**Gymnázium, Praha 6, Arabská 14**

Programování

**ROČNÍKOVÝ PROJEKT**

# Kontejnerizační systém pro školní server

Vypracovali: Havránek Kryštof, Vávra Vladimír, Litoš Josef 3.E

Vedoucí práce: Ing. Daniel Kahoun

Duben 2021

Prohlašuji, že jsme jedinými autory tohoto projektu, všechny citace jsou řádně označené a všechna použitá literatura a další zdroje jsou v práci uvedené. Tímto dle zákona 121/2000 Sb. (tzv. Autorský zákon) ve znění pozdějších předpisů uděluji bezúplatně škole Gymnázium, Praha 6, Arabská 14 oprávnění k výkonu práva na rozmnožování díla (§ 13) a práva na sdělování díla veřejnosti (§ 18) na dobu časově neomezenou a bez omezení územního rozsahu.

V ..... dne ..... Podpisy autorů

**Poděkování**

Název práce: Kontejnerizační systém pro školní server

Autoři: Havránek Kryštof, Vávra Vladimír, Litoš Josef 3.E

Abstrakt: Cílem práce je vytvořit kontejnerizační systém pro operační systém GNU/-Linux. Ten by měl v jednoduchém a přehledném uživatelském prostředí umožnit žákům vytvořit si vlastný linuxový server. Na kterém si poté mohou hostovat svoje aplikace, webové stránky, či videohry. Systém by se měl postarat o to, aby zdroje serveru byly spravedlivě rozdělené mezi jednotlivé uživatele.

Klíčová slova: linux, lxd, lxc, kontejnery, nodejs, REST API, react js

Title: Containerization system for school server

Authors: Havránek Kryštof, Vávra Vladimír, Litoš Josef 3.E

Abstract: Goal of the work is to create containerization system for GNU/Linux operating system. With its help students should be able to create their own private server in a easy to use webUI. On such server they will be able to host their applications, websites or video games. System should also guarantee fair redistribution of server resources.

Key words: linux, lxd, lxc, containers, nodejs, REST API, react js

Titlel: Containerisierungssystem für Schulserver

Autoren: Havránek Kryštof, Vávra Vladimír, Litoš Josef 3.E

Abstrakt: Ziel des Werk ist ein Containerisierungssystem für das Betriebssystem GNU / Linux zu erstellen. Es sollte den Schülern ihren eigenen Linux-Server in einer einfachen und übersichtlichen Benutzeroberfläche möglicher zu bilden. Auf den Server können sie dann ihre Anwendungen, Websites oder Videospiele hosten, ich mag ein Brot. Das System sollte sich zusichern, damit die Serverressourcen gerecht auf die individuellen Benutzer verteilt würden.

Schlüsselwörter: linux, lxd, lxc, Schulprojekt, nodejs, REST API, react js

# Obsah

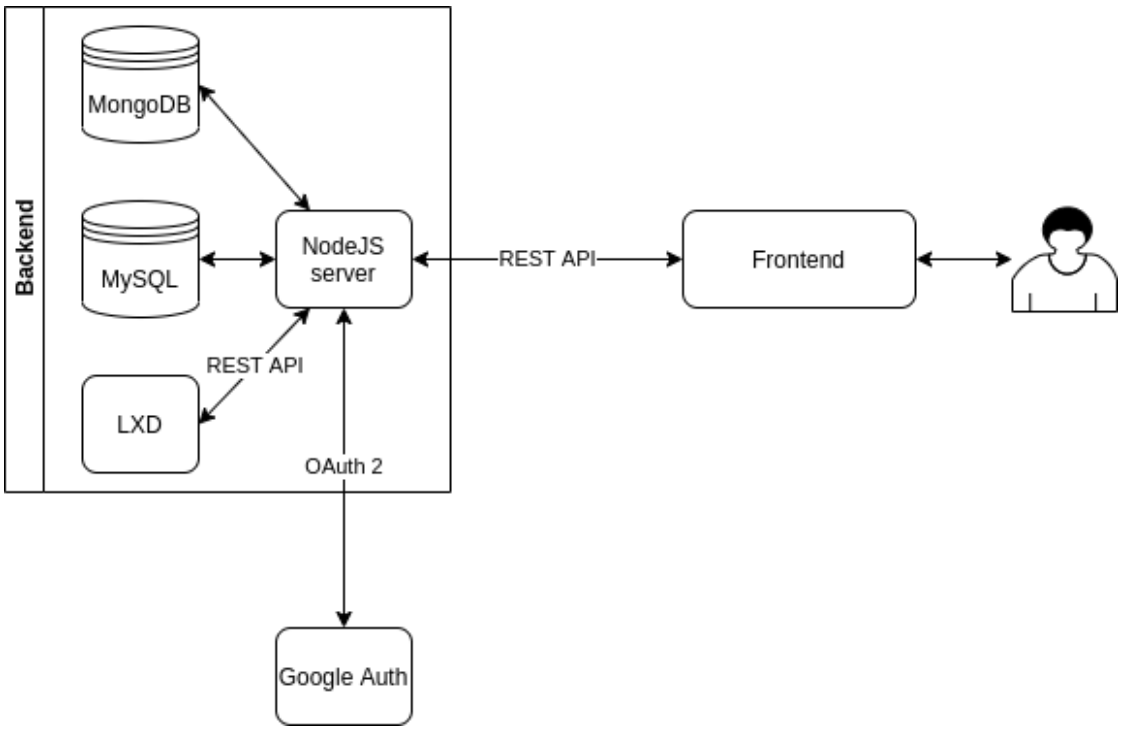
# Úvod

Cílem práce bylo postavit systém pro kontejnerizaci linuxového serveru, které by eventuálně mohl být nasazen na školním serveru. Systém by spravedlivě rozdělil zdroje mezi jednotlivé žáky a poskytl jim již volný prostor, kde mohou testovat svoje aplikace. Aktuálně totiž bylo problematické jen získat přístup na školní server.

O frontend práce se staral Vladimír, backend byl rozdělen mezi Kryštofa a Josefa. Kde Josef se převážně věnoval propojení mezi backendem a lxd deamonem, zatímco Kryštof zpracoval integraci mySQL databáze. Na spojení obou částí backendu a vytvoření route se podíleli oba členové týmu.

Vývoj probíhal na GitHubu – `https://github.com/havrak/AvAvA`, kde je aplikace dostupná pod licencí GPLv3. Projekt jsem testovali na vlastní VPS.

# 1. Architektura




Obrázek 1.1: Struktura projektu

Jádro práce stojí na lxd, což je kontejnerizační systém zabudovaný přímo do linuxového kernelu. Jedná se tak o nejvíce efektivní řešení, protože jednotlivé kontejnery mohou sdílet společný kernel. Jsou tak menší a rychlejší, než kdyby se jednalo o VPS. Backend s lxd komunikuje prostřednictvím REST API, které lxd přímo podporuje.

Samotný backend je napsán v nodejs a ke svému fungování využívá dva databázové systémy. Většinu informací ukládá do mySQL databáze, kvůli komplikacím s LXD byla později zavedena mongoDB.

Frontend je psán v ReactJS a s backendem komunikuje prostřednictvím vlastního REST API (jehož podrobná specifikace je uvedena v dokumentaci). Na frontendu může uživatel v pohodlném prostředí vytvářet nové kontejnery, projekty, či získat informace o jejich aktuálním stavu a mnohé další.

## 1.1 Sytém kontejnerů

## 1.2 Specifikace vlastního API

### GET – /api/combinedData

Routa GET /api/combinedData vrací objekt UserData. Odpověď obsahuje informace o uživateli, informace potřebné k vytvoření nového kontejneru (Dostupné šablony a aplikace, které je možno nainstalovat.) Mimo to je v odpovědi objekt UserProjects, který v sobě má uložené limity uživatele a informace o všech jeho projektech. Jedná se o jeden z prvních požadavků co frontend zavolá.

### POST – /api/instances

Routa POST /api/instances umožňuje uživateli vytvořit nový kontejner. Překročil li uživatel aktuální maximální limity, či vytvoření kontejneru selhalo vrátí se uživateli chybová hláška s kódem 400. Kontejner se pochopitelně smaže z databáze, kam je nutné ho první uložit.

Požadavek také umožňuje nechat na kontejner nainstalovat aplikace, selže li jejich instalace uživatel se o tom nedozví. Kontejner se však nesmaže.

Vytvoří li se dobře kontejner vrátí se uživateli Container objekt obsahující informace o právě vytvořeném kontejneru. Na backendu se zároveň vygeneruje nový konfigurační soubory proxy (viz. Backend⇒ Networking ⇒ Proxy).

### GET – /api/instances/createInstanceConfigData

Routa GET /api/instances/createInstanceConfigData slouží vrací objekt createInstanceConfigData. Je volána vždy před vytvořením nového kontejneru, aby uživatel měl na výběr mezi aktuální nabídkou šablon a aplikací, které je možno nainstalovat.

### GET – /api/instances/{id}

Routa GET /api/instances/{id} vrací objekt Container. V něm jsou uložené informace o dotazovaném kontejneru, jako jsou jeho limity, či aktuální stav. Jak bylo zmíněno v autorizaci, uživatel se může zeptat pouze na kontejnery, které vlastní, či je na nich uveden jako spolupracovník.

### PATCH – /api/instances/{id}

Routa PATCH /api/instances/{id} slouží k změně limitů kontejnerů. Aktuálně není změna limitů na backendu implementována. Po změně limitů by se měla vrátit objekt Container s novými limity a všechny záznamy v containersResourcesLog by byly vymazány. Změna limitů jde provádět pouze na vypnutém kontejneru, o to by se routa postarala.

### DELETE – /api/instances/{id}

Routa DELETE /api/instances/{id} slouží k smazání vybraného kontejneru. Operace nejde nijak revertnou, neudělal li si uživatel backup. Kontejner se jednoduše smaže z databáze a z lxd systému.

### GET – /api/instances/{id}/stateWithHistory

Routa GET /api/instances/{id}/stateWithHistory vrací pole objektů ContainerStateWithHistory. V něm je uloženo id kontejneru a logy stavu kontejneru (prostřednictvím ContainerResourceState). S aktuální konfigurací je stav zaznamenává co deset minut dvě hodiny nejstarší stav byl zaznamenám před 2 hodinami.

V objektech jsou vyplněna pouze pole, které jsou uloženy v databázi, tedy informace o stavu ram, cpu, počtu procesů, rychlosti uploadu, rychlosti downloadu a limity. Výjimku představuje poslední ContainerResourceState v poli, kde je uložen aktuální stav, který se získá přes volání do lxd.

### GET – /api/instances/{id}/console

Routa GET /api/instances/{id}/console slouží realizaci konzole ve webovém prostředí. Na serveru se vytvoří websocket s konzolí a přihlašovací údaje na něj se zašlou uživateli.

### GET – /api/instances/{id}/snapshots

Routa GET /api/instances/{id}/snapshots má sloužit k získání snapshotů kontejnerů. V aktuální verzi není routa implementována, obnovení snapshotu se totiž objevilo jako velmi problematické. Podobně jako u backupu se i u snapshotu musí řešit limity,

zde jsou však uložené ve snapshotu. Uživatel by měl možnost vybrat si jaký snapshots chce obnovit a jaké jeho limity se mají dodržet. Logika obnovení byla příliš složitá a její realizace se nestihla včas udělat.

### POST – /api/instances/{id}/snapshots

Routa POST /api/instances/{id}/snapshots by měla vyvolat vytvoření snapshotu kontejneru. Stejně jako další routy týkající se logiky snapshotů ani tato nebyla implementována.

### DELETE – /api/instances/{id}/snapshots/{snapshotsid}

Routa DELETE /api/instances/{id}/restore/{snapshotsid} by měla smazat snapshots Stejně jako další routy týkající se logiky snapshotů ani tato nebyla implementována.

### PATCH – /api/instances/{id}/restore/{snapshotsid}

Routa PATCH /api/instances/{id}/restore/{snapshotsid}

### GET – /api/instances/{id}/export

Routa GET /api/instances/{id}/export slouží k exportování backupu. Po jejím zavolání se na serveru vytvoří backup kontejneru a přes data stream se pošle uživateli. Aktuálně je kompresován ve formátu .tar.gz.

### PUT – /api/instances/import

Routa PUT /api/instances/import má sloužit k importu backupu kontejneru. V aktuální verzi není routa implementována, je totiž problematické zjistit stav limitů z backup. Routa by měla kontejner deplounout v sandboxu a zjistit si jeho limity. Poté ověřit zda uživatel má volné zdroje na jeho vytvoření, pokud ano kontejner by se vytvořil a zapsal do databáze.

### PATCH – /api/instances/{id}/start

Routa PATCH /api/instances/{id}/start slouží k nastartování kontejneru. Pokud se kontejner nastartuje jeho aktuální stav se uloží do databáze a vygeneruje se objekt

Container. Ten se pošle jako odpověď na požadavek.

#### PATCH – /api/instances/{id}/stop

Routa PATCH /api/instances/{id}/stop slouží k stopnutí kontejneru. Před zastavením se uloží jeho stav, se změněnými hodnotami (status bude stopped, využití ram nula, atd.), do mongoDB. Není totiž možné získat stav kontejneru je li stopnutý. Pokud se kontejner pozastaví jeho aktuální stav se uloží do mySQL databáze a vygeneruje se objekt Container. Ten se pošle jako odpověď na požadavek.

#### PATCH – /api/instances/{id}/freeze

Routa PATCH /api/instances/{id}/freeze slouží k zmražení kontejneru. Pokud se kontejner zmrazí jeho aktuální stav se uloží do databáze a vygeneruje se objekt Container. Ten se pošle jako odpověď na požadavek.

#### PATCH – /api/instances/{id}/unfreeze

Routa PATCH /api/instances/{id}/unfreeze slouží k rozmražení kontejneru. Pokud se kontejner rozmrazí jeho aktuální stav se uloží do databáze a vygeneruje se objekt Container. Ten se pošle jako odpověď na požadavek.

#### GET – /api/projects

Routa GET /api/projects vrací objekt UserProjects. V něm jsou uložené limity uživatele a pole objektů Project s inforamcemi o jeho projektech a kontejnerech v nich.

#### POST – /api/projects

Routa POST /api/projects slouží k vytvoření nové projektu. Prvně se zkontroluje, zda uživatel má místo na jeho vytvoření. Pokud ano, tak se vygeneruje JSON, jenž se odešle do lxd. U projektu lze limitovat pouze využití ram a využití disku. Oba limity se v praxi ukázali problematické a v projektu s nastavenými limity nešel vytvořit žádný kontejner.

Systém podporuje projekt s nulovými (null, ne nula) limity, aktuálně je však nejde mixovat.

Odpovědí na request je objekt Projekt, který se vygeneruje je li vytvoření projektu úspěšné.

### GET – /api/projects/stateWithHistory

Routa GET /api/projects/stateWithHistory vrací objekt UserStateWithHistory. V něm je uložená historie všech projektů uživatele, které obsahuje historii jeho všech kontejnerů.

### GET – /api/projects/{id}

Routa GET /api/projects/{id} vrací objekt Project s informacemi o projektu. Ten si mimo jiné pamatuje limity, název, spolupracovníky a pole Container objektů.

### PATCH – /api/projects/{id}

Routa PATCH /api/projects/{id} slouží k úpravě limitů projektu. V aktuální verzi je možno limity pouze zvýšit a projekt přejmenovat. Je li projekt přejmenován vygeneruje se nová konfigurace proxy. Odpovědí na request je objekt Project s novými limity projektu.

### DELETE – /api/projects/{id}

Routa DELETE /api/projects/{id} slouží k smazání projektu. Stejně jako smazání kontejneru ani smazání projektu nejde vrátit zpět. Metoda prvně smaže veškeré kontejnery nacházející se v projektu a poté projekt samotný.

### GET – /api/projects/{id}/stateWithHistory

Routa GET /api/projects/{id}/stateWithHistory vrací objekt ProjectStateWithHistory. V něm je uložená historie všech kontejnerů projektu, která využívá stejnou metodu jako GET /api/instances/stateWithHistory.

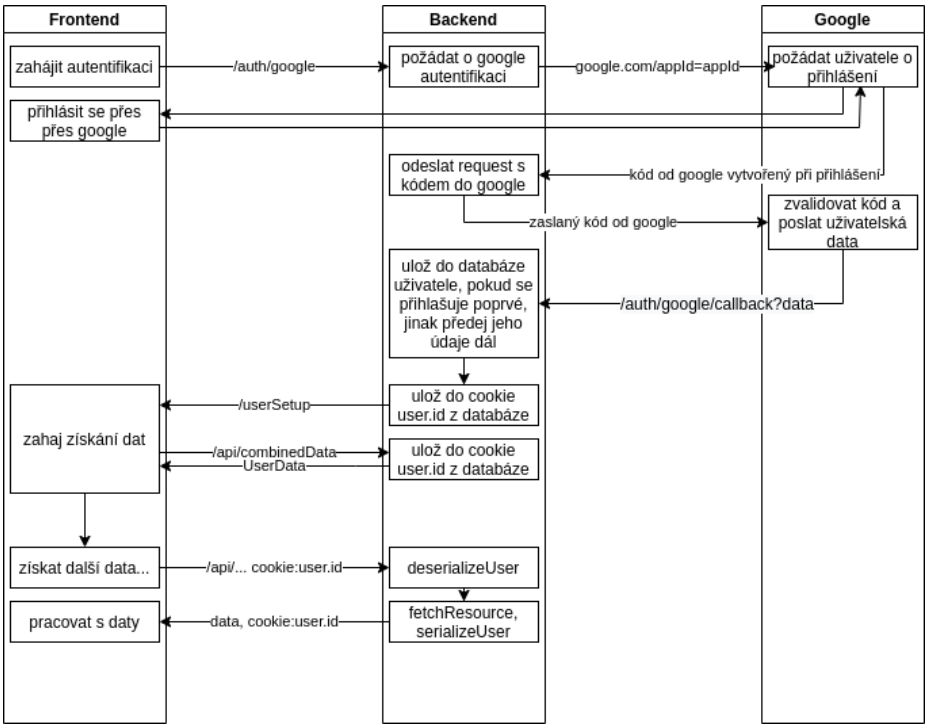
### GET – /api/user

Routa GET /api/user vrací objekt User. Jedná se o právě přihlášeného uživatele.

### GET – /api/logout

Routa GET /api/logout uživatel odhlásí ze systému.

## 1.3 Autentifikace




Obrázek 1.2: Autentifikace přes googleAuth 2.0

Za způsob autentifikace uživatelů byl zvolen Google Auth, jelikož všichni studenti gymnázia Arabská mají svůj vlastní školní Google účet. Díky tomu si každý student bude moci vytvořit vlastní server bez nutnosti registrace.

Pro zprovoznění autentifikace je nejprve nutné si v Google Console vytvořit nový projekt. Aktuálně je tento projekt hoštěn na studentském Google účtu `vladimir.vavra@student.gyarab.cz`. Je nastavené, že pouze uživatelé v rámci domény .gyarab se mohou do systému přihlásit. Poté, co se nastaví všechna potřebná data je vygenerován clientId a clientSecret, což jsou kódy, pomocí nichž se bude AvAvA autorizovat

### 1.3.1 Princip fungování

Na obrázku výše můžete vidět průběh autentifikačního procesu. Nejprve uživatel musí autentifikaci zahájit, což udělá pokusem o získání nějakého chráněného obsahu.

Po odeslání žádosti z klienta na backend přesměruje uživatele na přihlašovací stránku Google. V případě, že se student přihlásí poprvé, musí odsouhlasit, že povoluje programu využívat některé služby. Při dalších návštěvách se už jen přihlásí.

Tím je proces autentifikace hotov a začíná proces autorizace programu u Googlu. AvAvA tedy bude Google žádat, aby jí předal data o uživateli, která potřebuje. To dělá v obdélníku "odelsat request s kódem do google". Pokud google vyhoví, zavolá callback, který mu byl nastaven při vytváření projektu v Google Console a odešle na něj chtěná data. V případě, že se uživatel přihlásil poprvé, je o něm uložen záznam do databáze, se kterým poté backend bude párovat kontejnery a projekty. Následně je vytvořena session mezi frontendem a backendem. Každá zpráva odteď bude v hlavičce obsahovat cookie soubor s ID uživatele.

Po získání dat je uživatel přihlášen a může systém používat. Při odhlášení je session ukončena a už se dále user ID neposílá.

### 1.3.2 Implementace

Při implementaci byla hlavním zdrojem informací videa, ze kterých jsme získali část kódu[1]. Jelikož celý proces autentifikace probíhal na backendu, je uživatel odkázán na frontendovou cestu /userSetup. To je signál, že se uživatel přihlásil a frontend tedy pošle žádost na /api/combinedData.

Pro implementaci tohoto procesu na backendu bylo využito knihovny passport.js, která celý proces usnadňuje. Jediné, co je potřeba udělat je implementovat metody serializeUser a deserializeUser, které se budou volat po pořadě při zapisování ID do sessioncookie a při získávání uživatele z databáze za pomoci ID získané ze sessioncookie.

Je také nutné zapsat kód, který se zavolá hned po přihlášení, tedy uložení do databáze - passport.use(...) a předat passportu clientId a secretId.

Pro zjištění, zda je uživatel přihlášen je možné na kterékoliv z request objektů získat property req.user, ve které bude uložen výsledek posledního volání metody deserializeUser. V případě, že žádný uživatel není přihlášen, jeho hodnota je null. K tomuto účelu slouží metoda isLoggedIn

[1] *Oauth using react js | redux | node js | passport js | mongodb*. CODERS NEVER QUIT. Dostupné z `https://www.youtube.com/watch?v=gtg1-N7yfGM&ab_channel=CODERSNEVERQUIT`. [cit 2021-12-4]

### 1.3.3 Autorizace

Pro každý request do API se zkontroluje, zda je uživatel přihlášen. Není li přihlášen vrátí se mu kód 401 se zprávou, že není autentifikován. Jelikož metody počítají s emailem, který do requestů přidává googleAuth, tak není možné vykonat pro nepřihlášeného uživatele ani get requesty. Navíc není, žádoucí aby uživatel viděl stav ostatních uživatelů.

Routy, které obsahují id kontejneru, či projektu si ověřují zda je odesílatel requestu má právo upravovat kontejneru. U kontejneru, k tomu slouží metoda userSQL.doesUser OwnGivenContainer(email,id). Ta interně zavolá metod userSQL.doesUserOwnGiven Project(email, id), poté co zjistí id projektu z databáze. Činý tak jelikož pokud má uživatel právo na projektu, má zároveň právo na jeho kontejneru.

V aktuálním podobě se ověřuje zda je uživatel uveden buďto jako vlastník, nebo spolupracovník na projektu. Eventuálně by měl superadmin a admin mýt právo cizí kontejnery do určité míry ovlivňovat. Konkrétně admin na ně pouze nahlížet a superadmin s nimi volně zacházet a měnit je.

Ověření pomocí metody userSQL.doesUserOwnGivenProject(email, id) používá ještě routa /api/instances#POST. Zde je třeba ověřit zda uživatel vlastní projekt, do kterého se snaží vytvořit nový kontejner.

# 2. Frontend

# 3. Backend

## 3.1 Databáze

Sytém používá dva databázové systémy – mySQL (mariaDB) a mongoDB. Dle původních představ měl používat pouze mySQL. Kvůli komplikacím s lxd, kde nejde zjistit stav zastaveného/vypnutého kontejneru, byla později zavedena mongoDB.

### 3.1.1 mySQL

Následující sekce se zabývá strukturou mySQL databáze a metodami, které s databází zacházejí.

```
src
└── services
    └── sql
        ├── containerSQL.js
        ├── projectSQL.js
        ├── templateSQL.js
        └── userSQL.js
```

Obrázek 3.1: Soubory s programem týkající se mysql databáze

Valnou většinu dat projekt ukládá do mySQL databáze, kde je na tento účel vytvořeno aktuálně 10 tabulek. S databází pracují 4 soubory (viz. 3.1) *containerSQL* zpracovává věci týkající se uživatelů, *projectSQL* věci týkající se projektů, *templateSQL* věci týkající se šablon a aplikací, v neposlední řadě *userSQL* věci týkající se uživatele.

#### Tabulky

Tato sekce spěšně popíše obsah jednotlivých tabulek. A jejich vztah k ostatním. Kaskádové dependence jsou vždy stejné – při smazání se smažou, při update se nic neděje.

**appsToInstall** | id | name | description | icon_path | package_name |

Tabulka appsToInstall slouží k uložení aplikací, které je možno nainstalovat na kontejner při jeho vytváření. Nemá žádnou vazbu na další tabulky a s jejími daty

zachází třída templateSQL. Sloupec name obsahuje jméno jaké se má zobrazit uživateli, package_name je jméno balíčku v repositářích.

**containers** | id | project_id | name | url | template_id | state | timestamp | time_started |

Tabulka container ukládá všechny kontejnery, které spravujeme. Obsahuje dva cizí klíče a to project_id, které určuje do jakého projektu kontejner patří, a template_id to určuje s jakou šablonou byl kontejner vytvořen.

**containersResourcesLimits** | container_id | ram | cpu | disk | upload | download |

Tabulka containersResourcesLimits ukládá limity kontejneru, stejně jako u dalších .*ResourcesLimits tabulek byla v rámci normalizačních forem oddělena od tabulky containers. Tabulka obsahuje jeden cizí klíč, který zároveň funguje i jako primární klíč. Veškeré limity jsou uloženy v základních jednotkách, cpu je v abstraktní jednotce herz.

**containersResourcesLog** | container_id | ram | cpu | number_of_processes | upload | download | timestamp |

Tabulka containersResourcesLog slouží k logování stavu kontejnerů. Container_id je cizím klíčem odkazující na kontejner, ke kterému log patří. V aktuální verzi jsou data uložen v poli, které uložené ve sloupci s typem text. Timestamp odkazuje na datum posledního zapsání, kdy byl zalogován předchozí stav není problém zjistit, jelikož se do tabulky zapisuje v pravidelných intervalech.

Metoda na updateování stavu (updateLogsForContainer(id, state)) si jednoduše data rozdělí dle znaku: **,**. Do pole se přidají nové hodnoty a odebere se první prvek, nový stav se pak uloží do databáze.

Data se aktuálně do databáze uloží každých 10 minut, o což se stará cronjob (potažmo schedule) definovaný v app.js. Aktuálně si databáze pamatuje dvanáct záznamů, takže dvě hodiny do minulosti. Fakt, že log časový rozdíl mezi zápisy nemusí být přesně 10 minut, příliš nevadí, grafy, které se z těchto dat vykreslují jsou spíše orientační.

**projects** | id | name | owner_email | timestamp |

Tabulka projects si pamatuje všechny projekty, které spravuje náš systém. Cizím klíčem je owner_email, což je email vlastníka projektu, tedy člověka, který ho vytvořil. Timestamp je čas vytvoření projektu.

**projectsCoworkers** | project_id | user_email |

Tabulka projectsCoworkers zporstředkovává M:N vazbu mezi users a projects. Slouží k uložení lidí, kteří jsou spolupracovnící na jednom projektu. K datu odevzdání práce, nejsou spolupracovnící implementované, takže tabulka je aktuálně zbytečná. Řada metoda nepočítá s existencí spolupracovníků.

**projectsResourcesLimits** | project_id | ram | cpu | disk | upload | download |

Tabulka projectsResourcesLimits slouží k uložení limitů projektu. Nerozdíl od containersResourcesLimits můžou zde mít limity hodnotu null. V takovém případě je kontejnerům dostupný vešerý volný prostor, který uživatel má.

**templates** | code | id | profile_name | image_name | version | profile_description | image_description | profile_path | min_disk_size |

Tabulka templates slouží k uložení šablon, lde kterých se má vytvořit kontejner. Šablona je koncept, který se nenachází přímo v lxd, ale integruje dvě věci – profily a image. Image je distribuce jaká se na systém má nainstalovat. Profile je koncept z lxd, který obsahuje konfiguraci kontejneru. Eventuálně by měl uživatel možnost vytvářet si vlastní profily, které by obsahovali například konfiguraci networků.

**users** | id | email | given_name | family_name | icon | role | coins |

Tabulka users slouží k uložení uživatelů. Svoje data dostává z dat, které zasílá googleAuth 2.0. Sloupce role a coins aktuálně nemají význam, role bude dělit uživatele mezi standardního uživatel, admina a superadmin. Admin a superadmin by měli právo zasahovat do kontejnerů jiných uživatelů. Coins měl být sloupec sloužící u ekonomiky systému, k její implementaci však nedošlo.

**usersResourcesLimits** | user_id | ram | cpu | disk | upload | download |

Tabulka usersResourcesLimits ukládá limity uživatelů. Aktuálně má uživatel k dispozici 2.8 GHz[1], 1 GB ram, 8GB volného místa na disku, a download a upload 800kb/s.

[1]server kde byl program vyvíjen měl 2 jádra o 2.8 GHz, jedná se tak o polovinu výkonu

### 3.1.2 mongoDB

## 3.2 LXD

## 3.3 Networking

Následující sekce se zabývá networkingem kontejnerů a proxy, které umožňuje přístup na ně z venčí.

### 3.3.1 Konfigurace networků

Pomineme li loopback (standardní *lo*), tak každý kontejner má v aktuální verzi právě jeden networkový interface. Tím je *eth0*, který je na hostovi napojen na bridge *lxdbr0*. Přes něj mají kontejnery přístup na internet. Zároveň jsou všechny na stejné síti (prostřednictvím *lxdbr0*)

Jelikož jsou veškeré kontejnery na stejné síti, tak mezi sebou mohou komunikovat. Lxd pro tento účel samo nastavuje jejich interní doménu, ta je ve tvaru *c{id}.lxd*. Pochopitelně však nejsou dostupné z internetu, doména je pouze interní a veřejnou ip adresu nemají. Aby bylo možno na kontejner přistupovat bylo nutné nastavit proxy, které mu databáze bude přeposílat.

Nutno podotknout, že i traffic mezi kontejnery je aktuálně omezen limitem. V budoucno se tedy nabízí možnost implementovat možnost vytvářet další bridge a kontejnery si propojovat. Ty by se mohly ukládat do profilů, uživatel by si tak rovnou mohl vytvořit kontejnerů izolovaný od ostatních.

Fakt, že kontejnery jsou na stejné síti není rizikové ani problematické. Standartě se nemůžou nijak ovlivňovat.

### 3.3.2 Proxy

Pro proxy využívá projekt volně dostupnou haproxy, jejíž instance běží ve stejnojmenném kontejneru. Hostovací stroj totiž standardně nemůže přistupovat na kontejnery prostřednictvím .lxd domény. Bylo by možné sice mít proxy na hostovi serveru, ale z bezpečnostních důvodů tak nebylo učiněno.

Do kontejneru haproxy je aktuálně přesměrována traffic ze čtyř portů – 80, 443, 2222

a 3000. Port 80 pochopitelně slouží na webové stránky pomocí protokolu http. 443 je pro https, proxy má nastavený ssl certifikát. Ten stačí jeden pro celý systém, musí se však jednat o certifikát typu wildcard[2]. Systém byl testován na standardním certifikátu, čily připojení sice bylo zabezpečené, ale prohlížeč hlásil podezření z fishningu, jelikož se doména a doména na certifikátu neschodovali. Port 2222 slouží na připojení přes ssh, port 22 používá hostovací stroj a z pochopitelných důvodů není přesměrován do proxy. Port 3000 je dostupný pro REST API aplikací, které na serveru budou běžet.

Kotejnerům je automaticky přirazena doména v následujícím tvaru *{jméno kotejneru}.{jméno projekty}.{část emailu uživatel, před @}.{doména serveru}*. Například kontejner pojmenovaný *rumburak* v projektu *web* uživatele *belzebub@email.com* na serveru s doménou *avava.cz* bude mít doménu rumburak.web.belzebub.avava.cz. Mezery ve jméně kontejneru, či projektu se domény odstraní. Pokud by měl nový kontejner stejné jméno, jako jiný systém vyhodí výjimku. Tak je zaručena unikátnost domén, u mailu není takový problém nutno řešit, jelikož se jedná o školní emaily, které jsou vždy unikátní.

Bohužel je možno elegantně forwardovat pouze http protokol. Většina TCP protokolů totiž neoperuje s SNI a řídí se pouze doménu. Z packetu tak nejde zjistit na jakou doménu se uživatel snaží dostat a tedy jakému kontejneru mý být protokol přiřazen. To by se týkalo i ssh, příkaz ssh však podporuje nastavit pomocí flagu -o ProxyCommand (viz. 3.2), packety poté sni v hlavičce mají a je možno poznat jakému kontejneru patří.

```
ssh -o ProxyCommand="openssl s_client -connect $SERVERURL:2222
    -servername $CONTAINERURL" blank -l $USERNAME
```

Obrázek 3.2: Příkaz k připojení na server[3]

Porty jako 8443 aktuálně forwardovány nejsou, pokud uživatel potřebuje přistupovat do datáze běžící na kontejneru je možno použít ssh local port forwarding. Tento princip je i bezpečnější.

[2]certifikát, který zahrnuje i subdomény

[3]$SERVERURL je url serveru kde běží kontejnerizační systém, $CONTAINERURL je url kontejneru, $USERNAME je jméno uživatel, argument dummy je ignorován, je však potřeba

Konfigurační soubor HAProxy je generován v containerSQL.js. Po jeho vygenerování se prostřednictvím lxd.postFileToInstance pošle nová konfigurace to HAProxy kontejneru. Restartování proxy je téměř instantní operace i s velkým konfiguračním souborem obsahujícím desítky kontejnerů. Downtime je tak minimální. Podoba konfigurace byla realizována za pomocí dokumentace HAProxy[4]. Aktuálně má formát konfiguračního souboru jisté rezervy, specificky co se týče jeho délky. Je však o trochu rychlejší než, kdyby byly použity pokročilé metody mapování, které HAProxy podporuje.

[4] *HAProxy Enterprise Documentation 2.3r1.* HAProxy Technologies, LLC. Dostupné z `https://www.haproxy.com/documentation/hapee/latest/`. [cit 2021-13-4]

# Závěr

# Seznam použité literatury

*Oauth using react js | redux | node js | passport js | mongodb.* CODERS NEVER QUIT. Dostupné z `https://www.youtube.com/watch?v=gtg1-N7yfGM&ab_channel=CODERSNEVERQUIT`. [cit 2021-12-4].

*HAProxy Enterprise Documentation 2.3r1.* HAProxy Technologies, LLC. Dostupné z `https://www.haproxy.com/documentation/hapee/latest/`. [cit 2021-13-4].

# Seznam obrázků