# 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

Prohlašujeme, 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

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 vlastní linuxový server. Na kterém si poté mohou hostovat svoje aplikace, webové stránky. 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. 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** **v**

**1 Architektura** **1**

1.1 Specifikace vlastního API . . . . . 2

1.2 LXD . . . . . 7

1.2.1 Využité funkce LXD . . . . . 8

1.3 Autentifikace . . . . . 9

1.3.1 Princip fungování . . . . . 10

1.3.2 Implementace . . . . . 10

1.3.3 Autorizace . . . . . 11

**2 Frontend** **12**

2.1 Struktura souborů . . . . . 12

2.2 API . . . . . 16

2.3 Redux . . . . . 17

2.4 Navigace . . . . . 19

2.5 Přehled funkcí . . . . . 22

2.5.1 Hlavní panel . . . . . 22

2.5.2 Přehled projektů . . . . . 23

2.5.3 Informace o projektu . . . . . 26

2.5.4 Přehled kontejnerů . . . . . 27

2.5.5 Nastavení projektu . . . . . 28

2.5.6 Informace o kontejneru . . . . . 29

2.5.7 Terminál . . . . . 29

2.5.8 Backup . . . . . 30

2.5.9 Nastavení kontejneru . . . . . 31

**3 Backend** **32**

3.1 LXD . . . . . 32

3.1.1 Standard odpovědí z rozhraní s LXD . . . . . 32

3.1.2 Hlavní dorozumívací prvky . . . . . 33

3.1.3 Potíže s limity . . . . . 34
3.1.4 Zpřístupnění WebSocket terminálu . . . . . 34
3.2 Databáze . . . . . 36
3.2.1 MySQL . . . . . 36
3.2.2 MongoDB . . . . . 39
3.3 Networking . . . . . 39
3.3.1 Konfigurace networků . . . . . 39
3.3.2 Proxy . . . . . 40

**Závěr** **vi**

**Seznam použité literatury** **vii**

**Seznam obrázků** **viii**

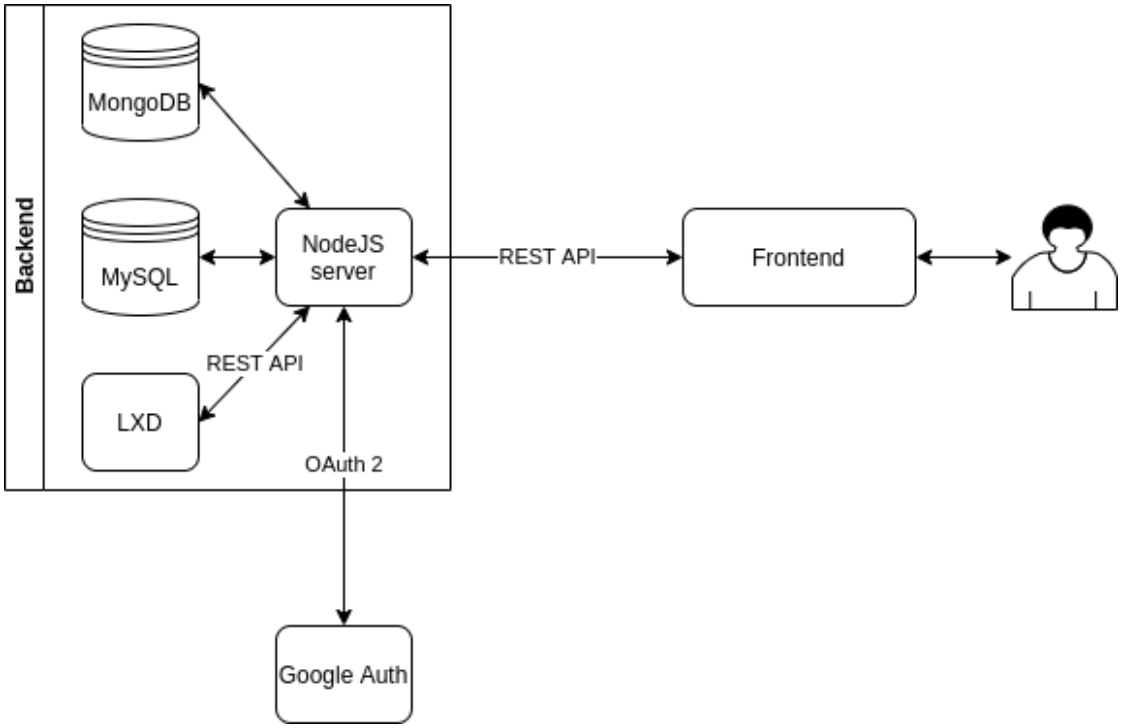
# Úvod

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 volný prostor, kde mohou testovat svoje aplikace. Aktuálně totiž bylo problematické jen získat přístup na školní server.

Co se týče rozdělení práce, tak frontend práce se staral Vladimír a 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. Všichni tři členové se během práce podíleli na vytváření Open API 3 specifikace určující nejen způsob komunikace mezi frontendem a backendem, ale také i funkce které má projekt vlastně mít.

Jednotlivé části byly vyvíjeny odděleně. Frontend byl testován na testovacím serveru, který byl vygenerován Swagger editorem, v němž se vytvářela API specifikace. Backend byl testován za pomocí jednoduchých skriptů umožňující posílat různé dotazy a požadavky. Komunikace probíhala přes vlastní Discord server a kód byl sdílen pomocí GitHubu – `https://github.com/havrak/AvAvA`, kde je aplikace dostupná pod licencí GPLv3. Pro vývoj kódu byl zřízen VPS server, na kterém jsou umístěny programy potřebné pro fungování backendu.

# 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 nejefektivnější řešení, protože jednotlivé kontejnery mohou sdílet společný kernel. Jsou tedy menší a rychlejší, než kdyby se jednalo o VPS (Virtual Private Server).

Hierarchie jednotlivých komponent projektu sestává z uživatelů, projektů, které těmto uživatelům náleží, a kontejnery, což jsou samotné virtuální servery, které si uživatel může pod hlavičkou projektu vytvořit, určené k testování, hostování webových stránek, či jiné. To umožňuje mít v serverech jistou organizaci, kde projekty představují třeba záložku a kontejnery pak její obsah. Zároveň také může uživatel omezit limity jednoho projektu, a získané prostředky přiřadit jinému projektu, kde třeba má testovací kontejnery, pro které potřebuje větší výkon atd.

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, ale kvůli komplikacím s LXD byla později zavedena i 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, získat informace o jejich aktuálním stavu a mnohé další.

## 1.1 Specifikace vlastního API

Následující sekce prochází jednotlivé routy. API bylo vytvořeno za pomocí nástroje `editor.swagger.io`. Jeho technická specifikace se nachází v souboru USER.yaml.

### GET – /api/combinedData

Routa GET /api/combinedData vrací objekt UserData. Odpověď obsahuje data 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 prvně 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 nejnovější 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. Došli jsem také k závěru, že snapshoty by jsou potenciální bezpečností riziko, jelikož nejde limitovat místo, které zabírají.

### 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} by měla obnovit stav kontejneru dle snapshotu. Stejně jako další routy týkající se logiky snapshotů ani tato nebyla implementována.

### 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.

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 ProjectStateWith History. 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.

## 1.2 LXD

Následující kapitola spěšně projede princip integrace LXD do našeho projektu. Programová stránka je později rozvedena v backednu.

Veškeré dorozumívání se projektu s LXD probíhá přes REST API[1]. Pro zjednodušení všech odchozích požadavků obsahuje backend soubor lxdRoute.js, který projektu poskytuje snadný a efektivní přístup k datům LXD. Tato kapitola uvádí výslednou strukturu komunikace mezi rozhraními, využité funkce poskytované LXD a s nimi spojené potřebné postupy – MongoDB a WebSocket komunikace.

#### Princip obecné komunikace s LXD

Všechny vnější požadavky na LXD prochází přes soubor lxdRoute.js, který je určen výhradně právě k tomuto účelu. Poskytuje metody pro každou jednoduchou akci potřebnou pro, jak zjištění/doplnění dat jistých objektů – projekty, kontejnery, jejich stav, zálohy nebo i snapshoty –, tak provedení určitých akcí – změnit stav/limity kontejneru, spustit příkaz/konzoli, nahrát/stáhnout soubory/zálohy... Výsledky jsou

[1] *LXD docs – REST API.* Canonical. Dostupné z `https://linuxcontainers.org/lxd/docs/master/rest-api`. [cit. 2021 14-4]

zpracovány do využívaných údajů, které se vrátí volajícímu. Tím úloha prostředníka mezi backendem a LXD démonem končí.

### 1.2.1 Využité funkce LXD

Celkové množství funkcí, které LXD poskytuje, zdaleka převyšuje potřebu tohoto projektu, je proto nutné uvést, kterých prvků a rout bylo nakonec využito. Některé funkce byly v plánu, avšak, jak jsme posléze zjistili, jsou špatně implementovány a jejich použití v lepším případě nemá význam, v horším pak vyvolává náhodné chování kontejnerů (viz Práce s limity).

Základními nezbytnými funkcemi jsou akce s kontejnery a vytváření a mazání projektů; funkce, jako vytváření profilů, .img souborů, clustery nebo práce se šablonami (templates), nebyly využity, neboť se veškerá omezení nastavují pomocí limitů a zdroje pocházejí z jednoho serveru, používá se proto všude jeden disk a jedno internetové připojení. Výběr z obrazů operačních systémů je také jen jeden – není důvod omezovat projektům výběr systému používaného pro kontejnery.

Oproti funkcím projektu, kde využíváme jen routy na vytvoření a smazání, jsou u kontejnerů užívané téměř všechny routy. Většinou se ale jedná o jednoduché poslání dotazu na stav nebo měnění stavu kontejneru.

#### Zakomponování LXD do MongoDB

Zjišťování stavu se jen lehce komplikuje při vypnutém kontejneru, jelikož LXD v tomto stavu nevrací data o internetu a využitém disku. Pročež je potřeba chybějící data uložit, ještě když jsou k dispozici. Za tímto účelem byl využit databázový framework MongoDB.

#### Práce s limity

Náročnějšími funkcemi jsou pak používání virtuálního terminálu kontejneru (viz Komunikace přes WebSocket), a také nastavování limitů, což tvoří jednu z nejkomplikovanějších částí projektu, která vyžaduje vícero dalších, dotazu předcházejících operaci.

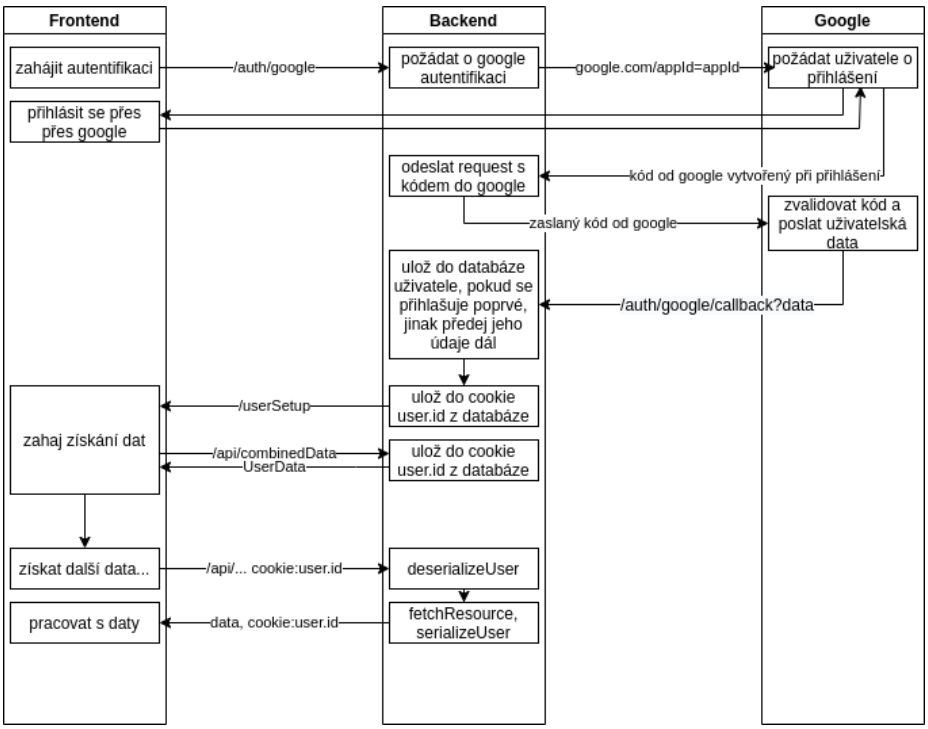
Zdroje, které jsou k dispozici kontejnerům, se vytváří podle limitů projektu, přesněji jejich zbývající části, která je dostupná. Limity projektů se sestavují ve stejném vztahu

k uživateli, jehož limity jsou předdefinované v základu stejně pro všechny uživatele. I uživatel si ale může navýšit své limity, a to zažádáním o navýšení limitů.

**Využití komunikace přes protokol WS (websocket)**

Pro jisté funkce bylo třeba zavést také používání protokolu WS. Dokumentace LXD uvádí, že je možné provádět veškeré operace tímto stylem, avšak pro jednoduchost a přehlednost byla tato možnost ponechána pouze pro nezbytné prvky projektu, kterými jsou funkce terminálu v kontejnerech a aktualizování rozměrů okna terminálu v LXD, aby byl výsledný text u klienta správně zobrazován. Tyto dvě funkce sdílejí stejnou cestu, na kterou se lze připojit, rozlišují se pomocí id websocketu – /websockets/terminals/[id websocketu].

## 1.3 Autentifikace




Obrázek 1.2: Autentifikace přes Google Auth

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. Při první přihlášení se 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[2]. 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

[2] *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 2021-13-4]

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.

### 1.3.3 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 Google Auth, 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 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 mít právo cizí kontejnery do určité míry ovlivňovat. Konkrétně admin na ně pouze nahlížet a superadmin s nimi volně zacházet a měnit je.

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

# 2. Frontend

Základem pro frontend se stal dashboard template[1]. Požadavky aplikace ovšem tento template ani zdaleka nedokázal splnit. Jedinými částmi, které tedy přebrány je layout aplikace (soubor User), design Sidebaru a Navbaru, včetně jejich mobilních verzí. I tyto části ovšem musely být modifikovány, aby vyhovovaly všem nárokům aplikace. Zároveň také z templatu zůstaly scss soubory. Ty jsou ovšem využívány některými komponentami, takže zatím soubory nebyly z projektu vymazány.

Kód byl napsaný v Javascriptu, konkrétně ReactJS a preprocessoru SASS. Velice důležité jsou i knihovny react-bootstrap a material-ui, pomocí nichž byla napsána většina komponent.

## 2.1 Struktura souborů

Následující sekce popisuje strukturu souborů clienta.

```
client
├── node_modules
├── public
└── src
    ├── actions
    ├── api
    ├── assets
    ├── components
    ├── layouts
    ├── reducers
    ├── service
    ├── views
    ├── App.js
    ├── index.js
    ├── logo.svg
    ├── routes.js
    └── setupProxy.js
```

Obrázek 2.1: Struktura souborů frontend

[1] viz. *Dashboard template.* Create-Tim. Dostupné z `https://github.com/creativetimofficial/light-bootstrap-dashboard-react`. [cit. 2021-13-4]

### Public

```
client
└── public
    ├── favicon.ico
    └── index.html
```

Obrázek 2.2: Struktura souborů frontend – public

Ve složce public se nachází pouze jeden HTML soubor s root divem, do kterého React zapisuje. Favicon je poté LXD ikona, která se objeví jako ikona karty prohlížeče.

Soubory .eslintcache, package=lock.json jsou automaticky generované a nejsou tedy důležité. Soubor gulpfile.js připne k HTML a CSS souborům hlavičku, kterou vyžaduje licence templatu. Soubor jsconfig.json poté dovolí importovat soubory pomocí absolutních cest vzhledem k src složce, což činí práci s importy mnohem pohodlnější

### Src

```
client
└── src
    ├── actions
    ├── api
    ├── assets
    ├── components
    ├── layouts
    ├── redurcers
    ├── service
    ├── views
    ├── App.js
    ├── inedex.js
    ├── logo.svg
    ├── routes.js
    └── setupProxy.js
```

Obrázek 2.3: Struktura souborů frontend – src

Zdaleka nejdůležitější je ovšem src složka, která obsahuje samotný kód. Půjdeme-li popořadě, tak první složkou je actions, kde se nacházejí akce manipulující s Redux storem (viz kapitola 2.3). S ním také manipulují reducery ve složce redurcers.

### Api

```
client
└── src
    └── api
        ├── api
        ├── model
        ├── ApiClient.js
        ├── index.js
        └── WebSockets.js
```

Obrázek 2.4: Struktura souborů frontend – api

Ve složce api se nachází kód na získání a odesílání dat na backend (viz kapitola 2.2).

### Assets

```
client
└── src
    └── assets
        ├── css
        ├── fonts
        ├── img
        └── scss
            ├── custom
            ├── lbd
            └── lbdr
```

Obrázek 2.5: Struktura souborů frontend – assets

Ve složce assets se nachází vše, co se týká stylů, obrázků a fontů potřebných pro projekt. Kód napsaný v assets/scss se díky preprocessoru zkompiluje do css souborů do složky assets/css, které se poté odesílají spolu s HTML souborem. Většina stylů byla zachována z templatu, jelikož jsou styly vcelku použitelné i na požadavky projektu. Například díky nim funguje responzivní navigace.

V některých případech bylo ovšem nutné napsat vlastní styly. Ty jsou uloženy ve složce assets/scss/custom. Do souboru light-bootstrap-dashboard je poté přidán import každého z custom souborů.

### Components

```
client
└── src
    └── components
        ├── Cards
        ├── Dialogs
        ├── Footer
        ├── Form
        ├── Icons
        ├── Navbar
        ├── RouteComponents
        ├── Sidebar
        ├── Tables
        ├── Terminal
        ├── ProjtecedRoute.js
        └── UserSetup.js
```

Obrázek 2.6: Struktura souborů – components

Složka components obsahuje znovupoužitelné komponenty, ze kterých se poté skládají jednotlivé stránky. Na ní přímo navazuje složka views, v níž se nachází kód, který jednotlivé komponenty spojuje do stránek. Když tedy uživatel zadá do URL baru cestu, zobrazí se mu právě jedna stránka složená z několik komponent. Tyto stránky jsou poté obaleny ještě do tzv. layoutů nacházejících se ve složce layout. Stránky jsou v nich spojeny s navigacemi a zápatím. Momentálně existuje pouze jeden layout a to layout pro uživatele (User.js). V budoucnu je ovšem plánováno rozšířit také projekt pro administrátorský přístup, layouty tedy přibydou.

### Service

Ve složce service se nacházejí služby, které mohou využívat jakékoliv další služby. Jedná se o různé druhy výpočtů, např. převádění jednotek, počítání stavů grafů atd...

Soubor index.js je nejvyšším souborem, který do root divu v HTML vloží celý frontend. Zároveň obsahuje kód pro integraci reduxu. Soubor App.js obsahuje komponentu, která seskupuje všechny layouty a dovoluje a podílí se na navigaci na frontendu (viz kapitola 2.4). Stejně tak si v kapitole probereme i soubor routes.js.

**View**

```
client
└── src
    └── views
        ├── Containers
        │   ├── Console.js
        │   ├── Container.js
        │   ├── Info.js
        │   ├── Snapshots.js
        │   └── StateHistory.js
        ├── Other
        ├── Projects
        │   ├── Container.js
        │   ├── Info.js
        │   └── Project.js
        └── User
            ├── Dashboard.js
            └── Project.js
                └── UserProfiel.js
```

Obrázek 2.7: Struktura souborů – view

Ve složce views sídlí jednotlivé stránky. Ty jsou v ní dále rozděleny podle toho, do jaké kategorie spadají. Zde jsou vidět všechny tři kategorie. Ve složce Other se poté nacházejí stránky z templatu pro případ, kdy by bylo možné někdy použít některé z template komponentů.

## 2.2 API

Veškerý kód, který má něco společného s komunikaci s backendem či jinými externími zdroji sídlí ve složce api. Obsah této složky kromě WebSockets.js je generován pomocí programu swagger-codegen v již zmíněném Swagger editoru.

Ve složce api/api se poté nachází soubor, kde jsou vygenerované metody odpovídající cestám na REST API, těmto metodám stačí předat specifikované parametry a na server se odešlou požadovaná data. Po získání odpovědi je zavolán předaný callback.

Swagger codegen vygeneroval soubory, které jako API klienta používají knihovnu superagent. Jeho chování je specifikováno pomocí vygenerovaného souboru ApiClient. Vygenerovaný kód ovšem nestačí, např. pokud by vrácená odpověď byla 401, tak bylo nutné do souboru přidat příkaz na spuštění autentifikace.

Ve složce models se nachází soubory odpovídající objektům z Open API specifikace. Na frontendu je ovšem vůbec nevyužívám, slouží tedy pouze samotnému fungování API.

Vše je poté přehledně dostupné v souboru index.js, ze kterého jsou exportovány všechny objekty z model složky a cesty z api/api/DefaultAPI.js

Posledním souborem je WebSockets.js, ve kterém se nachází metody vytvářející WebSocket spojení na backend. Momentálně se WebSockety využívají pouze pro vytvoření spojení pro terminál do kontejnerů, ale v budoucnosti je očekávané, že se jejich využití rozšíří. V případě, že bude spolu na jednom projektu spolupracovat více uživatelů, bude nutné pro vytvoření real-time systému updatovat změny pomocí duplexního spojení, kterým je právě WebSocket.

Jelikož při developmentu nejsou backend a frontend na jednom portu, dochází ke CORS erroru. Na jeho vyřešení byla implementována http proxy (soubor setupProxy.js), která veškeré specifikované requesty přesouvá na port 5000 se stejnou adresou.

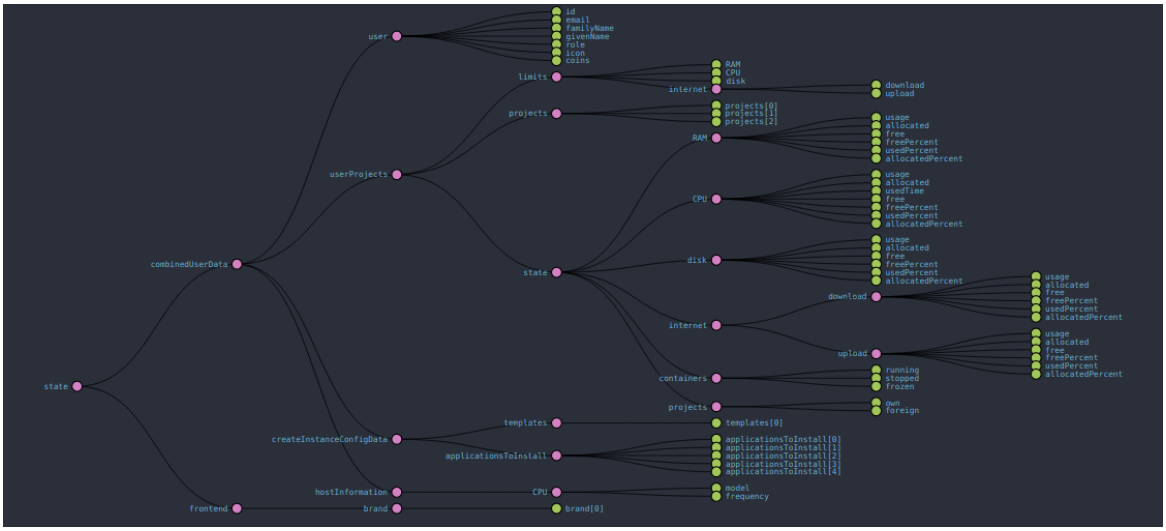
## 2.3 Redux

Redux je framework, který dovoluje různým React komponentům sdílet stav pomocí tzv. Redux store. V případě, že nějaký komponent potřebuje přistoupit k nějaké části storu, tak se napíše metoda mapStateToProps a jednotlivé proměnné jsou poté předány jako properties jednotlivým komponentám. V případě, že dojde ke změně redux storu, dojde k překreslení daného komponentu.

Základní chování Redux storu je, že se vynuluje při načtení stránky. Z tohoto důvodu se store ukládá do localstorage při každé jeho změně. Při načtení stránky se poté z localstorage načte zpět. O inicializaci a chování stavu se stará soubor index.js v src složce.

Se storem je možné manipulovat pouze pomocí tzv. akcí. Jedná se o objekty, které obsahují jméno a payload. Tyto akce se nacházejí ve složce actions v odlišných souborech pro uživatelské akce, akce s projekty, akce s uživateli a frontendové akce. Tyto akce mohou být tzv. dispatchnuty, čímž se dostanou k reducerům.

Reducer je metoda, která na základě jména a payloadu předané akce určí, co se má stát s Redux storem. Tento způsob změny stavu se může zdát zvláštní, každopádně




Obrázek 2.8: Graf stavu u Reduxu

```
client
└── src
    └── actions
        ├── ContainerAction.js
        ├── FrontendAction.js
        ├── ProjectAction.js
        └── UserAction.js
```

Obrázek 2.9: Struktura souborů frontend – actions

zajišťuje vynikající škálovatelnost systému. V současnosti existují v projektu dva reducery, combinedUserData reducer, který se stará o veškeré akce související se změnou dat definovaných pomocí Open API a frontend reducer, který má pouze jediný účel na frontendu. V budoucnosti pravděpodobně nebude nutné mít více než jeden reducer, jelikož logika, kterou frontend reducer vykonává pravděpodobně bude změněna do podoby, kdy frontend reducer nebude potřeba.

Akce mohou být i asynchronní, což se např. hodí pro případ, kdy chceme poslat request na API a jakmile získáme výsledek, tak udělat se získanými daty nějakou akci - např. akce containerIdDelete() ihned zobrazí u mazaného kontejneru v tabulce kontejnerů spinner a nápis deleting. Po smazání kontejneru se odešle na frontend odpověď a vykoná se akce smazání kontejneru z tabulky.

Jednotlivým komponentům jsou akce předávány pomocí implementace metody mapDispatchToProps, která funguje stejně jako mapStateToProps.

Hlavní účel využití Reduxu v projektu je tedy zpracování dat, které se získají z

API do podoby, se kterou mohou komponenty pracovat a poskytnutí možnosti tato data měnit s okamžitými účinky na frontendu.

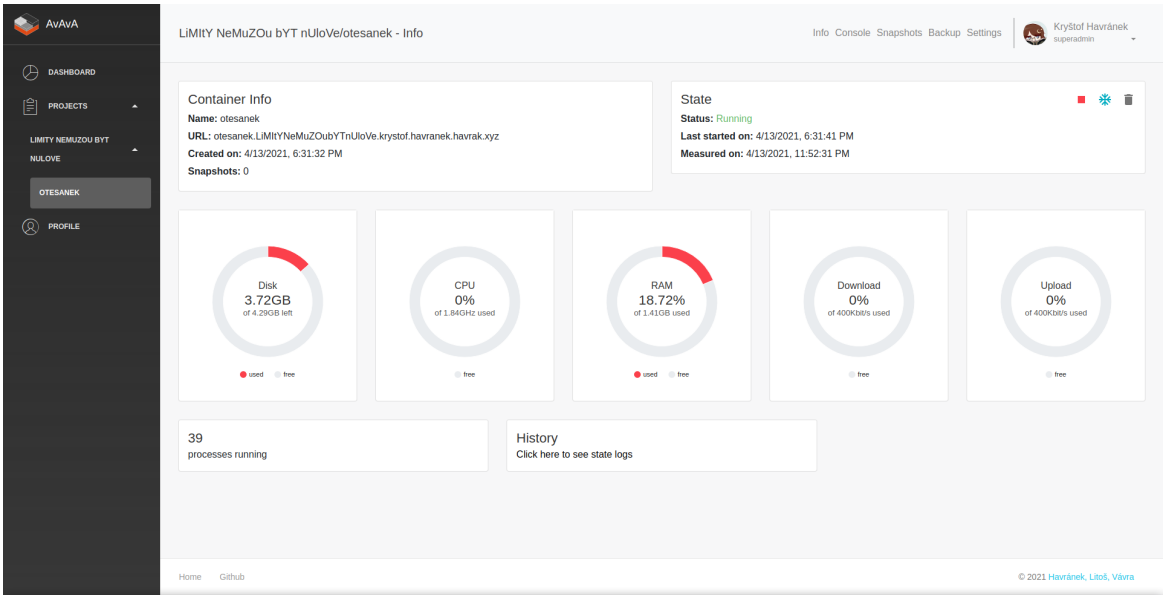
## 2.4 Navigace

O navigaci na frontendu se stará knihovna react-router. Každá cesta na frontendu zobrazující nějakou stránku začíná prefixem daného layoutu. V případě, že si tedy budeme chtít zobrazit hlavní panel, cesta na získání bude /user/dashboard. Tím se vždy odliší od volání do api, které má prefix /api.

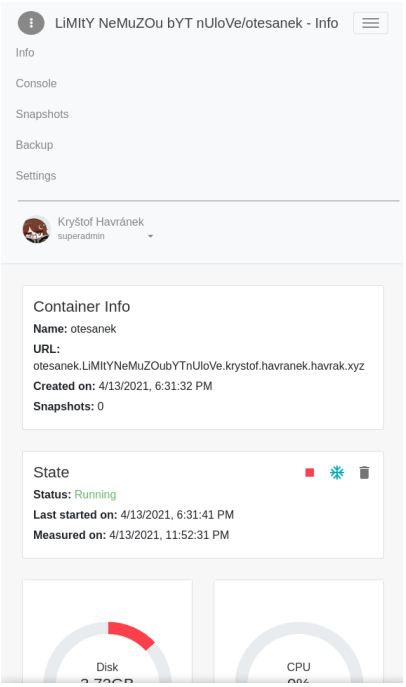
V projektu se momentálně nachází 1 + N BrowserRouterů. Jeden v app.js, který se stará o layouty a N v rámci jednotlivých layoutů (nyní N=1). V nich jsou vytvářeny Route objekty s adresami specifikovanými v routes.js souboru. V něm se u každé cesty nachází také jméno dané cesty, layout, do kterého patří, navigační linky a stránka, která se má zobrazit, když uživatel zadá danou cestu.

V případě, že uživatel není přihlášen a pokusí se získat chráněný obsah, je okamžitě přesměrován na přihlašovací stránku. Chráněným obsahem jsou přitom na frontendu s výjimkou jedné cesty úplně všechny. V případě, že je v Redux storu proměnná combinedUserData.user null, či undefined, tak uživatel není přihlášen. React-router ovšem neobsahuje žádný komponent ProtectedRoute, takže bylo nutné ho vytvořit. Jedná se o klasický Route objekt kontrolující, zda je výše jmenovaná proměnná null či nikoliv.

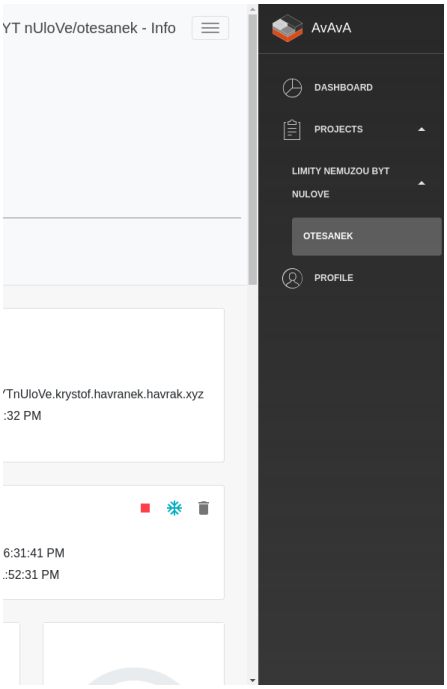
Pro snadný pohyb po stránce byly implementovány dva druhy navigace. Jak Navbar, tak i Sidebar jsou plně responzivní a přispůsobí se mobilnímu rozhraní, jak ve vidět na obrázcích 2.14, 2.11 a 2.14.




Obrázek 2.10: Uživatelské rozhraní na počítači




Obrázek 2.11: Responsivní interface na mobilu – overview




Obrázek 2.12: Responsivní interface na mobilu – sidebar

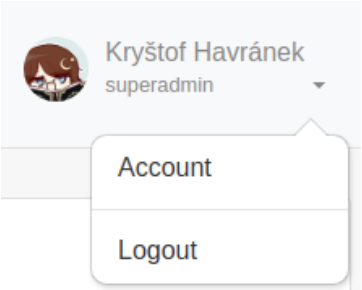
Frontend je strukturován do 3 základních kategorií:

- uživatelská - Dashboard, Projects
- projektová - Info, Containers, Settings

- kontejnerová - Info, Console, Snapshots, Backup, Settings

Sidebar - slouží převážně na přepínání mezi kategoriemi. Dovoluje tedy jednoduše navigovat mezi projektu a jejich kontejnery. Platí, že vždy je zvýrazněna světlým pozadím ta část sidebaru, která je momentálně zobrazovaná. Jednotlivé projekty či kontejnery v daném projektu je možné v sidebaru rozbalit či skrýt. Platí přitom, že nejde skrýt ta část, ve které je aktuálně zobrazený element. Když je zadána nějaká cesta, ale daný element je zabalený, tak se rodič automaticky rozbalí.

Navbar naproti tomu má za účel navigovat převážně mezi funkcemi jedné kategorie. Například na výše uvedeném obrázku jsou vidět funkce pro kategorie kontejner - Info, Console, Snapshots, Backup, Settings. Zároveň se zde nachází i karta s aktuálním uživatelem, na kterou když uživatel klikne, tak mu dropdown nabídne možnost odhlásit se (viz. obrázek 2.13).




Obrázek 2.13: Odhlášení uživatel

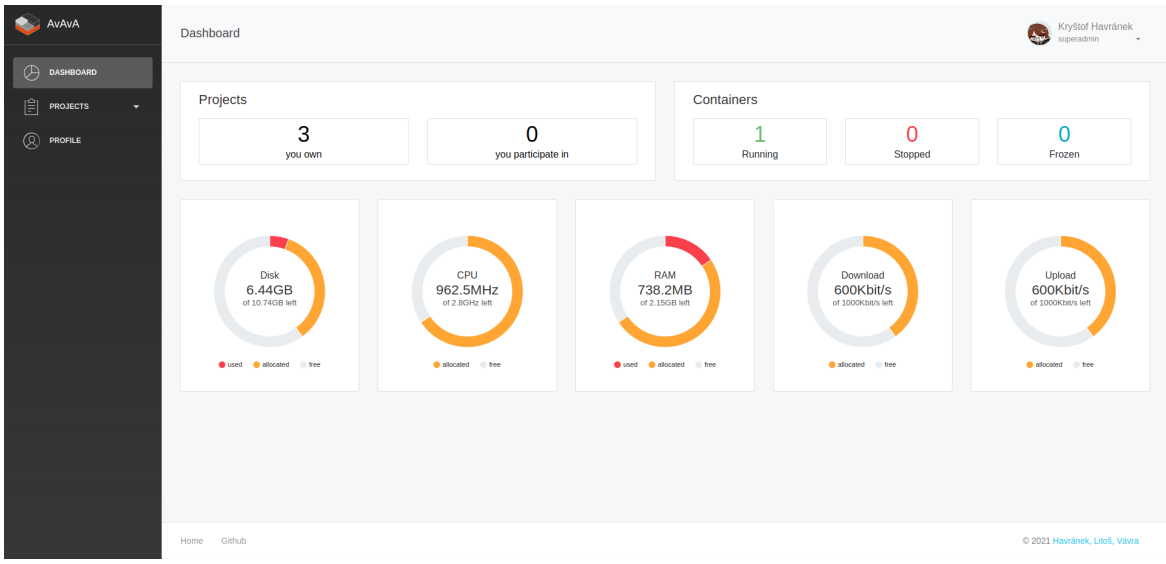
Vlevo se poté nachází jméno dané stránky - u kontejneru je to jméno projektu/jméno kontejneru - jméno stránky. Zajímavostí je, že při kliknutí na jméno projektu je uživatel odkázán na Info stránku rodičovského projektu.

U obou těchto navigací platí, že jsou implementovány pomocí linků, které odkážou na nějakou cestu, jež se vykreslí za pomoci react-routeru. Aby bylo jisté, že má uživatel vždy aktuální data, tak se při vykreslování jednotlivých cest vždy dispatchne nějaká akce, která pošle API request na server a po získání dat updatuje Redux store. Nejprve se tedy vykreslí stará data a jakmile přijde odpověď, tak se vykreslí nová. To dává pocit kontinuity a je to mnohem více uživatelsky přívětivé, než kdyby se čekalo na získání dat po každém kliknutí. V budoucnosti, když se bude řešit kooperace více uživatelů se tento systém pravděpodobně změní ve prospěch nějaké formy kontrolního WebSocketu.

## 2.5 Přehled funkcí

Tato kapitola je věnována detailnímu popisu jednotlivých stránek a komponentů, které potřebují pro své fungování.

### 2.5.1 Hlavní panel

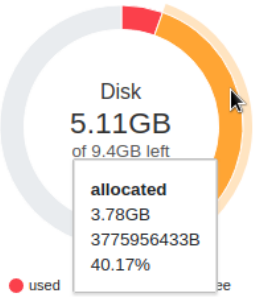



Obrázek 2.14: Uživatelské rozhraní na počítači

Dashboard, neboli hlavní panel zobrazuje uživateli aktuální informace o stavu všech jeho kontejnerů a projektů. Jedná se o první stránku, kterou po přihlášení uvidí. Nahoře se nachází dvě karty zobrazující počet projektů, které vlastní a počet různých stavů všech jeho kontejnerů. Nachází se zde také zobrazení počtu projektů, ve kterých se pouze účastní. Zatím sice není implementována možnost spolupráce více uživatelů na jednom projektu, ovšem její vytvoření je plánované a jelikož Dashboard vypadá lépe s touto funkcí, tak je tato informace zobrazena již nyní.

Ve druhé řadě se nachází 5 grafů (viz Obrázek 2.15) zobrazující agregovaný stav všech projektů. Na jejich implementaci byla použita knihovna react-google-charts - pie-chart. Červenou barvou se zobrazují zdroje, které jsou opravdu zkonzumovány jednotlivými kontejnery. Jedná se tedy o součet usage proměnných všech kontejnerů.

Oranžově jsou poté zobrazeny zdroje, které jsou přiděleny jednotlivým projektům či přímo jednotlivým kontejnerům v případě, že samotný projekt limit nemá pro daný




Obrázek 2.15: Detail grafu

zdroj. Tyto zdroje jsou zablokované a nejde je využít v jiném projektu či kontejneru než tam, kde jsou přiděleny.

Šedivě je zobrazeno volné místo, které je možné alokovat pro další projekty či kontejnery. Toto volné místo je přitom zobrazeno uvnitř samotného grafu v adekvátních jednotkách. V řešení problému umístění textu na střed grafu sehrál roli příspěvěk na StackOverflow[2]. Dashboard zobrazuje uživatel hlavně proto, aby měl přehled o zdrojích, které může projektům alokovat, dává tedy největší smysl zobrazit právě hodnotu volného místa. O převod ze základních jednotek se stará třída UnitsConvertor ve třídě service. Předtím se ještě spočítat všechna potřebná data – volné zdroje, alokované zdroje, procenta, atd. O to se stará třída service/StateCalculator.js, jejíž metody modifikují v reducerech stav Redux storu. Z toho plyne, že data z Redux storu již budou mít všechna potřebná data pro zobrazení stavu k dispozici.

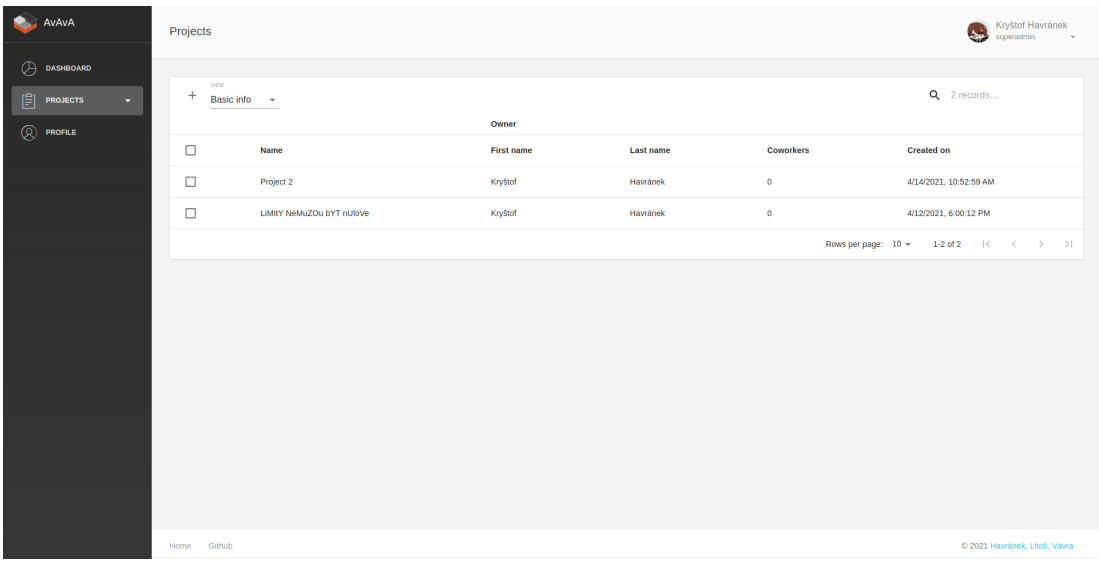
Komponenty grafů se nacházejí ve složce components/Cards/State. Podobný hlavní panel jako Dashboard pro uživatelskou kategorii se nachází i v projektové a kontejnerové kategorii. Tam ovšem fungují grafy odlišně, a tak se v této složce nacházejí ještě další dva soubory s odlišnými grafy.

### 2.5.2 Přehled projektů

Na této stránce má uživatel přehled o všech svých projektech pomocí tabulky. Ta využívá knihovny react-table a jedná o tabulku od material-ui[3] . Přejatý kód umož-

[2] *Display total in center of donut pie chart using google charts?* Kevin Brown. Dostupné z `https://stackoverflow.com/questions/32256527/display-total-in-center-of-donut-pie-chart-using-google-charts`. [cit. 2021 14-4]

[3] *Materia-UI.* react-table. Dostupné z `https://react-table.tanstack.com/docs/examples/material-ui-components`. [cit. 2021 14-4]

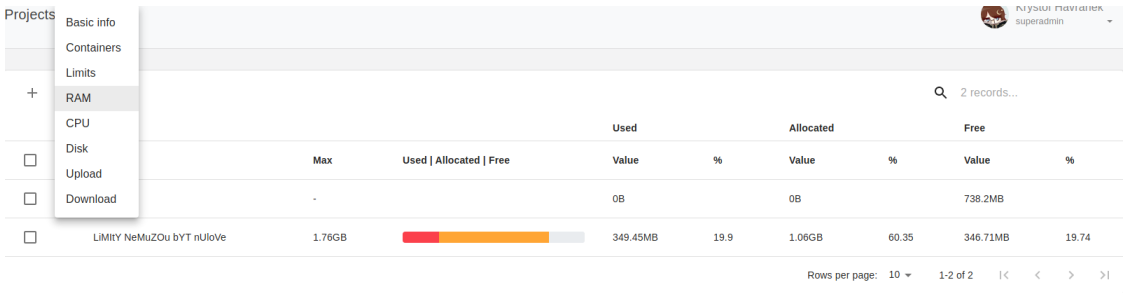



Obrázek 2.16: Přehled projektů

ňuje vybírat řádky na základě selectu, vyhledávat v tabulce pomocí globálního filtru vpravo nahoře (hledání ve všech sloupcích), řazení dle hodnot ve sloupci a také zajišťuje stránkování, kdy se na další stránku dá přejít pomocí tlačítek vpravo dole.

Stránku reprezentuje soubor views/User/Projects.js a tabulku reprezentují komponenty ve složce components/Table. Tabulky se nyní vytváření pomocí instancování komponentu EnhancedTable, které je předán Toolbar pro danou tabulku (bar s funkcemi nad daty) a samotná data. Aby bylo zajištěno, že data jsou aktuální, tak při každém vykreslení stránky se pošle GET request na /api/projects.

Vlastním přínosem do kódu bylo vytvoření SelectBoxu vlevo v Toolbaru, který dovoluje změnit pohled tabulky. Na obrázku níže (obrázek 2.17) můžete poté vidět seznam všech různých pohledů a také příklad jiného pohledu, konkrétně RAM.

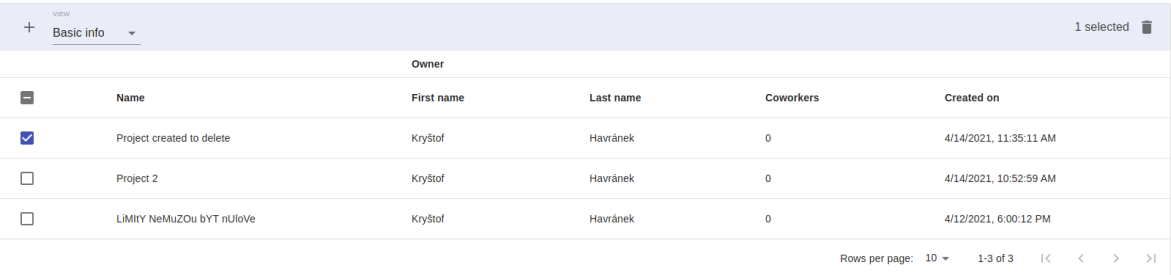



Obrázek 2.17: Pohled na ram

Pohledy RAM, CPU, Disk, Upload, Download zobrazují podobné informace jako jednotlivé grafy v sekci Informace o projektu (kapitola 2.5.3). Je ovšem výhodné je

vložit i do tabulky, jelikož lze kontejnery řadit podle volných, alokovaných či uložených zdrojů a díky tomu je při nedostatku místa snadné hledat projekty, ve kterých je možné ušetřit nějaké zdroje. Jestliže projekt nemá nastavené limity, nezobrazuje se progress bar ani procenta u žádné z hodnot, jelikož by to nemělo větší smysl. Zobrazují se ovšem stavy v daných jednotkách.

Uživatel může smazat více projektů najednou. Stačí vybrat projekt pomocí tlačítka nalevo a Toolbar se změní - zfialoví a zobrazí se počet zobrazených projektů následovaný tlačítkem Delete (viz obrázek 2.18).



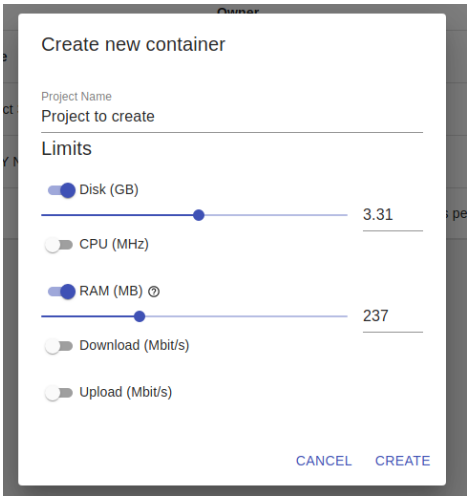

Obrázek 2.18: Výběr projektu

Při kliknutí na ikonu delete se zobrazí dialog, kde se uživatele aplikace zeptá, zda chce smazat tento projekt. Při kliknutí na Yes se vyšle akce na server a zároveň se změní Redux store. Díky tomu se místo select boxu zobrazí spinner (viz obrázek 2.19) s aktuální akcí.

Obrázek 2.19: Smazání projektu

Jakmile přijde ze serveru zpráva o smazání, okamžitě se kontejner smaže. Vytvořit projekt je možné pomocí tlačítka + úplně nalevo Toolbaru. Po stisknutí se otevře následující

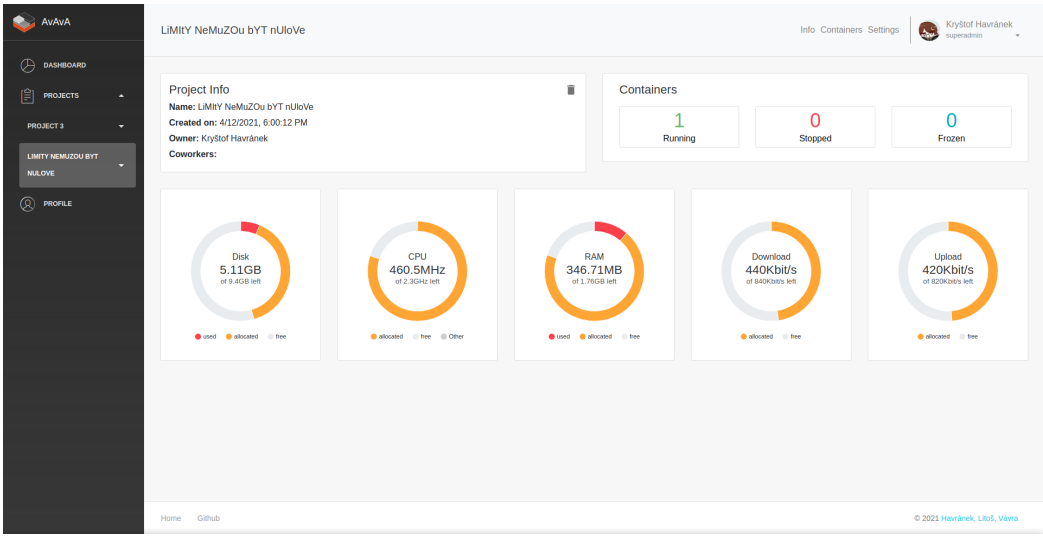
Při vytváření projektu je nutné specifikovat jeho jméno, které nesmí být delší než 30 znaků a nesmí se v listu projektů opakovat. Zároveň je možné specifikovat také limity. V případě, že se stiskne switch, je nastaven limit projektu na specifikovanou hodnotu, v opačném případě projekt nebude mít daný zdroj limitován a bude při výpočtech použit limit uživatele daného zdroje. Maximální limity jsou určeny jako volné místo ve stavu uživatele.




Obrázek 2.20: Vytváření projektu

Po kliknutí na create se zobrazí stejný spinner jako u smazání, akorát s textem creating.

### 2.5.3 Informace o projektu

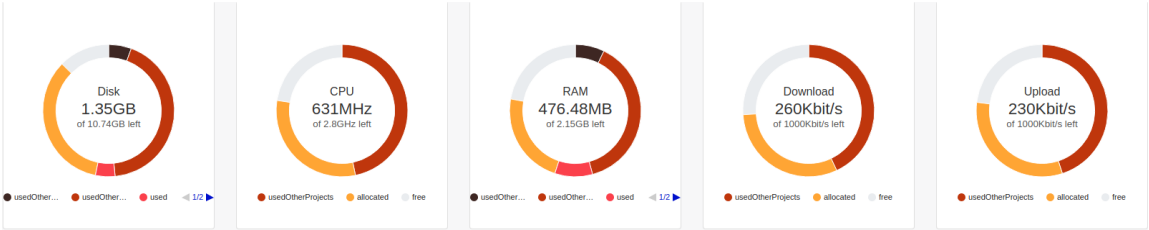



Obrázek 2.21: Informace o projektu

Stránka pro informace projektu je první ze stránek kategorie projektů. Na první pohled vypadá podobně jako Dashboard. Vpravo nahoře je vidět počet kontejnerů v projektu a dole jsou grafy, které fungují úplně na stejném principu jako grafy v Dashbaord.

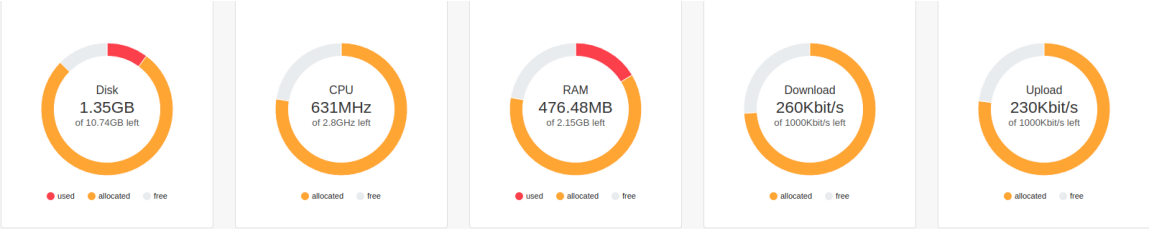
Liší se však v případě, že kontejner nemá nastavené limity u některého ze zdrojů. V takovém případě se zdroje kontejnerů vztahují přímo k uživatelským limitům. Platí

tedy, že součet limitů zdroje projektů a limitů zdrojů kontejnerů v projektech, kde limity zdroje nejsou musí být menší než limit zdroje uživatele. Grafy v takovém případě vypadají takto. Oranžově jsou zobrazeny alokované zdroje pro kontejnery v daném projektu, červeně využité zdroje kontejnerů v projektu, tmavě hnědé jsou alokované zdroje uživatele u ostatních projektů a černě využité zdroje ostatních projektů.




Obrázek 2.22: Alokace zdrojů u projektů

Všimněme si, že volná část zdrojů daného projektu musí být stejná jako volná část daného uživatelského zdroje.




Obrázek 2.23: Alokace zdrojů u uživatele

Další rozdíl je v levé horní kartě, která zobrazuje základní info o projektu. Pro budoucí využití je také zobrazován Owner a Coworkers v daném projektu, přestože tyto funkce budou implementovány až v budoucnu. Nachází se zde také tlačítko Delete, které funguje stejně jako tlačítko Delete v předchozí kapitole.

Oproti základní stránce také přibyly linky v navigaci, pomocí nichž se dá navigovat v rámci projektové kategorie. Info ukazuje na tuto stránku, Containers na přehled kontejnerů a Settings spustí dialog na nastavení projektu.

### 2.5.4 Přehled kontejnerů

Principielně se stránka nijak neliší od přehledu projektů. Zobrazovány jsou základní informace, limity a stav kontejnerů. Liší se hlavně funkcemi, které nabízí Toolbar. Ten je zde rozšířen o funkce start, stop a freeze.

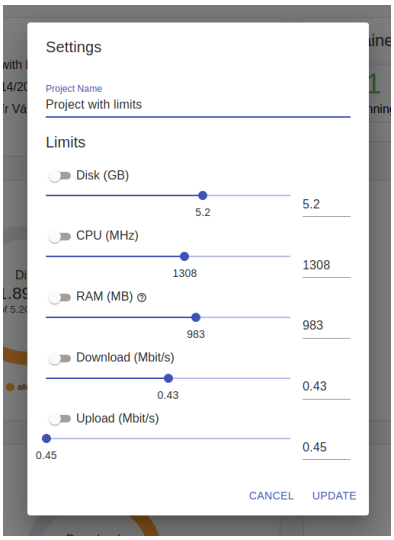
Obrázek 2.24: Toolbar pro kontejner

Po kliknutí na jednu z těchto ikon se vyvolá daná akce pro všechny označené kontejnery, u kterých je daná akce vhodná. V případě, že se např. pokusí uživatel nastartovat již běžící kontejner, akce se neprovede. Forma, kdy se zobrazí pouze takové akce, které jsou vhodné, není možná kvůli možnosti vykonat akci pro více kontejnerů najednou.

Dialog pro vytvoření kontejneru je téměř totožný jako dialog pro vytvoření projektu. Nastavení limit je ovšem povinné, a tak jsou odstraněny switche. Jsou také přidány inputy pro heslo na root účet v daném kontejneru, checkbox button list pro základní aplikace, které je možné nainstalovat a radio button list pro výběr templatu. Zobrazí se přitom pouze ty templaty, jejichž minimální usage je menší než aktuální volné místo, které kontejner může potenciálně zabrat.

### 2.5.5 Nastavení projektu

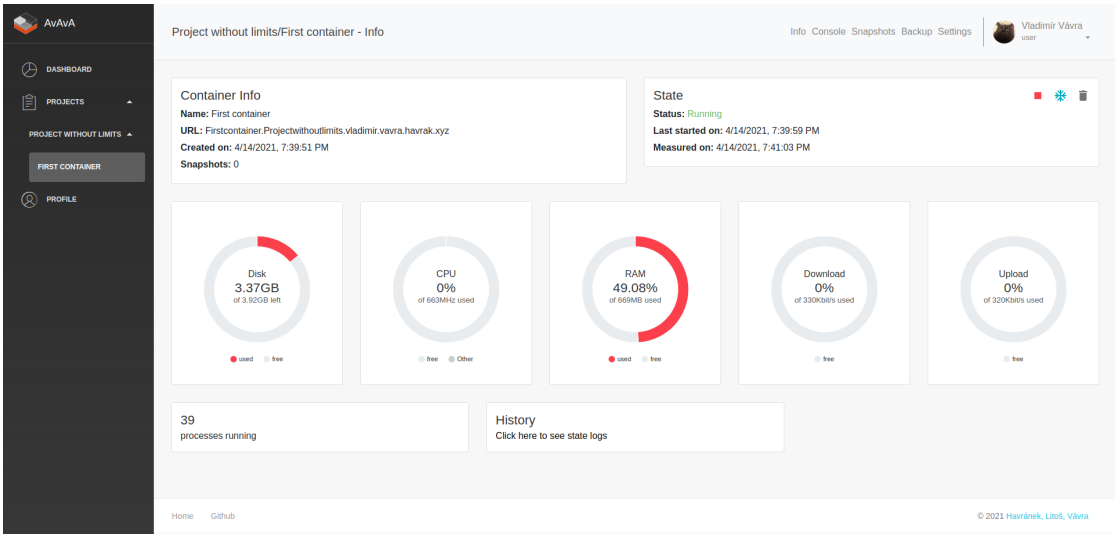
Přes link settings v navigaci může uživatel změnit nastavení daného projektu.




Obrázek 2.25: Nastavení projektu

Na začátku se v dialogu zobrazí aktuálně nastavené jméno a limity, kdy počáteční limit je reprezentován labelem na slideru. Pokud se klikne na tlačítko update, odešle se žádost na server o změnu.

### 2.5.6 Informace o kontejneru




Obrázek 2.26: Informace o kontejneru

Stránka informací o projektu stojí opět na stejném základě jako ostatní stránky s informacemi. Nahoře se nachází karty s informacemi a tlačítky pro ovládání. Narozdíl od tabulky kontejnerů se zde tlačítka pro změna stavu mění na základě aktuálního stavu. Při spuštěném kontejneru tedy nemůžete kliknout na tlačítko start.

Liší se také v grafech. Zatímco předešlé informační panely zobrazovaly, kolik zdrojů zbývá, zde je tomu kromě disku naopak, jelikož není potřeba zobrazovat alokované zdroje. V takovém případě je tato možnost uživatelsky přívětivější. Přibyla také karta pro zobrazení aktuálního počtu běžících procesů a také možnost zobrazit si historii stavů 2 hodiny nazpět. Tato stránka je ovšem stále ještě ve výrobě, nebudu jí tedy věnovat více prostoru.

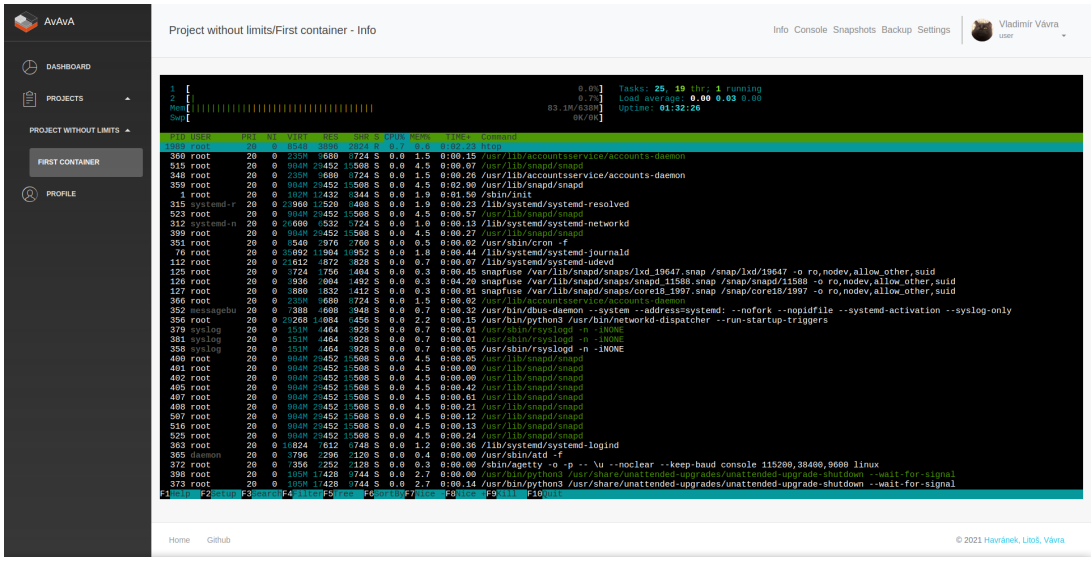
### 2.5.7 Terminál

Jako terminálový emulátor byla využita knihovna xterm a při integraci do aplikace byly využity části kódu z githubu[4].

Pro vytvoření spojení s terminálem kontejneru se pošle API request na backend, který vrátí data potřebná pro vytvoření spojení s terminálem. Ihned poté se na základě získaných dat vytvoří dva WebSockety. Přes první z nich se posílají terminálové příkazy

[4] *xterm react example.* sengmo. Dostupné z `https://github.com/seongmo/xterm-react-example`. [cit. 2021 14-4]

a získávají výsledky. Druhý WebSocket slouží poté jako kontrolní WebSocket umožňující s manipulací terminálu. Posílají se přes něj příkazy pro změnu rozměrů kontejneru. To zajišťuje, že terminál je plně responzivní.




Obrázek 2.27: Terminál

Spojení s kontejnerem se začne vytvářet okamžitě po vykreslení komponentu (metoda componentWillMount) s konzolí a ukončuje tehdy, když je tento komponent odstraněn ze struktury (metoda compontWillUnmount).

Jelikož jsou pro každý terminál vytvořené oddělené WebSockety, je možné otevřít více terminálů najednou. Dokonce je možné vytvořit i více terminálů do stejného kontejneru.

### 2.5.8 Backup

Backup je momentálně implementován tak, že při kliknutí na link Backup bude uživateli zobrazen odkaz, na který když klikne, bude otevřen nový tab, přes který se soubor stáhne. Záloha se nezačne odesílat automaticky, chvíli totiž trvá, než ji backend vytvoří. Tato prodleva je okolo 20 sekund.

Aktuální způsob vůbec nepoužívá na stažení zálohy API klienta, veškeré stahování zařizuje přímo prohlížeč. Tento systém byl zvolen z důvodu, že http proxy nepropustí odpověď serveru typu stream. Jelikož se problém nepodařilo zatím vyřešit, bylo implementováno toto nouzové řešení.

Jakmile bude však problém vyřešen, je připraven kód pro uživatelsky přívětivé

stahování backupu. Při kliknutí na tlačítko se zobrazí spinner se zprávou Creating backup, který bude následně vystřídán textem, jež bude popisovat proces stahování.

### 2.5.9 Nastavení kontejneru

Při kliknutí na link Settings se zobrazí podobný dialog jako při nastavování projektu. Je možné změnit limity, jméno a pokud uživatel bude chtít, tak i root heslo. To poté nahradí stávající heslo kontejneru. Je tak vyřešená změna hesla pro kontejnery.

# 3. Backend

Kapitola backend se věnuje backednu progamu. V první sekci se zaměří na progamovou implementaci práce s LXD a v druhé popíše databáze, jenž projekt využívá. Poslední sekcí je networking, kde je rozebrána problematika sítí ve spojení s kontejnery.

## 3.1 LXD

Tato kapitola podrobněji vysvětluje samotné naprogramování podoby funkčnosti uvedené v kapitolách architektury LXD. Také jsou zde uvedeny prvky, které poskytují prostor pro snadné rozšíření stávajících funkcí. Veškerá komunikace projekt s LXD probíhá přes REST API. Pro zjednodušení požadavků obsahuje backend soubor lxdRoute.js, který zbytku backendu umožňuje snadno a efektivně přistupovat k údajům LXD. Ve zprostředkování všech těchto užitečných funkcí potřebných pro fungování tohoto projektu stálo ale nemálo překážek. Tato kapitola uvádí úhlavní problémy, jimž bylo čeleno, dále výslednou strukturu, a odděleně také sekci pro komunikaci přes WebSocket.

### 3.1.1 Standard odpovědí z rozhraní s LXD

Na začátek můžeme uvést některé standardy návratových hodnot z lxdRoute.js. Většinou je návratovou hodnotou metod status provedení akce, který je v tomto projektu reprezentován objektem OperationState. Ten obsahuje pouze proměnné status a statusCode. Jeho přední využití u všech metod je uložení dat o výsledném stavu akce/metody. Stav může být jakýkoli z http kódů, které LXD navrátí, pokud vrátí chybovou hlášku, její error_code je uložen do pole statusCode a popis důvodu chyby samotné bude v statusu. Ve zbylých případech, kdy se nevrací OperationState, jako výchozí hodnota při úspěšném provedení akce, se jedná o doplnění dat do předaného objektu, kupříkladu již dříve zmíněné zjištění stavu.

Zvláštní výjimku tvoří metoda execInstance, jež taktéž používá OperationState, ale pole status obsahuje text, vypsaný příkazem spuštěným v daném kontejneru, případně výstup do errorového proudu, pokud byl příkaz vykonán s návratovou hodnotou jinou než 0. StatusCode je potom buď 200, pokud se příkaz vykonal úspěšně, nebo 400,

pokud příkaz skončil s jakýmkoli nenulovým exit statutem. K dispozici je ještě druhá forma vráceného OperationState objektu, tu získáme, pokud výslovně uvedeme, že nechceme obdržet výstup příkazu. Potom dostaneme statusCode s hodnotou představující návratovou hodnotu příkazu, který jsme zadali spustit.

### 3.1.2 Hlavní dorozumívací prvky

Komunikace všeho druhu má obvykle sjednocující formát, jako například výše uvedený systém návratových hodnot; REST API od LXD se tomu příliš neliší, hodí se proto jisté základní konsolidační funkce, které tyto společné prvky komunikace zpracovávají v jednotný a snadno použitelný výstup.

Jelikož LXD poskytuje pouze zabezpečené připojení (HTTPS, WSS), stává se první nutností načtení klíče a certifikátu LXD. Ty jsou uloženy v serverové části pod složkou config, jejíž obsah je znám pouze backendu. Tyto, a pár dalších základních, neměnných dat, obsažených v podrobnostech každého dotazu na LXD, sestavuje metoda mkOpts(). Aby vyplnila všechny běžně potřebné informace k vytvoření requestu, přidává také předanou cestu a metodu (výchozí je GET). Tato metoda je určena pouze k připravení nezbytných základních dat, je tedy součástí každého dotazu na LXD.

Vyšší, již specifikovanější nadstavbu, tvoří metoda mkRequest(), která je určena k maximálnímu zjednodušení vytváření běžných dotazů, aniž by tím výrazně omezila možnosti poskytované nodejs. Základy získává přes mkOpts() a případná JSON data k odeslání dále zpracuje a odešle LXD v requestu. Vrací vždy až kompletní odpověď LXD, už jako čitelný objekt. Tento přístup nelze využít pouze u metod, které pracují s proudy dat, jako je například posílání souborů a stahování nebo nahrávání záloh kontejnerů.

Poslední ústřední metodu tvoří getOperation(), která na základě předaného objektu výsledné odpovědi LXD může vyčkat, dokud nedoběhne jakákoliv akce, na kterou se daná operace vztahuje. Častěji využívanou funkcí této metody je ale její návratový objekt OperationState, vyjadřující stav, či výsledek jakéhokoli LXD objektu, který je funkci předán. Použití tedy vede, mimo vyčkání na dokončení operací neurčité délky, především k odchytávání potenciálně vzniklých chyb, které LXD vrátilo a předat je jako jednoduchý, formátem vždy identický objekt. Dle něj se dále posuzuje, zda ukončit, či pokračovat, aktuální činnost volající metody.

Tyto tři funkce tvoří hlavní rozhraní pro všechny metody v lxdRoute.js, jsou proto naprosto nepostradatelné a značně napomáhají psaní a přehlednosti jakýchkoliv metod komunikujících s LXD.

### 3.1.3 Potíže s limity

LXD ukazuje, že v něm existují všechny základní potřebné limity pro omezení využívání hardwarových zdrojů. Bohužel po dlouhém zkoumání a testování jsme si ověřili, jak moc jsou tyto informace nepravdivé.

Na začátek je důležité uvést, že z původně plánovaného ovládání limitů (času procesoru, použité paměti RAM, využívaného disku a provozu internetu na projektech i kontejnerech LXD) bylo bohužel nezbytné přístup k limitům značně změnit.

V první řadě se limity projektů vůbec nenastavují v LXD, jsou tedy používány pouze k výpočtu limitů kontejnerů, důvodem tohoto rozhodnutí bylo nesmyslné chování LXD, které po nastavení limitů projektu neumožňovalo vytvářet kontejnery. Důvodem prý bylo, že kontejnery musejí mít také nastavené limity, které byly nastaveny u projektu, přestože jsme toto při vytváření kontejnerů dodržovali. Druhým ústupkem z původního plánu bylo upuštění od omezování zabraného místa na disku, v tomto případě měly takto omezené kontejnery náhodné chování při startování – většinou se nespustily a vrátily chybovou hlášku, která byla naprosto zcestná[1].

Startovní limity se nastavují kontejnerům vždy při vytvoření. Změny limitů jsou pak možné jejich záplatováním (voláním requestů s metodou PATCH). Výslednými limity, které zde byly nakonec použitelné, byly tedy pouze u kontejneru, a to procesor, pamět RAM a trafika komunikace na síti (download, upload).

### 3.1.4 Zpřístupnění WebSocket terminálu

Správné websockety dostane klient HTTP dotazem na /api/instances/[id]/console. Ten vrátí buď chybovou zprávu – něco pokazí (přeruší spojení), nebo pokud uživatel nemá oprávnění k přístupu ke kontejneru daného [id] –, nebo objekt s poli terminal a control, která představují ID websocketů, na které se klient může připojit pomocí již

[1] *Failed to change ownership on new container storage.* Canonical. Dostupné z `https://discuss.linuxcontainers.org/t/failed-to-change-ownership-on-new-container-storage/10158/10`. [cit. 2021 14-4]

zmíněné cesty /websockets/terminals/[id websocketu], Socket terminálu přijímá jakýkoli vstup a rovnou jej přesměrovává na LXD, control oproti tomu sice také přeposílá cokoli dostane, ale pro projevení účinku musí každá zpráva odpovídat formátu vyžadovanému LXD (viz obrázek 3.1). Zde si lze povšimnout podivného zápisu čísel jako řetězců, domníváme se, že tento zápis byl zvolen pro umožnění změn šířky dle jiných jednotek, oproti výchozí jednotce – znak. Zda již je něco takového implementováno, nelze bohužel kvůli nedostačující dokumentaci LXD zjistit.

```
{
    "command": "window-reize",
    "args":{
            "width": "80",
            "height": "50,
    }
}
```

Obrázek 3.1: resize příkaz

Websockety je nutné uložit, protože se vytvářejí spolu s voláním na získání jejich id od klienta, aby se na ně mohl poté připojit. Soubor websocket.js ve složce services, který se stará o veškerou práci s websockety mezi LXD a klientem, obsahuje k tomuto účelu mapu, jejíž záznamy jsou uloženy pod stringem id svého websocketu, který reprezentují. Objekt záznamu obsahuje samotný websocket, pod jehož id je uložen, dále proměnnou control, nebo terminal, obsahující id druhého LXD websocketu, se kterým se vážou, a pak také proměnnou s klientským websocketem, který se na jejich lxd websocket napojuje.

Tyto proměnné se využívají ve websocketách klientova terminálu a obou lxd socketů (terminál a kontrola velikosti – control) – na nichž jsou zaregistrovány callbacky na ukončení spojení s websocketem, což ukončí i ostatní –, kde poskytují odkazy na ostatní websockety, aby mohly být při konci relace také zavřeny. Významově hlavní callback je na lxd socketu control, který se zavře a ukončí i zbylé sokety v moment, kdy uživatel zadá exit v hlavním shellu, který představoval terminál jeho kontejneru. Díky tomu není třeba zpracovávat zprávy ani jedné strany, a přesto poznáme, kdy uživatel vyšel z

hlavního procesu. Pokud spojení ukončí klientský socket terminálu, uzavře se nejdřív lxd socket control, jehož ukončení již zajistí ukončení všech ostatních.

## 3.2 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.

### 3.2.1 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 3.2: 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. 3.2) *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 podřadné záznamy 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[2] .

**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 uložena 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.

[2]Maximální hodnota je počet jader*kmitočet procesoru, do LXD se přepočítává na procenta

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 dá 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 Google Auth 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[3], 2 GB ram, 8GB volného místa na disku, a download a upload 800kb/s.

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

### 3.2.2 MongoDB

MongoDB je výhodné pro použití při ukládání objektových struktur. V našem případě je tímto stav kontejneru, ze kterého musíme do databáze uložit velikost na disku a všechna data o síťových spojeních. Používáme pouze jednu databázi – lxd; tabulky by podle databázových pravidel SQL měly být tři (projekty, stavy kontejnerů a napojení stavů kontejnerů na projekty, případně ještě hlubší rozdělení na tabulku s id kontejnerů a id záznamů jejich stavu), ovšem pro zbytečnou komplikovanost, kterou by tento přístup představoval, se vytvářejí tabulky dynamicky a reprezentují stejnojmenné projekty (např. p1, p2). Záznamy jsou potom objekty, jejichž unikátní _id je id kontejneru (c1, c2) a obsahují pole disk – kam se ukládá velikost disku – a pole pro objekt obsahující síťové informace.

Tabulky se vytvářejí a mažou spolu s projekty, stejně tak je tomu i u záznamů jednotlivých kontejnerů, avšak ty vytvoří pouze záznam s _id, data o stavu se zapíšou pouze před každým vypnutím kontejneru, aby byla nejaktuálnější a zároveň obnovena pouze tehdy, když jsou potřeba – pro doplnění scházejících dat z volání LXD při vypnutém kontejneru.

## 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 vlastní ssl certifikát. Ten stačí jeden pro celý systém, musí se však jednat o certifikát typu wildcard[4]. 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 hošťovací počítač 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 IP adresou. Z packetu tak nejde zjistit na jakou doménu se uživatel snaží dostat a tedy jakému kontejneru mý být protokol

[4]Certifikát, který zahrnuje i subdomény. Žáci si tak nemusí pořizovat vlastní.

přiřazen. To by se týkalo i ssh, příkaz ssh však podporuje nastavit pomocí flagu -o ProxyCommand (viz. 3.3), 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 3.3: Příkaz k připojení na server[5]

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.

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[6]. 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.

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

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

# Závěr

Ačkoliv jsme původně měli vyšší nároky na náš projekt, které se kvůli špatnému vyhodnocení obtížnosti způsobené malými předchozími zkušenostmi členů týmu s full stack vývojem a špatným dokumentacím závislostí práce, zatím nepodařilo splnit, hodnotíme práci jako úspěch. Kritické části práce fungují, uživatel si může jednoduše vytvořit kontejner. V něm deploynout svojí aplikaci, a vyzkoušet si tak, jaké to je být adminem na linuxovém serveru. Byl tedy vytvořen funkční systém pro vytváření a správu kontejnerů, který může být po menších úpravách nasazen na školní server.

V budoucnosti bychom rádi dokončili všechny funkce, které jsme původně chtěli implementovat. V aktuálním systému jsou jimi práce se Snapshoty kontejnerů, import záloh, zobrazení historie stavů kontejnerů a projektů a systém na dokoupení uživatelských limitů. Je také nutné otestovat, při jakých limitech je možné vytvářet dobré podmínky pro studentské aplikace, aby se maximalizovala efektivita rozdělování zdrojů.

Systém bychom ovšem chtěli rozšířit ještě víc tím, že bychom přidali administrátorský účet a super-administrátorský účet na správu samotné aplikace. Nakonec máme ještě v plánu vytvořit real-time systém pro kooperaci uživatelů na jednom projektu. S již vylepšenou prací bychom se rádi příští zúčastnili soutěže SOČ.

# Seznam použité literatury

*LXD docs – REST API.* Canonical. Dostupné z https://linuxcontainers.org/lxd/docs/master/rest-api. [cit. 2021 14-4].

*Failed to change ownership on new container storage.* Canonical. Dostupné z https://discuss.linuxcontainers.org/t/failed-to-change-ownership-on-new-container-storage/10158/10. [cit. 2021 14-4].

*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 2021-13-4].

*Dashboard template.* Create-Tim. Dostupné z https://github.com/creativetimofficial/light-bootstrap-dashboard-react. [cit. 2021-13-4].

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

*Display total in center of donut pie chart using google charts?* Kevin Brown. Dostupné z https://stackoverflow.com/questions/32256527/display-total-in-center-of-donut-pie-chart-using-google-charts. [cit. 2021 14-4].

*Materia-UI.* react-table. Dostupné z https://react-table.tanstack.com/docs/examples/material-ui-components. [cit. 2021 14-4].

*xterm react example.* sengmo. Dostupné z https://github.com/seongmo/xterm-react-example. [cit. 2021 14-4].

# Seznam obrázků

1.1 Strukura projektu, vlastní tvorba . . . . . 1
1.2 Autentifikace přes Google Auth, vlastní tvorba . . . . . 9

2.1 Struktura souborů frontendu, vlastní tvorba . . . . . 12
2.2 Struktura souborů frontendu – public, vlastní tvorba . . . . . 13
2.3 Struktura souborů frontendu, vlastní tvorba – src . . . . . 13
2.4 Struktura souborů frontendu – api, vlastní tvorba . . . . . 14
2.5 Struktura souborů frontendu – assets , vlastní tvorba . . . . . 14
2.6 Struktura souborů – components, vlastní tvorba . . . . . 15
2.7 Struktura souborů – view, vlastní tvorba . . . . . 16
2.8 Graf stavu u reduxu, vlastní tvorba, vygenerováno pomocí redux dev tools . . . . . 18
2.9 Struktura souborů frontendu – actions, vlastní tvorba . . . . . 18
2.10 Uživatelské rozhraní na počítači, vlastní tvorba . . . . . 20
2.11 Responsivní interface na mobilu – overview, vlastní tvorba . . . . . 20
2.12 Responsivní interface na mobilu – sidebar, vlastní tvorba . . . . . 20
2.13 Odhlášení uživatele, vlastní tvorba . . . . . 21
2.14 Uživatelské rozhraní na počítači, vlastní tvorba . . . . . 22
2.15 Detail grafu, vlastní tvorba . . . . . 23
2.16 Přehled projektů, vlastní tvorba . . . . . 24
2.17 Pohled na ram, vlastní tvorba . . . . . 24
2.18 Výběr projektu, vlastní tvorba . . . . . 25
2.19 Smazání projektu, vlastní tvorba . . . . . 25
2.20 Vytváření projekt, vlastní tvorba . . . . . 26
2.21 Informace o projektu, vlastní tvorba . . . . . 26
2.22 Informace o kontejneru, vlastní tvorba . . . . . 27
2.23 Informace o kontejneru, vlastní tvorba . . . . . 27
2.24 Toolbar pro kontejner, vlastní tvorba . . . . . 28
2.25 Nastavení projekt, vlastní tvorba . . . . . 28
2.26 Informace o kontejneru, vlastní tvorba . . . . . 29

2.27 Terminál, vlastní tvorba . . . . . 30

3.1 resize příkaz . . . . . 35
3.2 Soubory s programem týkající se mysql databáze, vlastni tvorba . . . . 36
3.3 Příkaz k připojení na server, vlastní tvorba . . . . . 41