ale mohou být jiného typu a tedy mít jinou velikost. Vektor ale potřebuje znát velikost prvku již při kompilaci, což *Box* (ukazatel) má.

```rust
let adapters: Vec<Box<dyn LogRequestPort + Send + Sync>> = vec![
    Box::new(NatsAdapter::new(config.nats.clone())),
    Box::new(HttpAdapter::new(config.http_adapter.clone())),
    // ...another adapter can be added here
];
```

Ukázka kódu 6.4: Registrace adaptérů

*dyn LogRequestPort + Send + Sync* je datovým typem objektu, na který ukazuje typ *Box* představující ukazatel. To znamená jakýkoliv objekt mající zároveň vlastnosti *LogRequestPort*, *Send* a *Sync*. *LogRequestPort* jsme si již

vysvětlili, *Send* a *Sync* jsou zde jen proto, aby adaptér mohl být spuštěn v asynchronní úloze. Tyto vlastnosti není navíc potřeba ručně implementovat, kompilátor to provede sám pokud uzná, že je to možné.

### 6.2.3 Implementace adaptéru pro NATS

Jako příklad implementace adaptéru pro přijímání log požadavků si ukážeme adaptér pro komunikaci pomocí NATS.

Nejprve si vytvoříme strukturu *NatsAdapter*, ve které budeme udržovat stav adaptéru. Jediná hodnota, kterou budeme potřebovat pamatovat, je nastavení adaptéru viz ukázka kódu 6.5.

```
pub struct NatsAdapter {
    config: NatsConfig,
}
```

Ukázka kódu 6.5: Vytvoření struktury dat pro NatsAdapter

Následně si ukážeme jak vypadá implementace privátních metod a metody *new*, která slouží k vytvoření nové instance adaptéru.

```
impl NatsAdapter {
    pub fn new(config: NatsConfig) -> Self {
        Self { config }
    }

    async fn handle_msg(&self, msg: NatsMessage) -> Result<
    LogRequest, GenericError> {
        LogRequest::parse(
            msg.subject
                .strip_prefix(&self.config.topic_prefix)
                .expect("acquire message with subscribed topic"),
            &msg.data,
        )
    }

    async fn serve_subscription(&self, sub: &Subscription,
    ch_sender: &Sender<LogRequest>) {
        info!("NATS: Awaiting messages");
        while let Some(msg) = sub.next().await {
            match self.handle_msg(msg.into()).await {
                Ok(r) => {
                    ch_sender.send(r).await.expect("always send");
                }
                Err(err) => { error!("{}", err); }
            }
        }
    }
}
```

Ukázka kódu 6.6: Implementace metod pro NatsAdapter

Metoda *new* v kódu 6.6 pouze přijme instanci konfigurace, vytvoří a vrátí instanci sama sebe pomocí klíčového slova `Self`, které je v podstatě alias pro strukturu, ve které je metoda implementovaná. Zde se nabízí otázka, jestli by nebylo lepší přijmout pouze referenci, ale jelikož Rust provede přemístění, je toto řešení efektivní a adaptér se stává vlastníkem konfigurace. Navíc by byla v pořádku i hluboká kopie dat vzhledem k tomu, že by proběhla pouze jednou při spuštění aplikace a nezpomalovala by samotné zpracovávání požadavků.

Následuje metoda *handle_msg*, která má za úkol zpracovat přijatá data a vytvořit z nich obecnou strukturu *LogRequest*, se kterou umí pracovat jádro.

Jelikož struktura *LogRequest* je připravená na parsování některých formátů, využijeme metody *LogRequest::parse* a předáme jí pouze formát dat, který získáme z předmětu (např. „j“ pro *json*) a *data*.

Poslední metodou je *serve_subscription*, která pro každou přijatou zprávu vytvoří strukturu *LogRequest* pomocí již zmíněné metody *handle_msg* a předá ji do kanálu.

Nakonec musíme implementovat vlastnost *LogRequestPort*, díky které umí jádro pracovat s adaptérem, protože nehledě na ostatní implementaci musí mít tuto vlastnost všechny adaptéry pro přijímání log požadavků.

```
#[async_trait]
impl LogRequestPort for NatsAdapter {
    async fn start(&self, ch_sender: Sender<LogRequest>) -> Result
    <(), GenericError> {
        let url = format!("{}:{}", self.config.host, self.config.
    port);
        match async_nats::connect(url.as_str()).await {
            Ok(nc) => {
                let topic = format!("{}*", &self.config.
    topic_prefix);
                match nc.subscribe(topic.as_str()).await {
                    Ok(sub) => {
                        self.serve_subscription(&sub, &ch_sender).
    await;
                        Ok(())
                    }
                    Err(e) => Err(e.into()),
                }
            }
            Err(e) if e.kind() == ErrorKind::ConnectionRefused =>
    Err(GenericError::new(
                "Could not connect to NATS server".to_string(),
            )),
            Err(e) => Err(e.into()),
        }
    }
}
```

Ukázka kódu 6.7: Implementace vlastnosti LogRequestPort pro NatsAdapter

Implementaci této vlastnosti můžeme vidět v ukázce kódu 6.7. Tato vlastnost požaduje implementování metody *start* s jedním parametrem typu *Sender<LogRequest>*.

Tato metoda se pokusí připojit k *message broker* službě na nakonfigurované adrese a začít odebírat zprávy s nakonfigurovaným předmětem. Pokud se tyto akce provedou bez chyb, začne odebírané zprávy obsluhovat pomocí metody *serve_subscription* implementované v předchozích krocích.

### 6.2.4 Implementace HTTP API

Pro získávání historie log zpráv si vytvoříme HTTP API popsané v kapitole 5.7.1, které nebude vzhledem k potřebným funkcionalitám nijak komplexní. Stačí nám pouze dva koncové body, a proto není třeba využívat žádného API standardu jako například REST.

První z koncových bodů slouží k získání výčtu identifikátorů aplikací, které používají LightLog. Díky tomuto výčtu může klientská aplikace filtrovat právě podle identifikátoru aplikace. Druhým koncovým bodem je samotné získání historie zpráv na základě několika filtračních parametrů, které si ukážeme v následujících odstavcích.

Pro každý koncový bod si napíšeme vlastní modul. Abychom si mohli vysvětlit výhodu tohoto přístupu, nejprve si ukážeme koncový bod pro získání historie zpráv. Tento modul s koncovým bodem můžeme vidět v ukázce kódu 6.8. Uvnitř modulu si nejprve vytvoříme strukturu *Query*, která znázorňuje filtrační parametry, které klient může poslat v *GET* požadavku. Právě díky tomu, že používáme modul, můžeme mít takovou strukturu u každého koncového bodu a nemusíme vymýšlet složité názvy jako například *GetLogsQuery*. Navíc je tato struktura skryta před ostatními moduly, čímž je kód lépe strukturován a oddělen.

Tato struktura obsahuje volitelné filtrační parametry a je z požadavku parsována pomocí knihovny *serde*. Tato knihovna při nesprávném formátu dat vytvoří chybu s popisem, co je v datech nesprávně. Díky tomu klient získá rozumnou chybovou odpověď.

Následuje generická asynchronní metoda *handle*. Generický typový parametr `T` s vlastností *DbPort* je datový typ poskytnutého adaptéru pro práci z databází. Díky tomu můžeme konkrétní databázi změnit a při dodržení této vlastnosti u adaptéru bude tato část programu fungovat nadále bez zásahu. Jediným parametrem této metody je požadavek typu `Request<State<T>>`, který obsahuje všechny informace o HTTP požadavku a navíc přibalený stav HTTP API, který obsahuje přístup k databázi. Tímto způsobem může bezstavová funkce *handle* používat databázi, jak můžeme vidět na řádku 18, kde si získáme instanci připojení k databázi a následně si z ní vyžádáme záznamy pomocí metody *get_logs* na řádku 21, které zároveň předáme filtrační parametry. Pokud vše projde bez chyb, z metody *handle* vrátíme odpověď pomocí

struktury *Response*, kterou vytvoříme z *json* dat. Tedy tělo odpovědi bude ve formátu *json*.

```
mod get_logs {
    use super::*;
    use chrono::{DateTime, Utc};

    #[derive(Deserialize, Debug)]
    struct Query {
        last_loaded_record: Option<i64>, // can be understood as max_time in a way
        min_time: Option<DateTime<Utc>>,
        max_time: Option<DateTime<Utc>>,
        apps: Option<Vec<String>>,
        search_regex: Option<String>,
    }

    pub(super) async fn handle<T: DbPort>(r: Request<State<T>>) -> Result {
        let db = r.state();
        let q: Query = r.query().unwrap();

        match db.db_pool.get().await {
            Ok(client) => Ok(Response::from(json!(
                &client
                    .get_logs(
                        q.last_loaded_record,
                        q.min_time,
                        q.max_time,
                        q.apps,
                        q.search_regex
                    )
                    .await
            ))),
            Err(_e) => {
                warn!("Could not get db client from pool");
                Err(Error::from_str(StatusCode::InternalServerError, ""))
            }
        }
    }
}
```

Ukázka kódu 6.8: Modul pro zpracování koncového bodu získání zpráv

### 6.2.5 WebSocket

Pro posílání nových zpráv klientům v reálném čase poskytuje webový prohlížeč, ve kterém běží klient, technologii WebSocket. Abychom WebSocket spojení mohli se serverem navázat, musíme si nejprve zpracovat poslouchání a ukládání těchto spojení.

V ukázce kódu 6.9 můžeme vidět zapnutí přijímání *TCP* spojení pomocí metody *bind* ve struktuře *TcpListener*. Při úspěšném spuštění začneme odebírat spojení a předávat je funkci *accept_connection*, kterou spustíme v asynchronní úloze. Zároveň funkci předáme kolekci, která udržuje všechny aktivní WebSocket spojení.

Funkce *accept_connection* následně TCP spojení povýší na WebSocket spojení provedením tzv. *WebSocket handshake*. [44] Poté WebSocket spojení přidá do kolekce předané v parametru. Nakonec čeká na ukončení spojení a po ukončení toto spojení zase z kolekce odebere. Díky tomu jsou v kolekci validní aktivní spojení.

```
match TcpListener::bind(&addr).await {
    Ok(listener) => {
        while let Ok((stream, _)) = listener.accept().await {
            task::spawn(accept_connection(stream, ws_connections.
    clone()));
        }
    }
    Err(err) => { warn!("{}", err);}
}
```

Ukázka kódu 6.9: Přijímání WebSocket spojení

Uložených WebSocket spojení v kolekci využije asynchronní úloha, ve které je spuštěna funkce *broadcast_log_records*, která z parametrů získá jednak stranu kanálu, do kterého chodí vytvořené log záznamy, a také právě zmíněnou kolekci. Následně pro každý nový záznam tato úloha záznam rozešle všem aktivním WebSocket spojením viz kód 6.10.

```
pub async fn broadcast_log_records(
    record_rx: async_std::channel::Receiver<LogRecord>,
    ws_connections: Arc<Mutex<WSConnections<TcpStream>>>,
) {
    while let Ok(log_record) = record_rx.recv().await {
        let mut ws_conns_guard = ws_connections.lock().await;
        let message = Message::Text(to_string(&log_record)
            .expect("serialize"));
        for (addr, conn) in (*ws_conns_guard).iter_mut() {
            if let Err(e) = conn.send(message.clone()).await {
                warn!("Err sending record to {}\n\t{}", addr, e);
            }
        }
        drop(ws_conns_guard);
    }
}
```

Ukázka kódu 6.10: Rozesílání nových log záznamů připojeným klientům

### 6.2.6 Konfigurace aplikace

Aplikace nabízí možnost nastavit některá její chování. Často jsou to porty, na kterých se spustí jednotlivé služby.

První částí nastavení jsou parametry při spuštění aplikace. Se získáním těchto parametrů nám pomůže knihovna *clap*, která umí i vypsat manuál možných parametrů pomocí přepínače *-h*. [45]

Nejprve si vytvoříme strukturu *Opts*, do které nám knihovna *clap* následně vloží zpracované parametry nebo odpoví uživateli chybou při chybějícím nebo nesprávném parametru. Tuto strukturu můžeme vidět v ukázce kódu 6.11. Obsahuje jeden atribut *config_file* typu *String*. To znamená, že se uživateli umožňuje předat programu cestu ke konfiguračnímu souboru. Pokud tento parametr nespecifikuje, použije se výchozí hodnota */etc/llog/config.yml*.

```
#[derive(Clap, Debug)]
#[clap(
    name = "Light Log",
    version = "0.3.0",
    author = "INSOFT s.r.o. (Dominik Dosoudil) <dosoudil@insoft.cz>"
)]
pub struct Opts {
    #[clap(short, long, default_value = "/etc/llog/config.yml")]
    pub config_file: String,
}
```

Ukázka kódu 6.11: Struktura parametrů programu při spuštění

Druhou částí je konfigurace pomocí souboru. K tomu využijeme knihovnu *serde* pro parsování a *async_std* pro otevření a přečtení souboru.

Podobně jako pro *clap* si i pro *serde* vytvoříme strukturu viz 6.12, podle které proběhne validace konfigurace. Po validaci knihovna nakonfigurované parametry načte do této struktury.

Prvky struktury jsou struktury podobné této s konkrétními konfiguracemi pro jednotlivé části programu.

```
#[derive(Debug, Deserialize, Eq, PartialEq)]
#[serde(default)]
pub struct Config {
    #[serde(default)]
    pub nats: NatsConfig,
    #[serde(default)]
    pub db: DbConfig,
    #[serde(default)]
    pub api: ApiConfig,
    #[serde(default)]
    pub http_adapter: HttpAdapterConfig,
}
```

Ukázka kódu 6.12: Struktura konfiguračního souboru

## 6.3 Frontend

Klientskou část si vytvoříme jako webovou aplikaci, aby ji administrátor mohl pohodlně spustit v prohlížeči bez nutnosti instalace. K sestavení aplikace využijeme knihovnu *React* a *material-ui* pro design.

### 6.3.1 Kořenová komponenta App

Aplikace má jednu kořenovou komponentu nazvanou *App*. Všechny ostatní komponenty jsou uvnitř této. Komponenta *App* se stará hlavně o validaci konfigurace, která jí může být poskytnuta skrz *prop*, což je něco jako parametr funkce. Následně se stará o obalení zbytku aplikace některými dalšími komponentami, které poskytují funkčnost. To vše můžeme vidět v ukázce kódu 6.13.

```
interface AppProps {
  cfg: unknown;
}

function App({ cfg }: AppProps) {
  try {
    jsonSchemaValidation(cfg, configurationSchema);
  } catch (exception) {
    if (exception instanceof ValidationException) {
      console.info('Configuration validation error');
      exception.errors.forEach((err) => { console.error(err); });
      return (
        <>
          <h1>Configuration validation errors:</h1>
          {exception
            .errors
            .map((error: ValidationError) => (<p key={
    error.toString()}>{error.toString()}</p>))}
        </>
      );
    }
    throw exception;
  }

  return (
    <ConfiguredI18nProvider>
      <ConfiguredStoreProvider config={cfg}>
        <WebSocket url={makeApiUrl(Protocol.Ws, cfg.api)}>
          <MuiThemeProvider theme={theme}>
            <Wrapper />
          </MuiThemeProvider>
        </WebSocket>
      </ConfiguredStoreProvider>
    </ConfiguredI18nProvider>
  );
}
```

Ukázka kódu 6.13: Komponenta App

Korektnost poskytnuté konfigurace ověříme pomocí knihovny *jsonschema*, která podle předem definovaného schéma ověří správnost předaných dat. Pokud konfigurace není správná, aplikace zobrazí chyby místo obvyklého uživatelského rozhraní.

Pokud je konfigurace správná, komponenta vykreslí komponentu *Wrapper* obalenou komponentou *ConfiguredStoreProvider*, která umožňuje ostatním komponentám nahlížet do globálního stavu aplikace a vytvářet akce, které tento stav mohou změnit pomocí knihovny *react-redux*, kterou vysvětlíme v následující kapitole (6.3.2). Dalšími obalujícími komponentami jsou *WebSocket*, která aplikaci připojí k serveru pomocí *WebSocket* spojení, a *MuiThemeProvider*, která nastaví aplikaci barevné téma.

### 6.3.2 Redux

*Redux* udržuje globální stav aplikace. Obvykle se globální stav považuje za nesprávný návrh, ale v tomto případě tomu tak není. K tomuto stavu mohou přistupovat pouze komponenty uvnitř zmíněné komponenty *ConfiguredStoreProvider* a upravovat jej nemohou přímo, ale pouze skrze předem definované akce. Tedy stav se mění pouze předvídatelným způsobem.

```
const initialState: State = {
  records: [],
  loadingRecords: false,
  noMore: false,
  apps: [],
  filter: {
    dateRange: null,
    apps: [],
    regex: null,
  },
};

const reducer = (state = initialState, action: Action) => {
  switch (action.type) {
    ...
    case LOAD_APPS_SUCCESS:
      return { ...state, apps: action.apps };
    case APPLY_FILTER:
      return {
        ...state,
        records: initialState.records,
        noMore: initialState.noMore,
        loadingRecords: initialState.loadingRecords,
        filter: action.filter,
      };
  }
};
```

Ukázka kódu 6.14: Redux stav a reducer

Nejprve si v ukázce kódu 6.14 nadefinujeme počáteční stav aplikace. Následně vytvoříme funkci *reducer*, která jako parametry přijme předchozí stav a akci. Na základě této akce tato funkce nějakým způsobem může upravit stav a tento upravený stav vrátí. Díky tomu získáme právě předvídatelné změny stavu. Není možné stav změnit jen tak odkudkoliv. Je pouze možné vytvořit akci, na základě které se stav změní předdefinovaným způsobem.

Akci, na základě které se může změnit stav, můžeme vidět v ukázce kódu 6.15. Akce je vytvořena funkcí *applyFilter*, kterou předáme funkci *dispatch*. Ta akci předá systému *Redux*. Funkce *dispatch* je poskytována aplikaci stejně jako globální stav pomocí React kontextu. Má ji tedy k dispozici každá komponenta uvnitř *ConfiguredStoreProvider*.

```
export const APPLY_FILTER = 'APPLY_FILTER';
export interface ApplyFilterAction { type: typeof APPLY_FILTER,
    filter: Filter }
export const applyFilter = (filter: Filter): ApplyFilterAction =>
    ({ type: APPLY_FILTER, filter });
```

Ukázka kódu 6.15: Příklad redux akce

## 6.4 Řešené problémy při implementaci

### 6.4.1 Potřeba *dyn* místo generického typu

Při registraci adaptérů viz 6.16 je potřeba, aby měl každý adaptér vlastnost *LogRequestPort*. Proto měl tento vektor původně generický typ *T* implementující právě potřebnou vlastnost. To ovšem fungovalo jen do té doby, dokud byl zaregistrovaný pouze jeden adaptér, protože v tu chvíli se generický typ zaměnil za typ použitého adaptéru.

Ve chvíli, kdy přibyl i *HttpAdapter*, nastala chyba, kdy typ druhého adaptéru neodpovídal prvnímu, který se zaměnil za generický při kompilaci. Proto bylo potřeba datový typ z generického změnit na dynamický objekt implementující vlastnost *LogRequestPort*.

```
let adapters: Vec<Box<dyn LogRequestPort + Send + Sync>> = vec![
    Box::new(NatsAdapter::new(config.nats.clone())),
    Box::new(HttpAdapter::new(config.http_adapter.clone())),
    // ...another adapter can be added here
];
```

Ukázka kódu 6.16: Registrace adaptérů pro přijímání log požadavků

### 6.4.2 Potřeba ruční implementace vlastnosti *Clone* u generického typu

Tento problém byl zajímavý, protože se z mého pohledu makro *derive* chová neintuitivně. Obecně platí, že pokud chceme umožnit klonování nějaké struk-

```
#[derive(Clone)]
struct State<T: DbPort> {
    db_pool: Pool<T, crate::db::postgres::PoolError>,
}
```

Ukázka kódu 6.17: Struktura *State* způsobující problém při *derive[Clone]*

tury, tato struktura musí implementovat vlastnost *Clone*. Obvykle je možné tuto vlastnost struktuře implementovat pomocí makra *derive*, které jednoduše naklonuje i všechny prvky této struktury. Pokud ovšem alespoň jeden prvek struktury nemá vlastnost *Clone*, nelze ani struktuře tuto vlastnost takto automatizovaně implementovat.

V tomto případě ale všechny prvky mají požadovanou vlastnost viz 6.17. Typ *Pool* implementuje *Clone* dle [46], a tedy *derive* by mělo fungovat.

Pomocí příkazu 6.18 si můžeme nechat kompilátorem ukázat, jak vypadá kód po expanzi maker. V ukázce kódu 6.19 můžeme vidět, jak implementace vlastnosti proběhla. Problém je hned na prvním řádku, kde je vidět, že generický typ $T$ musí mít vlastnost *::core::clone::Clone*, což ale není pravda, protože typ $T$ je použitý až uvnitř dříve zmíněného typu *Pool*.

```
cargo +nightly rustc -- -Zunstable-options --pretty=
    expanded
```

Ukázka kódu 6.18: Zobrazení expanze maker

```
impl <T: ::core::clone::Clone + DbPort> ::core::clone::Clone for
 State<T> {
    #[inline]
    fn clone(&self) -> State<T> {
        match *self {
            State { db_pool: ref __self_0_0 } => State{db_pool: ::
    core::clone::Clone::clone(&(*__self_0_0)),},
        }
    }
}
```

Ukázka kódu 6.19: Nesprávná automatická implementace *Clone*

Musíme tedy vlastnost *Clone* implementovat sami tak, jak to je v ukázce kódu 6.20. Tady je již správně u generického typu $T$ pouze nutnost vlastnosti *DbPort*.

```
impl<T: DbPort> Clone for State<T> {
    fn clone(&self) -> Self {
        State { db_pool: self.db_pool.clone() }
    }
}
```

Ukázka kódu 6.20: Správná implementace *Clone*

Kapitola 7

# Testování

## 7.1 Unit testy

Pro ověření funkčnosti klíčových prvků serverové části aplikace byly použity unit testy. Unit testy testují malé části kódu izolované od zbytku aplikace. Těmito částmi kódu mohou být například jednotlivé funkce, metody, nebo objekty. Obvykle je velmi čitelné, jaký se od testovaného kódu očekává výstup při určitém vstupu. Také je velmi snadné dohledat problém při skončení testu chybou, a to právě díky tomu, že testovaný kód je krátký.

Tyto testy se nachází přímo v souborech se zdrojovým kódem, tak jak je doporučeno pro jazyk Rust. Aby se oddělil zdrojový kód od testů, jsou testy obaleny do modulu.

Takto vytvořené testy se vyhodnocují automaticky po spuštění příkazu `cargo test`.

Otestované části kódu:

- ošetření neexistujícího nebo prázdného konfiguračního souboru
- parsování konfigurace
- použití výchozích hodnot v konfiguraci
- validace log požadavků s popisem chyb
- parsování log požadavků
- zpracování zprávy přijaté z NATS

## 7.2 Uživatelské testování

Celkové chování a funkčnost serverové i klientské části bylo vyzkoušeno při uživatelském testování, které provádí tester osobně používáním aplikace tak, jak

by to dělal uživatel. Při testování se postupuje zkoušením předem sepsaných funkcionalit nebo scénářů.

Testované funkcionality:

1. načtení historie záznamů
2. načtení starších záznamů při vyjetí nahoru (*infinite scroll*)
3. okamžité zobrazení nově vytvořených zpráv
4. zvýrazňování textu v popisu log zprávy pomocí regulárních výrazů
5. zobrazení metadat log zprávy
6. nastavení zobrazených metadat
7. filtrování podle textu, času, metadat a zdrojové aplikace
8. aplikace filtru na nově příchozí zprávy

### 7.2.1 Průběh

Nejprve byla aplikace otestována autorem této práce a byly vyzkoušeny všechny výše popsané funkcionality. Toto testování odhalilo několik chyb: zbytečné posílání více dotazů na server a duplikace zpráv (při 2), neošetřená hodnota při časovém omezení pouze z jedné strany (při 7) a ignorování zpráv při nastaveném prázdném filtru metadat (při 8). Všechny tyto chyby byly opraveny. První problém byl vyřešen záměnou funkce `takeEvery` za `takeLeading`, která ignoruje všechny požadavky o načtení starších zpráv dokud nepřijde odpověď na právě probíhající dotaz. Pro opravení druhé chyby stačilo ošetřit kromě hodnoty `undefined` i hodnotu `null`. Poslední chyba byla vyřešena odfiltrováním metadat s prázdným klíčem. Všechny tyto chyby vznikly a byly opraveny v klientské části.

Následně byla aplikace experimentálně nasazena na jeden z produkčních serverů firmy INSOFT s.r.o., kde proběhlo vyzkoušení funkcionalit. Ačkoliv produkce nemusí znít jako správné místo pro otestování aplikace, v tomto případě to není problém, jelikož logovací aplikace nijak nezasahuje do samotné funkcionality zbytku systému a nemůže ho tedy narušit.

Při tomto testování bylo nejprve odhaleno, že v aplikaci chybí zobrazování a filtrování metadat. Bez této funkcionality nebylo možné aplikaci nasadit. Tento nedostatek byl proto doplněn.

Následně proběhlo opětovné nasazení, kde LightLog služba fungovala bez pádů a splnila požadavky firemních administrátorů, kteří byli schopni vyhledávat záznamy ze systémů. Zároveň obstála pod produkční zátěží, nevytížila server a neblokovala běh ostatních služeb.

# Závěr

Cílem této práce bylo vytvořit službu pro agregaci a prezentaci log zpráv zaznamenávajících děj a stav jiných služeb systému. Požadavkem byla nízká náročnost na paměť a zátěž procesoru. Dalším požadavkem byla možnost rozšíření projektu, a tedy návrh a implementace projektu tak, aby bylo snadné vytvořit další moduly pro komunikaci i přes jiné protokoly.

Z těchto důvodu byl pro službu vybrán efektivní jazyk Rust, jenž splňuje aspekty efektivity z hlediska paměti a rychlosti. Zároveň splňuje i aspekty rozšiřitelnosti díky typovému systému a správě paměti. Aplikace byla navíc navržena v rámci hexagon architektury, která umožňuje rozšíření o další komunikační kanály.

V bakalářské práci byla nejprve provedena analýza konkurence, ze které vyplynulo, že již existující podobná řešení nejsou plně vhodná pro zadané požadavky. Následně byla provedena analýza technologií, která zahrnuje výběr a popis vhodného jazyka pro backend a vysvětlení použitých protokolů a knihoven. Nakonec byla probrána realizace projektu, která popisuje klíčové části implementace.

Serverová část byla implementována za pomoci kanálů pro synchronizaci dat mezi asynchronními úlohami a využití vlastností pro odstínění konkrétní implementace adaptérů od jejich použití v jádře. Konkrétně byly implementovány adaptéry přijímání požadavků o zaznamenání log zprávy pro protokoly HTTP a NATS.

Klientská část byla vytvořena v jazyku TypeScript za použití knihovny React. Díky tomu je moderní a přehledně naprogramovaná. Umožňuje uživateli prohledávání historie záznamů i pozorování nově vytvořených záznamů v reálném čase. Navíc je možné filtrování na základě času vytvoření záznamu, názvu aplikace, jenž záznam vytvořila, nebo klíčových slov v samotné zprávě záznamu.

Jelikož aplikace splnila požadavky firmy INSOFT s.r.o., bude použita v reálném provozu a umožňovat administrátorům a vývojářům call centra monitorovat a dohledávat dění v systému.

Protože byla zvolena modulární architektura, je možné aplikaci rozšířit o další komunikační kanály. Proto je možné ji použít, i pokud neumí potřebný komunikační protokol, a tento protokol si implementovat. Obdobně je možné aplikaci napojit i na jiné databázové systémy, pokud se napíše vrstva kompatibility.

# Literatura


[1] Parker, J.: Syslog Trap Levels – What are They and Diagram Tutorial! [online]. [cit. 2021-04-17]. Dostupné z: https://www.pcwdld.com/syslog-trap-levels

[2] Prokop, D.: New in Grafana v6.3: Introducing Loki's Log Row Context Viewer [online]. [cit. 2021-04-06]. Dostupné z: https://grafana.com/blog/2019/08/14/new-in-grafana-v6.3-introducing-lokis-log-row-context-viewer/

[3] InfluxData Community: Analyze logs with Chronograf [online]. [cit. 2021-04-06]. Dostupné z: https://docs.influxdata.com/chronograf/v1.8/guides/analyzing-logs/

[4] Rust Community: What Is Ownership? [online]. [cit. 2021-04-09]. Dostupné z: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

[5] Copes, F.: Unidirectional Data Flow in React [online]. [cit. 2021-04-19]. Dostupné z: https://laptrinhx.com/unidirectional-data-flow-in-react-1814911746/

[6] Benchmarks Game Team: C++ g++ versus Java fastest programs [online]. [cit. 2021-04-08]. Dostupné z: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gpp-java.html

[7] Benchmarks Game Team: C++ g++ versus Rust fastest programs [online]. [cit. 2021-04-08]. Dostupné z: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gpp-rust.html

[8] Gite, V.: Linux Log Files Location And How Do I View Logs Files on Linux? [online]. [cit. 2021-04-17]. Dostupné z: https://www.cyberciti.biz/faq/linux-log-files-location-and-how-do-i-view-logs-files/

[9] Plesky, E.: Linux Logs Explained [online]. [cit. 2021-04-17]. Dostupné z: https://www.plesk.com/blog/featured/linux-logs-explained/

[10] Infopulse: What is Microservices Architecture? [online]. [cit. 2021-04-18]. Dostupné z: https://www.infopulse.com/blog/the-importance-of-microservices-architecture-for-modern-applications/

[11] Amazon Web Services, Inc.: The ELK stack [online]. [cit. 2021-04-20]. Dostupné z: https://aws.amazon.com/elasticsearch-service/the-elk-stack/#:~:text=The%20ELK%20stack%20is%20an,Elasticsearch%2C%20Logstash%2C%20and%20Kibana.

[12] Opster Team: Elasticsearch Memory Usage Guide [online]. [cit. 2021-04-20]. Dostupné z: https://opster.com/elasticsearch-glossary/elasticsearch-memory-usage/#:~:text=The%20Elasticsearch%20process%20is%20very,for%20index%20values%20on%20disk.

[13] Bobriakov, I.: Grafana vs Chronograf and InfluxDB [online]. [cit. 2021-04-07]. Dostupné z: https://www.metricfire.com/blog/grafana-vs-chronograf-and-influxdb/

[14] Grafana Labs: Loki Documentation [online]. [cit. 2021-04-19]. Dostupné z: https://grafana.com/docs/loki/latest/

[15] Grafana Labs: Loki clients [online]. [cit. 2021-04-19]. Dostupné z: https://grafana.com/docs/loki/latest/clients/

[16] ThoughtWorks, Inc.: TICK Stack [online]. [cit. 2021-xx-xx]. Dostupné z: https://www.thoughtworks.com/radar/platforms/tick-stack

[17] InfluxData Community: Analyze logs with Chronograf [online]. [cit. 2021-04-07]. Dostupné z: https://docs.influxdata.com/chronograf/v1.8/guides/analyzing-logs/

[18] Aasen, G.: Introduction to InfluxData's InfluxDB and TICK Stack [online]. [cit. 2021-04-19]. Dostupné z: https://www.influxdata.com/blog/introduction-to-influxdatas-influxdb-and-tick-stack/

[19] Net-informations.com: Differences between Stack and Heap [online]. [cit. 2021-04-18]. Dostupné z: http://net-informations.com/faq/net/stack-heap.htm

[20] Bisht, A.: Stack vs Heap Memory Allocation [online]. [cit. 2021-04-18]. Dostupné z: https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/

[21] Gilmore, S.: Advances in Programming Languages: Memory management [online]. [cit. 2021-04-08]. Dostupné z: https://homepages.inf.ed.ac.uk/stg/teaching/apl/handouts/memory.pdf

[22] W3 Schools: C++ Data Types. [cit. 2021-04-09]. Dostupné z: https://www.w3schools.com/cpp/cpp_data_types.asp

[23] Kumar, H.: shared_ptr - basics and internals with examples [online]. [cit. 2021-04-09]. Dostupné z: https://www.nextptr.com/tutorial/ta1358374985/shared_ptr-basics-and-internals-with-examples

[24] Altvater, A.: What is Java Garbage Collection? How It Works, Best Practices, Tutorials, and More [online]. [cit. 2021-04-09]. Dostupné z: https://stackify.com/what-is-java-garbage-collection/

[25] Rust Community: References and Borrowing [online]. [cit. 2021-04-14]. Dostupné z: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[26] Křivánek, P.: Statická vs. dynamická typová kontrola. *Root.cz [online]*, červen 2004, [cit. 2021-04-15]. Dostupné z: https://www.root.cz/clanky/staticka-dynamicka-typova-kontrola/

[27] Rust Community: Traits: Defining Shared Behavior [online]. [cit. 2021-04-28]. Dostupné z: https://doc.rust-lang.org/book/ch10-02-traits.html

[28] Gazarov, P.: What is an API? In English, please. [online]. [cit. 2021-04-17]. Dostupné z: https://www.freecodecamp.org/news/what-is-an-api-in-english-please-b880a3214a82/

[29] Red Hat: What is an API? [online]. [cit. 2021-04-17]. Dostupné z: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces

[30] Mozilla and individual contributors: MDN Web Docs - HTTP [online]. [cit. 2021-04-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP

[31] Mozilla and individual contributors: HTTP request methods [online]. [cit. 2021-05-11]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[32] Mozilla and individual contributors: The WebSocket API (WebSockets) [online]. [cit. 2021-04-28]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[33] Cloud Native Computing Foundation: NATS [online]. [cit. 2021-04-16]. Dostupné z: https://nats.io/

[34] Cloud Native Computing Foundation: NATS Docs - Publish-Subscribe [online]. [cit. 2021-04-16]. Dostupné z: https://docs.nats.io/nats-concepts/pubsub

[35] Cloud Native Computing Foundation: NATS Docs - Request-Reply [online]. [cit. 2021-04-16]. Dostupné z: https://docs.nats.io/nats-concepts/reqreply

[36] NATS Authors: async_nats [online]. [cit. 2021-04-18]. Dostupné z: https://docs.rs/async-nats/0.9.12/async_nats/

[37] Rust Community: Why Async? [online]. [cit. 2021-04-28]. Dostupné z: https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html

[38] Microsoft: What is TypeScript? [online]. [cit. 2021-04-19]. Dostupné z: https://www.typescriptlang.org/

[39] Microsoft: Type Inference [online]. [cit. 2021-04-19]. Dostupné z: https://www.typescriptlang.org/docs/handbook/type-inference.html

[40] Facebook Inc.: React [online]. [cit. 2021-04-19]. Dostupné z: https://reactjs.org/

[41] Facebook Inc.: Tutorial: Intro to React [online]. [cit. 2021-04-19]. Dostupné z: https://reactjs.org/tutorial/tutorial.html#why-immutability-is-important

[42] The PostgreSQL Global Development Group: ostgreSQL: The World's Most Advanced Open Source Relational Database [online]. [cit. 2021-04-19]. Dostupné z: https://www.postgresql.org/

[43] Glavina, S.: Crate async_channel [online]. [cit. 2021-04-22]. Dostupné z: https://docs.rs/async-channel/1.6.1/async_channel/

[44] Mozilla and individual contributors: Writing WebSocket servers [online]. [cit. 2021-04-27]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

[45] Kevin K.: Crate clap [online]. [cit. 2021-04-28]. Dostupné z: https://docs.rs/clap/2.33.3/clap/

[46] Jung, M. P.: Struct deadpool::managed::Pool [online]. [cit. 2021-04-29]. Dostupné z: https://docs.rs/deadpool/0.7.0/deadpool/managed/struct.Pool.html

[47] aws: The ELK stack [online]. [cit. 2021-04-07]. Dostupné z: https://aws.amazon.com/elasticsearch-service/the-elk-stack/

[48] Mozilla and individual contributors: MDN Web Docs - URL [online]. [cit. 2021-04-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/URL

[49] Amenit s.r.o.: Softwarové Firewally [online]. [cit. 2021-05-10]. Dostupné z: https://www.antivirovecentrum.cz/firewally.aspx

[50] techopedia: Software Stack [online]. [cit. 2021-04-07]. Dostupné z: https://www.techopedia.com/definition/27268/software-stack

Příloha A

# Snímky klientské aplikace


Obrázek A.1: Výpis záznamů s aplikovaným zvýrazňováním


Obrázek A.2: Formulář pro filtrování log zpráv

Příloha B

# Seznam použitých zkratek

**API** Application Programming Interface [28]

**ELK** Elasticsearch, Logstash, Kibana [47]

**HTML** Hyper Text Markup Language

**HTTP** Hypertext Transfer Protocol [30]

**JVM** Java Virtual Machine

**LIFO** Last In First Out

**RAM** Random Access Memory

**TICK** Telegraf, InfluxDB, Chronograf, Kapacitor [13]

**URL** Uniform Resource Locator [48]

Příloha C

# Slovník

**C++** staticky typovaný programovací jazyk

**firewall** *„zařízení či software oddělující provoz mezi dvěma sítěmi, přičemž propouští jedním nebo druhým směrem data podle určitých předem definovaných pravidel“* [49]

**JVM** běhové prostředí pro aplikace naprogramované v jazyku Java

**Rust** moderní staticky typovaný jazyk s rozsáhlou kontrolou kódu předcházející výskytu některých chyb při běhu programu

**software stack** skupina programů spolupracujících za účelem dosažení společného cíle ([50] překlad autora)

**URL** je textový řetězec specifikující kde se zdroj (webová stránka, obrázek, video) nachází na Internetu. ([48] překlad autora)

Příloha D

# Obsah přiložené SD karty

- `README.md` ..... obsah
- `bin` ..... adresář se spustitelnými aplikacemi pro linux
  - `frontend-build` .... balíček souborů klienta připravený pro produkční nasazení na webový server
  - `light-log-macos` ..... serverová část (zkompilovaná pro MacOS)
  - `light-log-x86_64-linux` ... serverová část (zkompilovaná pro Linux)
  - `nats-publish-log-request-macos` .. příkladová aplikace, která posílá zprávy skrz NATS (zkompilovaná pro MacOS)
  - `nats-publish-log-request-x86_64-linux` příkladová aplikace, která posílá zprávy skrz NATS (zkompilovaná pro Linux)
- `src`
  - `light-log` ..... cargo projekt s implementací server části
    - `src` ..... zdrojové kódy
    - `sql` ..... inicializační kódy pro postgres databázi
    - `README.md` ..... popis možností spuštění aplikace
  - `light-log-ui` ..... npm projekt s implementací klient části
    - `src` ..... zdrojové kódy
    - `README.md` ..... popis možností spuštění klient aplikace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF