VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

SYSTÉM PRO AUTOMATICKOU SPRÁVU SERVERŮ

SYSTEM FOR AUTOMATED SERVER ADMINISTRATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MARTIN PAVELKA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MARTIN KRČMA**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Pavelka Martin, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Systém pro automatickou správu serverů**
System for Automated Server Administration
Kategorie: Informační systémy

Zadání:

1. Seznamte se s nástroji OpenStack, VMware a jinými, určenými pro virtualizaci hardware.
2. Nastudujte moderní principy komunikace prostřednictvím technologie GraphOL.
3. Analyzujte vhodné scénáře pro automatickou manipulaci s virtuálními a dedikovanými servery.
4. Implementujte skripty určené pro správu serverů.
5. Navrhněte a implementujte procesy pro bezobslužný běh celého systému.
6. Vytvořte uživatelské rozhraní umožňující správu uživatelských dat a serverů.
7. Napojte existující informační systém na API správy serverů a GraphOL backend poskytující uživatelská data.
8. Zhodnoťte vytvořenou realizaci a navrhněte další rozvoj projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Krčma Martin, Ing.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 26. října 2018

Abstrakt

Cieľom tejto diplomovej práce je návrh užívateľského rozhrania a implementácia informačného systému ako webovej aplikácie. Systém prostredníctvom implementovanej knižnice komunikuje so serverom GraphQL, ktorý spracováva požiadavky nad užívateľskými dátami klienta. Práca popisuje možnosti riešenia automatizácie u fyzických serverov. Aplikácia sprístupňuje cez vlastné aplikačné rozhranie automatickú správu virtuálnych serverov. Všetky operácie sú vykonávané bez zásahu ľudského prvku. Prepojenie s virtualizačnými technológiami je zabezpečené cez poskytované webové rozhrania alebo využitím vlastných skriptov spúšťaných na virtuálnom terminály systému. Nad projektom je postavený systém monitorovania prevádzky jednotlivých komponentov, riešenie integrácie systému využitím nástroja Gitlab a spracovávanie konfiguračných požiadaviek použitím Unix CRON úloh.

Abstract

The goal of this diploma thesis is to design the user interface and implement the information system as a web application. Using the custom implemented library the system communicates with GraphQL server which manages the client data. The thesis describes possible solutions for physical servers automatization. The application provides the application interface to manage virtual servers. Automatization is possible without human interaction. Connection to the virtualization technologies is handled by web interface APIs or custom scripts running in the virtual system terminal. There is a monitoring system built over project components. The thesis also describes the continuous integration using Gitlab tools. Running the configuration task is solved using the Unix CRON system.

Kľúčové slová

VMware, OpenStack, Proxmox, KVM, LXC, cloudové služby, virtuálne privátne servery, cloud computing, fyzické servery, automatická správa serverov, informačný systém, asynchrónne spracovávanie, užívateľské rozhranie, virtualizácia, hypervisor, dependency injection, GraphQL, WebPack, TypeScript, Nette, PHP, NodeJS, Doctrine, Redis, Babel, systém monitorovania, Grafana, systém integrácie a nasadzovania, Gitlab, Cachet

Keywords

VMware, OpenStack, Proxmox, KVM, LXC, cloud services, virtual private servers, cloud computing, physics servers, automatic server manipulation, information system, asynchronous requests, user interface, virtualization, hypervisor, dependency injection, GraphQL, WebPack, TypeScript, Nette, PHP, NodeJS, Doctrine, Redis, Babel, monitoring system, Grafana, continuous integration and development, Gitlab, Cachet

Citácia

PAVELKA, Martin. *Systém pro automatickou správu serverů.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Krčma

Systém pro automatickou správu serverů

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Martina Krčmu. Uviedol som všetky literárne pramene, z ktorých som čerpal.

....
Martin Pavelka
21. mája 2019

Poďakovanie

Rád by som poďakoval pánovi Ing. Martinovi Krčmovi, za vedenie tejto diplomovej práce, Zdeňkovi Vodákovi za pomoc pri návrhu užívateľského rozhrania, Štěpánovi Blažekovi za technické konzultácie v oblasti virtualizačných technológií, Martinovi Urbanczýkovi za nápady využité pri implementácii aplikácie a firme Master, ktorá mi umožnila zrealizovať tento projekt. Tiež by som chcel poďakovať mojej rodine za podporu a pomoc pri písaní práce.

Obsah

1 Úvod **5**

2 Prieskum konkurencie **7**
2.1 Digital Ocean 7
2.2 Rackspace 8

3 Teoretické východiská **9**
3.1 Jazyk PHP 9
3.2 Framework Nette 10
3.2.1 Vývoj a architektúra 10
3.2.2 Životný cyklus presentera 10
3.2.3 Dependency injection 11
3.2.4 Kompatibilita a vývojový cyklus 12
3.3 Preprocesor SASS 12
3.4 Preprocesor TypeScript 13
3.5 Balíčkovacie nástroje 13
3.6 Webpack 14
3.7 Redis 14
3.8 Doctrine ORM 15
3.9 HighCharts 16
3.10 Babel 16
3.11 GraphQL API 17

4 Virtualizácia **19**
4.1 Architektúra virtualizačných technológii 19
4.2 Pojem cloud computing 20
4.3 Výhody použitia virtuálnych serverov 21
4.4 Virtualizácia technológiou OpenStack 21
4.4.1 Architektúra clusteru 21
4.4.2 Zoznam entít systému 23
4.5 Virtualizácia technológiou VMware 24
4.5.1 Architektúra clusteru 24
4.5.2 Zoznam entít systému 25
4.6 Virtuálne privátne servery 26
4.6.1 Proxmox VE 26

5 Technologické možnosti **28**
5.1 Návrh automatizácie systému OpenStack 28

5.1.1 Autentifikácia požiadaviek 28
5.1.2 Vytvorenie používateľa 29
5.1.3 Vytvorenie projektu 29
5.1.4 Vytvorenie siete 30
5.1.5 Vytvorenie inštancie 30
5.2 Návrh automatizácie systému VMware 31
5.2.1 Autentifikácia požiadaviek 31
5.2.2 Vytvorenie organizácie 32
5.2.3 Vytvorenie klienta 32
5.2.4 Vytvorenie virtuálneho datacentra 33
5.2.5 Vytvorenie siete VDC 33
5.2.6 Vytvorenie virtuálnej aplikácie 34
5.2.7 Vytvorenie virtuálneho servera 34
5.3 Návrh automatizácie systému Proxmox VE 35
5.3.1 Autentifikácia požiadaviek 35
5.3.2 Vytvorenie LXC 35
5.3.3 Vytvorenie KVM 36
5.4 Fyzické servery 37
5.4.1 Automatizácia predaja fyzických serverov 37

6 GraphQL SDK 38
6.1 Architektúra knižnice 38
6.2 Použitie SDK vo webovej aplikácii 39
6.3 Implementácia tried repozitárov 40
6.4 Generovanie požiadaviek 40
6.5 Zasielanie žiadostí na server 41
6.6 Spracovanie odpovede servera 42

7 Webová aplikácia 43
7.1 Špecifikácia požiadaviek 43
7.2 Návrh užívateľského rozhrania 44
7.2.1 Oneskorené načítanie dát 44
7.2.2 Responzívne zobrazenie 44
7.2.3 Zobrazenie grafov 45
7.2.4 Dashboard 46
7.3 Objektová architektúra 47
7.3.1 Vrstva presenterov 47
7.3.2 Implementácia modelov 48
7.4 Implementácia asynchrónnosti 49
7.4.1 Modul responzívnosti 49
7.5 Implementácia designu aplikácie 50
7.6 Správa dočasného úložiska 51
7.7 Kompatibilita TypeScript modulu 51
7.8 Automatické testovanie GraphQL API 51
7.9 Systém priebežnej integrácie 52

8 Automatická správa serverov 53
8.1 Konfigurácia prostredia 53

8.2 Objektovo relačný model . . . 53
8.3 Modul automatizácie systému OpenStack . . . 54
8.3.1 Definícia DTO entít . . . 54
8.3.2 Implementácia aplikačnej logiky . . . 55
8.3.3 Implementácia kontroléra systému . . . 55
8.4 Modul automatizácie systému VMware . . . 56
8.4.1 Definícia DTO entít . . . 56
8.4.2 Implementácia aplikačnej logiky . . . 56
8.4.3 Implementácia kontroléra systému . . . 57
8.5 Modul automatizácie systému Proxmox VE . . . 57
8.5.1 Definícia DTO entít . . . 57
8.5.2 Implementácia aplikačnej logiky . . . 58
8.5.3 Implementácia kontrolérov systému . . . 58
8.6 Automatické testovanie systému . . . 58
8.7 Rozhranie automatickej správy serverov . . . 59
8.7.1 Implementácia rozhrania . . . 59
8.7.2 Komunikačný protokol API . . . 60
8.8 Automatizácia procesov systému . . . 61

9 Testovanie systému 62
9.1 Spôsob testovania . . . 62
9.1.1 Segmentácia testovacích užívateľov . . . 62
9.2 Testovanie prototypu . . . 63
9.2.1 Validácia služieb . . . 64
9.2.2 Overenie štatistík . . . 64
9.2.3 Analyzovanie grafov . . . 64
9.2.4 Testovanie dočasného úložiska . . . 64
9.2.5 Automatizácia tvorby serverov . . . 64
9.2.6 Podnety z testovania systému . . . 65
9.3 Testovanie výkonnosti . . . 65
9.3.1 Optimalizácia rýchlosti aplikácie . . . 65
9.3.2 Rýchlosť spracovávania požiadavkov cez API . . . 66
9.3.3 Časová náročnosť tvorby serverov . . . 67
9.4 Verifikácia vydanej aplikácie . . . 67
9.4.1 Spätná väzba od zákazníka . . . 67
9.4.2 Hlásenie chýb aplikácie . . . 68
9.4.3 Monitorovanie servera . . . 68
9.4.4 Podpora pri výpadkoch a údržbe . . . 69
9.5 Metriky kódu . . . 70

10 Záver 71

Literatúra 73

A Konkurenčné systémy 75

B Ukážky webovej aplikácie 77

C OOP vizualizácia automatizácie 84

D Zoznam služieb systému **90**

E Príklady volaní API **92**

F Expirácia Redis záznamov **94**

G Inštalácia a konfigurácia systému **95**
G.1 Závislosti 95
G.2 Inštalácia 95

H Obsah pamäťového média **96**

Kapitola 1

Úvod

Pojem *cloud computing* a *virtualizácia* sa stávajú veľmi populárnymi. Pre menších poskytovateľov služieb prestáva byť výhodné vlastniť fyzické servery. Ich nákup môže byť veľmi nákladný a prevádzka vyžaduje záruku stáleho internetového pripojenia a napájania. V prípade výpadku dochádza k nedostupnosti poskytovaných služieb. Na ochranu pred týmto druhom rizík je potrebné vybudovať záložné napájanie použitím generátora elektrickej energie a prepojenie internetového pripojenia cez viacero poskytovateľov a záložných vetiev. Výhodnejším riešením sa ukazuje využitie prenájmu výpočtových zdrojov poskytovaných dátovými centrami po celom svete. Tie častokrát okrem nižších nákladov na prevádzku servera, poskytujú vysokú mieru dostupnosti služieb, ktorých zdroje sú často redundantné pre zníženie množstva výpadkov. Okrem veľkých hráčov na trhu ako je Amazon AWS, Google, Apple alebo Microsoft, je v niektorých prípadoch výhodnejšia voľba lokálneho poskytovateľa s lepšou podporou a možnosťou výhodnejšej ponuky v prípade väčšieho odberu služieb.

Moderným trendom je možnosť automatického procesu nákupu a konfigurácie virtualizovaných služieb, ktoré poskytovateľ poskytuje. Napriek existencii takýchto systémov je veľmi obtiažne nájsť komplexnú prácu, ktorá by popisovala možnosti napojenia virtualizačných nástrojov na informačný systém. Práca sa zaoberá najrozšírenejšou technológiou virtualizácie zdrojov od spoločnosti VMware. Skúmaná je rovnako robustná architektúra virtualizácie OpenStack. Na rozdiel od produktu VMware ide o open-source alternatívu. Medzi menej komplexnú technológiu patrí virtualizačný nástroj projektu Proxmox VE. Krátka časť je venovaná téme napojenia fyzických serverov na automatický predaj.

Kapitola 2 popisuje podobné projekty, ktoré umožňujú automatickú správu serverov. Jedná sa o služby Rackspace a Digital Ocean, ktoré sú jednými z najväčších na trhu. Práca sleduje ich rôzne vlastnosti, ktoré by bolo možné využiť pri návrhu užívateľského rozhrania alebo implementácie podobnej webovej aplikácie.

Popis teoretických východísk, ktoré sú potrebné pri implementácii práce je možné nájsť v kapitole 3. Zameriava sa na jazyk PHP a systém Nette, ktorý bol využitý pri vývoji webovej aplikácie. Podrobne je rozobraný spôsob jeho fungovania a analyzované sú návrhové vzory, ktoré využíva. Kapitola popisuje možnosti implementácie funkčnosti cez jazyk JavaScript a jeho kompilátor TypeScript. Skúmané sú možnosti kombinovania aplikácie napísanej v jazyku PHP a TypeScript modulu prostredníctvom knižnice Webpack. Záver kapitoly sa venuje komunikácii s rozhraním servera využívajúceho technológiu GraphQL. Podrobne je zobrazený formát požiadavkov a odpovedí zasielaných aplikáciou a serverom.

V kapitole 4 a 5 sa nachádza hlavná časť, ktorá sa zaoberá virtualizáciou. Pojednáva o možnostiach tvorby automatických skriptov na základe skôr zistených poznatkov o virtualizačných technológiách. Definuje možnosti komunikácie s virtualizačnými nástrojmi ako

aj možnosti vytvorenia nových virtuálnych inštancií. Na základe týchto skriptov je možná implementácia systému. Krátka sekcia sa zaoberá problematikou fyzických serverov a analyzuje možnosti ich automatizácie prostredníctvom ich interných mechanizmov.

Implementácia riešenia začína návrhom a implementáciou knižnice pre komunikáciu so serverom **GraphQL** v kapitole 6. Popísaná je architektúra implementovaného riešenia, definovaný je protokol komunikácie a spôsob implementácie tried využívajúcich túto knižnicu. Záver kapitoly popisuje možnosti ukladania komunikácie a hlásenia chýb.

Kapitola 7 popisuje implementovanie **webovej aplikácie**. Začiatok špecifikuje požiadavky, ktoré by mala výstupná implementácia pokrývať. Popisovaný je návrh užívateľského rozhrania a jeho prvkov ako aj možnosť grafického zobrazenia vykonávania asynchrónnych operácií. Druhá časť sa zaoberá samotnou implementáciou v jazyku PHP a JavaScript a zobrazuje architektúru implementovaného riešenia. Záver popisuje spôsob automatickej integrácie aplikácie cez nástroj Gitlab.

Kapitola 8 sa zaoberá **systémom automatickej správy serverov** a jeho implementáciou pomocou rozhraní virtualizačných technológií. Začiatok sa venuje možnosti konfigurácie systému a jeho architektúrou. Ďalšie časti podrobnejšie popisujú návrh architektúry a jej implementáciu pre jednotlivé virtualizačné technológie. Záver sa zaoberá testovaním systému, celkovou automatizáciou využitím CRON úloh a implementáciou aplikačného rozhrania pre zasielanie požiadaviek na tvorbu serverov z iných systémov.

Testovanie výsledného produktu je definované v kapitole 9. Popisované je užívateľské testovanie aplikácie. Návrhy na zlepšenie aplikácie, ktoré boli výsledkom testovania sú popísané v sekcii 9.2.6. Testovaná bola rýchlosť reakcií aplikácie, čas tvorby serverov a rýchlosť spracovávania požiadavkov aplikačného rozhrania pri komunikácii zo vzdialeného servera. Skúmané boli riziká a možnosti verifikácie riešenia. Koniec kapitoly popisuje metriky kódu.

Záver krátko pojednáva o výsledkoch práce ako celku a návrhoch na jej rozšírenie do budúcnosti. Po prečítaní práce by mal byť čitateľ schopný nadobudnúť predstavu a inšpiráciu pri implementácii moderného webového systému spolu s prepojením na virtualizačné technológie. Prílohy bližšie zobrazujú a vysvetľujú rôzne časti práce.

Kapitola 2

Prieskum konkurencie

Pred návrhom a implementáciou projektu je potrebné získať čo najviac informácií o podobných riešeniach. Potrebné je posudzovať tieto riešenia z hľadiska dostupných use-case riešení ako aj po grafickej stránke. Je jednoduchšie nájsť existujúce riešenia problémov ako sa ich snažiť implementovať vlastným spôsobom, ktorý častokrát nemusí byť tým najlepším. Pri prieskume trhu som sa zameral na dvoch väčších hráčov na trhu a porovnal som nimi implementované systémy, ponúkajúce správu užívateľských dát a automatizácie. Ukážky konkurenčných systémov je možné vidieť v prílohe A.

2.1 Digital Ocean

Služba Digital Ocean[1] ponúka systém založený na takzvaných *Droplets*. Predstavujú abstraktné inštancie serverov, ktoré si je možné do systému pridávať a vykonávať nad nimi rôzne operácie. Pri tvorbe nového Dropletu je užívateľ postupne smerovaný procesom konfigurácie nového servera. Takéto riešenie procesu tvorby servera je veľmi pekné a do budúcna bude implementácia podobného scenára zahrnutá do môjho projektu.

Nad každým Dropletom je možné vykonávať rôzne operácie, ekvivalentné s tými, ktoré sú dostupné prostredníctvom rozhraní virtualizačných nástrojov. Pre zavolanie požadovanej operácie nad serverom je potrebné prejsť do menu detailu servera. Vo vrchnej časti stránky sa zobrazujú aktuálne informácie o inštancii. Pod nimi je zobrazená lišta, ktorá prostredníctvom ikon a popisov ponúka možnosti správy servera. Po zvolení akcie sa zobrazí jej detail, kde sa akcia bližšie špecifikuje a stlačením tlačidla vykoná. V spodnej časti stránky je zobrazená tabuľka s históriou akcií, ktoré boli vykonané nad inštanciou.

Systém implementuje zobrazenie grafov. Grafy sú zobrazené klasicky reprezentáciou hodnôt na dve kolmé osy X a Y, kde os X obsahuje časové hodnoty a os Y zobrazuje hodnotu parametrov. Grafy sú veľmi jednoduché a nie je možné nad nimi vykonávať rozšírenejšie funkcie. Možné je zvoliť časové okno pre zobrazenie hodnôt konkrétneho grafu.

Služba Digital Ocean je asi jednou z najlepšie riešených webových aplikácií, na ktoré som pri prieskume narazil. Jej užívateľské rozhranie je jednoduché a veľmi intuitívne. Užívateľ je vedený jednotlivými use-case, vždy tak aby nemusel venovať pozornosť príliš veľkému počtu prvkov na stránke. Grafy sú síce jednoduché, ale veľmi prehľadné. Tvorba servera je jednoduchá a operácie, ktoré je možné nad ním vykonávať, je možné konfigurovať bez väčšej obtiažnosti a znalostí koncového užívateľa aplikácie.

[1] Aplikácia Digital Ocean https://www.digitalocean.com

2.2 Rackspace

Služba Rackspace[2] nemá oproti službe Digital Ocean príliš prepracovaný design a užívateľské rozhranie. Napriek tomu disponuje prepracovanými riešeniami, ktoré by bolo možné využiť pri implementácii podobnej webovej aplikácie.

Po prihlásení do systému je užívateľ presmerovaný na základný dashboard. Práve touto stránkou služba vyniká. Zobrazené sú všetky možnosti, ktoré môže užívateľ po prihlásení do systému potrebovať. Informácie sú členené do jednotlivých blokov. Každý blok organizuje položky daného typu. Zobrazuje sa zoznam tiketov, ktoré užívateľ aktuálne rieši s podporou. V prípade problému so samotnými servermi zobrazuje blok monitorovania aktuálne problémy, ktoré boli nájdené pri diagnostike. Ďalší blok zobrazuje aktuality a správy zaslané z marketingového oddelenia firmy. Posledný blok obsahuje rýchle odkazy, ktoré je možné upravovať užívateľom prihláseným do systému. Celý dashboard je veľmi prehľadný a je možné sa v ňom orientovať bez väčších problémov .

Zobrazenie serverov je formou tabuliek. Pri každom serveri je zobrazený jeho názov, priradená IP adresa a aktuálny stav diagnostiky monitorovacieho systému. Pomocou tlačidiel nad zoznamom serverov je možné vytvoriť nový server. Detail servera nedisponuje príliš užívateľsky prívetivou grafikou a obsahuje niekoľko záložiek podľa operácie, ktorú sprístupňuje. Po kliknutí na požadovanú záložku je možné akciu ďalej konfigurovať. Detail serveru je celkovo pomerne komplikovaný a designom nedostatočný pre využitie v modernom systéme.

[2] Aplikácia Rackspace https://www.rackspace.com

Kapitola 3

Teoretické východiská

Súčasťou zadania práce je vytvorenie webovej aplikácie, ktorá bude poskytovať rozhranie na manipuláciu s užívateľskými dátami s možnosťou tvorby požiadaviek na server, ktorý bude automatizovať úlohy spojené s administráciou serverov. Aplikácia by mala ponúkať jednoduché, avšak plnohodnotné riešenie takéhoto informačného systému. Webový vývoj napreduje značne rýchlo. Pri výbere technológií by sa mala potreba dlhodobej údržby systému stretať so záujmom využitia moderných technológií. Kapitola sa zaoberá možnosťami implementácie dynamického informačného systému, tvorbou aplikačných rozhraní a komunikáciou využitím modernej technológie GraphQL.

3.1 Jazyk PHP

Jedným z najpoužívanejších programovacích jazykov na tvorbu webových aplikácií je jazyk PHP[1]. Projekt je distribuovaný ako open-source[2] a vznikol už v roku 1994. Výhodou jazyka je pomerne jednoduchá syntax kódu. Využíva princípu **server-side**[3] spracovávania požiadaviek, pri ktorom je server zodpovedný za vytvorenie výsledného zobrazenia, ktoré zasiela klientovi. PHP poskytuje možnosť tvorby skriptov slúžiacich pre príkazový riadok. Kladom ktorý treba spomenúť, je vysoká **flexibilita**. Skripty je možné spúšťať skoro na každej platforme a výstupom odpovede servera nemusí byť vždy len HTML dokument, ale napríklad aj súbor vo formáte PDF. Samotný jazyk v svojom jadre ponúka množstvo funkcií zameraných práve na tvorbu webových aplikácií a je ho možné rozšíriť o ďalšie funkcie inštalovaním nových **rozšírení**, medzi ktoré patrí napríklad podpora pre systém ukladania dát Redis popísaný v kapitole 3.7. [12]

Dôležitou zmenou je verzia 7 s podporou **dynamickej typovej kontroly**[4] a dátových typov u parametrov funkcií a ich návratových hodnôt. Táto vlastnosť umožňuje tvorbu omnoho čitateľnejšieho kódu, ktorý sa stáva lepšie udržiavateľný. V porovnaní výkonnosti oproti predchádzajúcim verziám, dosahuje podľa niektorých testov až dvojnásobok rýchlosti spracovávania požiadaviek zasielaných na webový server [4]. V budúcich verziách je plánované pridanie podpory pre dátové typy premenných, ktoré sú definované v triedach. Technológia *Just-in-time* funguje na princípe priameho prekladu zdrojových súborov do strojového kódu procesora. Výsledkom by malo byť zrýchlenie funkcií náročných na procesorový výkon. Ich vykonávanie nebude naďalej interpretované virtualizáčným jadrom PHP.

[1]PHP – Hypertext Preprocesor, oficiálna stránka http://php.net

[2]Stránka iniciatívy Opensource https://opensource.org

[3]Server-side viď. https://www.codeconquest.com/website/client-side-vs-server-side/

[4]Dynamická typová kontrola https://www.root.cz/clanky/staticka-dynamicka-typova-kontrola/

3.2 Framework Nette

Frameworky sú knižnice[5], ktoré uľahčujú prácu pri programovaní aplikácií a poskytujú podporu pri písaní správneho kódu. Použitím získavame kratší kód, ktorý spravidla neobsahuje také veľké množstvo chýb, ako v prípade vlastnej implementácie funkcií poskytovaných frameworkom. Nette je napísaný v programovacom jazyku PHP a využíva väčšiny možností spojených s moderným OOP programovaním. Jeho použitie je voľné a podpísané licenciou BSD[6]. Projekt pochádza z Českej republiky, v ktorej je jedným z najpoužívanejších frameworkov napísaných v tomto jazyku. Podľa testov patrí medzi tie najrýchlejšie frameworky, ktoré sú napísané v jazyku PHP [2]. Vývoj zabezpečuje rozsiahla komunita ľudí. Aktuálne bola vydaná nová verzia, na ktorú bude možné aplikáciu v budúcnosti inovovať.

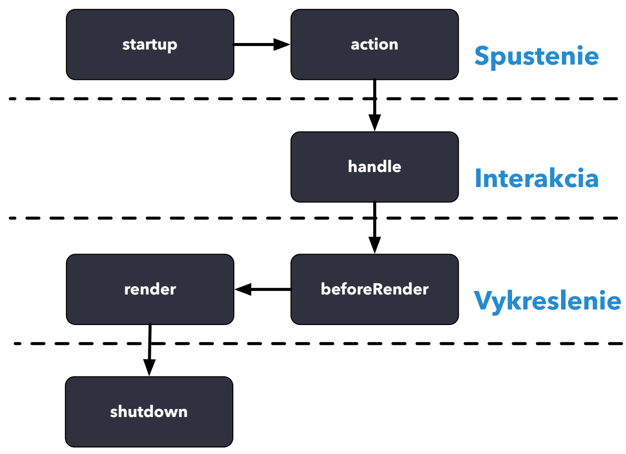
3.2.1 Vývoj a architektúra

Na ladenie kódu je dostupný nástroj *Tracy*[7], ktorý zobrazuje chyby spolu so všetkými dostupnými informáciami, ktoré by mohol programátor potrebovať pri riešení daného problému. Pri návrhu architektúry aplikácie je využívaný návrhový vzor **MVP**, ktorý vychádza z architektúry MVC[8]. MVP rozdeľuje celú aplikáciu na menšie logické celky, ktoré medzi sebou navzájom komunikujú. **Model** aplikácie je zložený z množiny služieb. Služby implementujú aplikačnú logiku ako napríklad komunikáciu s API[9]. **Presenter** sa stará o získanie dát z modelu, pričom tieto dáta poskytuje poslednej vrstve view. **View** tieto dáta prijíma a ďalej ich zobrazuje užívateľovi pomocou predpísaných šablón. Framework umožňuje písanie *unit testov*[10]. Skript na ich spúšťanie je schopný vytvoriť štatistiku pokrytia kódu testami spolu s vyznačenými nepokrytými časťami. Prienik s jazykom **JavaScript** popísaným v kapitole 3.4 a technológiou *Ajax*[11] poskytuje rozhranie takzvaných *snippetov*. Tieto prvky umožňujú namiesto presmerovania alebo obnovenia aktuálnej stránky, obnoviť len jej určitú časť. Komunikácia prebieha prostredníctvom serializácie vo formáte *JSON*[12]. Použitím knižnice umožníme aj iným prvkom stránky aby fungovali **asynchrónne** bez nutnosti obnovenia. Dôležitým prvkom aplikácie je ukladanie stavu do session. Nette umožňuje uloženie relácie prostredníctvom interného úložiska alebo použitím iného nástroja ako *Redis*, ktorý je popísaný v kapitole 3.7. [7]

3.2.2 Životný cyklus presentera

Po vytvorení presentera sa zavolá metóda **startup**. Môže sa využiť na inicializáciu premenných alebo na overenie či je užívateľ autorizovaný. **Action** metóda funguje podobne ako *render*. Na rozdiel od nej nič nevykresľuje, ale len vykonáva. Je však dôležité vedieť, že metóda *action* sa vždy vykoná pred *render*. **Handle** vykonáva buď spracovávanie formulárov alebo *Ajax* požiadaviek. Táto metóda tiež nič nevykresľuje, ale môže vyvolať požiadavku na prekreslenie stránky. **BeforeRender** je metóda ktorá sa volá tesne pred funkciou *render*. Slúži napríklad na nastavenie hodnôt pre šablónu. **Render** je samotné vykreslenie šablóny a jej naplnenie dátami. **Shutdown** sa volá pri ukončení životného cyklu presentera. [3]

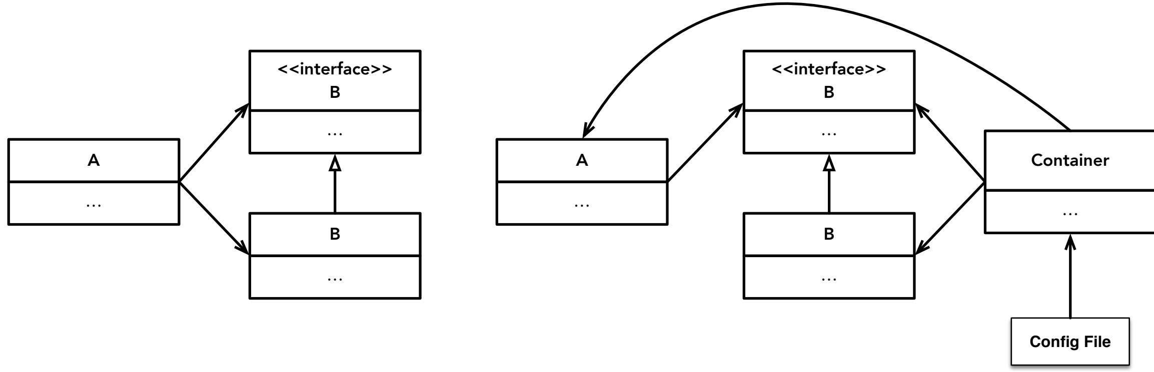
[5] Termín knižnica viď. https://www.webopedia.com/TERM/L/library.html
[6] The 3-Clause BSD License https://opensource.org/licenses/BSD-3-Clause
[7] Tracy viď. viac https://tracy.nette.org/cs/
[8] MVC špecifikácia http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf
[9] API viď. https://medium.freecodecamp.org/what-is-an-api-in-english-please-b880a3214a82
[10] Unit testovanie https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters
[11] Ajax viď. https://www.w3schools.com/xml/ajax_intro.asp
[12] JSON viď. https://www.copterlabs.com/json-what-it-is-how-it-works-how-to-use-it/



Obr. 3.1: Životný cyklus presentera vo frameworku Nette

3.2.3 Dependency injection

Jednou z najdôležitejších požiadaviek pri tvorbe architektúry je **dlhá životnosť**. Dodržiavanie tohto bodu môže viesť k zníženiu časovej náročnosti projektu a úspore peňazí pri jeho vývoji. Štúdie poukazujú, že návrhový vzor **depencency injection** dokáže výrazne zredukovať závislosti jednotlivých modulov. Tým sa stáva kód lepšie prehľadný a modifikovateľný. Technika umožňuje programátorovi vkladať odkazy na objekty do tried použitím *kontajnera*, ktorý konfigurujeme externe a ktorý ich aj vytvára. Príklad nižšie zobrazuje **triedu A** so závislosťou na **triede B**, ktorá implementuje **rozhranie B**. Pridaním *kontajnera* odstránime závislosť na triede B. V tomto prípade sa o závislosť postará práve *kontajner*. Celú situáciu zobrazuje obrázok 3.2. [18]



Obr. 3.2: Príklad použitia dependency injection [18]

Nette využíva na implementáciu *dependency injection* balíček **Nette DI**[13], ktorý obsahuje implementáciu kontajneru. Následne je možné získavať závislosti dvoma spôsobmi. Pomocou použitia notácie **inject** v komentári nad premennou, do ktorej sa objekt vkladá alebo predaním potrebných závislostí **konštruktoru**[14] presentera alebo služby.

[13] Repozitár Nette DI https://github.com/nette/di
[14] Konštruktor - PHP dokumentácia http://php.net/manual/en/language.oop5.decon.php

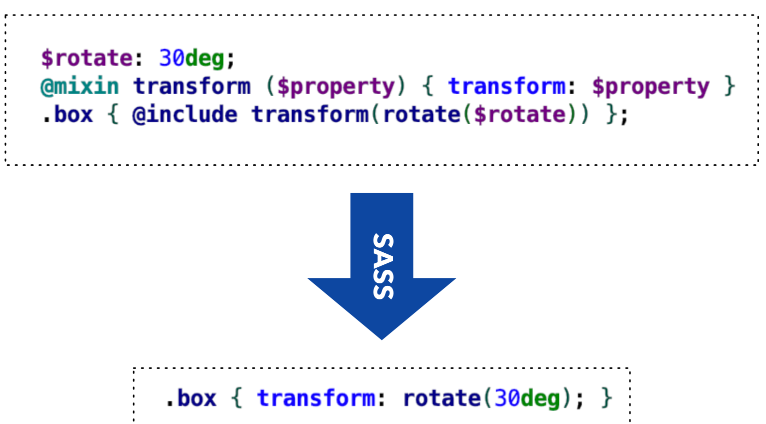
3.2.4 Kompatibilita a vývojový cyklus

Výhodou Nette je veľmi dobrá **spätná kompatibilita**. V prípade vydania novej verzie nie je potrebné väčšie prepisovanie kódu a všetky metódy, ktoré budú v budúcnosti odstránené sú označené ako *deprecated*. Framework je rozdelený do menších balíčkov, ktoré majú vlastný **vývojový proces**, čo zlepšuje aktualizácie. Nie je potrebné aktualizovať framework ako celok, ale len jednotlivé balíčky. V prípade potreby rozšírenejších funkcionalít je možné pridať rôzne rozšírenia pomocou balíčkovacích nástrojov popísaných v kapitole 3.5. Tie sú registrované v konfiguračnom súbore a môžeme ich používať ako služby cez *kontajner*.

3.3 Preprocesor SASS

Kaskádové štýly CSS[15] sú štandardom pre štylizovanie štruktúrovaných dokumentov. Podľa štúdií ich využíva až 90% webových programátorov každý deň. Syntax jazyka je veľmi jednoduchá, avšak neobsahuje žiadne z dnešných moderných programovacích prvkov. Na zvýšenie flexibility a abstrakcie obyčajného CSS sa dnes vo veľkej miere využívajú **preprocesory** *Less*, *Stylus* alebo **SASS**[16]. Ponúkajú **rozšírený zápis**, ktorý dopĺňa tieto chýbajúce štruktúry. Zo zápisu štýlu pomocou vlastnej syntaxe vygenerujú čisté CSS, čím zaručujú plnú **kompatibilitu**. Jednou z hlavných vlastností je možnosť používať **premenné**. Vďaka ním môžeme vykonať istú mieru **abstrakcie** nad celým designom stránky. Tieto premenné nemajú typy, ale umožňujú definovať rozsah ich platnosti. Ďalšou významnou vlastnosťou je používanie **hierarchických štruktúr**. Triedy objektov môžeme skladať od menej špecifickejšej po špecifickejšiu napríklad pomocou použitia **kombinátorov**. Poslednou spomínanou výhodou je používanie blokov nazývaných aj **mixins**[17]. Tie umožňujú vyňať určitú časť štýlu a substituovať ju. Napodobňujú tak správanie funkcií v bežných programovacích jazykoch. Tieto bloky môžu obsahovať aj vstupné parametre. [14]

Obrázok 3.3 popisuje zápis funkcie, ktorá vykoná otočenie objektu. Definovaná je premenná *rotate*, ktorú je možné použiť znovu v nasledujúcich častiach kódu. *Mixin* definuje funkciu, ktorá vykoná otočenie v závislosti od parametra popisujúceho uhol. Kľúčovým slovom *@include*, je možné vygenerovať vlastnosť CSS v prvku s triedou *box*. Blok kódu pod šípkou zobrazuje vygenerovaný štandardný kód kaskádových štýlov.



Obr. 3.3: Príklad kompilácie SASS kódu rotovania prvku

[15] CSS definícia https://www.jakpsatweb.cz/enc/kaskadove-styly.html
[16] Stránka projektu Sass https://sass-lang.com
[17] Mixins príklady https://scotch.io/tutorials/how-to-use-sass-mixins

3.4 Preprocesor TypeScript

Napriek úspechu programovacieho jazyka **JavaScript**[18] je jedným z jeho hlavných problémov zabezpečenie kompatibility. **TypeScript**[19] je rozšírením JavaScriptu, ktoré má riešiť jeho viaceré nedostatky. Ponúka rozšírenú syntax pracujúcu nad štandardom *EcmaScript*[20]. Tento preprocesor pridáva **podporu hierarchickej štruktúry modulov, tried a rozhraní**. Najdôležitejšou vlastnosťou je podpora **dátových typov** a *statickej typovej kontroly*. Ponúka veľmi jednoduché a flexibilné riešenie pre programátora pracujúceho s JavaScript aplikáciami. Podporuje množstvo bežných programovacích praktík, s ktorými je možné sa stretnúť pri písaní klasického JavaScript kódu. Preprocesor po syntaktickej kontrole preloží kód do klasického JavaScript kódu, čím je zaručená **kompatibilita**. Napriek typovej kontrole a dátovým typom, nie je táto kontrola striktná a dá sa obísť, čo môže viesť k typovým problémom po spustení aplikácie. [1]

Všetok kód JavaScriptu je v prípade webových aplikácií vykonávaný v internetovom prehliadači. Táto vlastnosť, ktorou sa odlišuje od PHP, ktorý je *server-side*, umožňuje väčšiu distribúciu výkonu pomedzi všetkých účastníkov, ktorý daný web navštívia. Tento prístup sa potom nazýva **client-side**[21]. Problémom JavaScriptu je **kompatibilita**. Pretože je jeho kód interpretovaný naprieč všetkými druhmi internetových prehliadačov, je prakticky nemožné zaručiť plnú kompatibilitu medzi nimi. Tento problém sa snaží riešiť práve spomínaný štandard *EcmaScript*. Ďalším problémom je, že tento štandard sa veľmi rýchlo mení a jeho nová verzia vychádza skoro každý rok. Tento problém spôsobuje, že starší projekt môže byť značne náročné prepísať do nového štandardu. Na riešenie tohto problému sa používa knižnica **Babel**[22] popísaná v kapitole 3.10 a čiastočne je riešený aj prostredníctvom TypeScript preprocesora, ktorý generuje vždy aktuálny kód a prípadnú kompatibilitu je možné nastaviť v jeho konfiguračnom súbore.

```
class Greeter {
    greeting: string;
    constructor (message: string) {
        this.greeting = message;
    }
}
```

TS

```
const Greeter = /** class */ (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    return Greeter;
}());
```

Obr. 3.4: Príklad kompilácie TypeScript kódu

3.5 Balíčkovacie nástroje

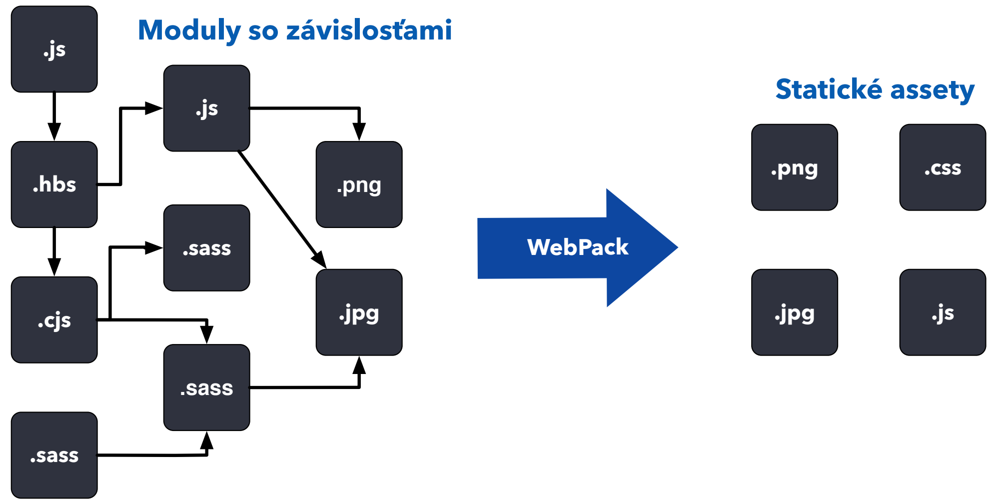
S nárastom veľkosti projektu strácame prehľad nad jeho rôznymi **závislosťami** a ich **verziami**. Bolo by prakticky nemožné, hlavne v prípade JavaScript aplikácií, spravovať tieto závislosti manuálne. Problém vzniká aj v prípade, že závislosti ukladáme priamo do repozitára projektu, kedy sa zhoršuje prehľadnosť zmien kódu a veľkosť repozitára narastá. Nie je možné kombinovať balíčky napísané v jazyku PHP a Javascript. Pre každý jazyk je potrebná voľba samostatného nástroja. V prípade jazyka PHP existuje jediná možnosť a to nástroj **Composer**.[23]. *Composer* umožňuje automatickú správu závislostí a ich verzií. Pomocou

[18] JavaScript návod https://javascript.info
[19] Stránka projektu TypeScript https://www.typescriptlang.org
[20] Organizácia ECMA https://www.ecma-international.org/default.htm
[21] Client-side viď. https://www.upwork.com/hiring/development/how-scripting-languages-work/
[22] Projekt Babel https://babeljs.io
[23] Stránka projektu Composer https://getcomposer.org

konfiguračného súboru je možné definovať jednotlivé balíčky a použitím jedného príkazu ich všetky nainštalovať v požadovanej verzii. U balíčkov napísaných v jazyku JavaScript alebo CSS je na výber z väčšieho počtu nástrojov. Medzi najznámejšie patrí *NPM*[24] alebo novší **Yarn**[25]. Oba nástroje fungujú podobne ako *Composer* a definujú svoje závislosti v konfiguračnom súbore spolu s ich verziami.

3.6 Webpack

Webpack je nástroj pre **JavaScript moduly**. Vytvára graf závislostí, ktoré aplikácia potrebuje. Vo výsledku tieto balíčky zabalí do jedného alebo viacero výstupných súborov. Funkčnosť frameworku popisuje lepšie obrázok 3.5. Pomocou nástroja je možné vytvoriť viacero kompilovaných konfigurácií, čo sa dá využiť pri rozličných **prostrediach** [10]. Webpack sa využíva aj v prípade, že niektoré balíčky nie sú priamo určené pre použitie na webe. Výhodou jeho použitia, je možnosť vygenerovať z množstva menších súborov jeden, ktorý sa následne **komprimuje** pomocou efektívnych algoritmov a výsledný kód je možné na produkcii *obfuskovať*[26], čím zvýšime **bezpečnosť** aplikácie. Webpack podporuje viacero **rozšírení**, ktoré je možné podľa potreby využívať a tak rozširovať jeho funkcionalitu. Jedným z rozšírení je napríklad podpora pre *Babel* kompiláciu popísanú v sekcii 3.10.



Obr. 3.5: Webpack - grafické znázornenie

3.7 Redis

Redis (Remote Dictionary Server)[27] je serverová služba napísaná v *ANSI C* jazyku, ktorá poskytuje **kľúčovú databázu** uloženú priamo v pamäti servera, na ktorom táto služba beží. Server je **neblokujúci** a **asynchrónny** s podporou **master a slave replikácie** dát[28]. Rozdiel medzi Redis serverom a inými servermi založenými na kľúčových databázach je, že

[24]Stránka projektu NPM https://www.npmjs.com
[25]Stránka projektu Yarn https://yarnpkg.com
[26]Práca popisujúca obfuskáciu https://core.ac.uk/download/pdf/44389468.pdf
[27]Stránka projektu Redis https://redis.io
[28]Replikácia viď. https://redis.io/topics/replication

dokáže ukladať a manipulovať so **zložitými dátovými typmi**. Výhodou je jeho vynikajúca rýchlosť, jednoduchosť a **atomická manipulácia s dátami**, ktoré sú v ňom uložené. Vďaka týmto vlastnostiam je možné ho využívať na miestach, ktoré by sme nemohli realizovať pomocou klasickej relačnej databázy. Najčastejšie sa využíva na ukladanie dočasných dát, v tomto smere sa stáva konkurenciou oproti klasickému serveru *Memcache*[29] používanom po dlhé roky nielen pri vývoji webových aplikácií. V novej verzii umožňuje podporu *distribuovaných serverov* a využitie paradigmat ako **publish a subscribe**[30], čím môže nahrádzať technológiu *RabbitMQ*[31], ktorá je pomerne rozšírená pri vývoji webových aplikácií na zasielanie asynchrónnych notifikácií. [21]

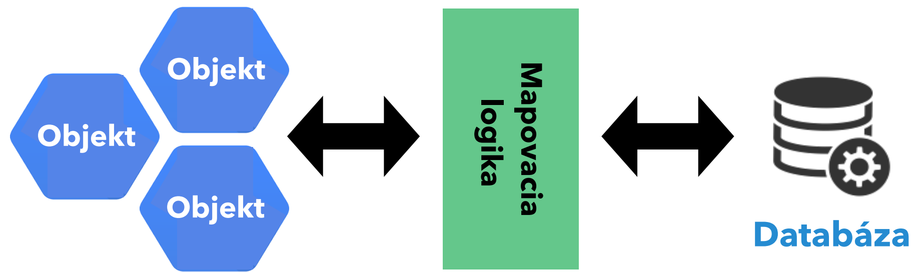
3.8 Doctrine ORM

Skratka ORM označuje technológiu nazývanú *object relational mapping*[32]. Je to metodológia a mechanizmus pre **objektovo orientované systémy**, ktorým poskytuje spôsob ukladania ich **perzistentných dát** prostredníctvom **databázy**. Nezáleží pritom, či táto databáza je relačná alebo *NoSQL*[33]. Prvá implementácia ORM bola využitá v systéme *Hibernate*[34], ktorý bol vytvorený pre systémy písané v jazyku *Java*. [15]

Pre programátora implementujúceho objektovo orientovaný systém je výhodnejšie pracovať s dátami prostredníctvom **objektov**, namiesto písania SQL požiadaviek. Objektovo relačné systémy umožňujú prepojenie **tabuliek databázy s objektami**, ktoré sa vyskytujú v systéme. Tieto objekty sú nazývané aj *DTO - data transfer objects*. Medzi entitami je možné definovať **kardinálne vzťahy**. Implementácia systému ORM je schopná porovnávať zmeny v entitách a automaticky tvoriť skripty na **migrovanie medzi verziami**. [15]

Na implementáciu ORM systému vo webovej aplikácii je dostupný projekt **Doctrine**[35]. Doctrine ponúka plnohodnotný systém, ktorý umožňuje definovanie jednotlivých entít, typov parametrov entít a vzťahov medzi nimi. Prostredníctvom konzoly a príkazov je možné databázu vytvoriť alebo vykonať jej **aktualizáciu** pri zmene definície niektorej z entít.

Nad systémom Doctrine je implementovaná nadstavba **Kdyby/Doctrine**[36], ktorá implementuje systém ORM, ako rozšírenie pre využitie vo frameworku Nette, popísanom v sekcii 3.2. Svoje služby poskytuje celej aplikácii prostredníctvom *DI kontajnera*, viď 3.2.3.

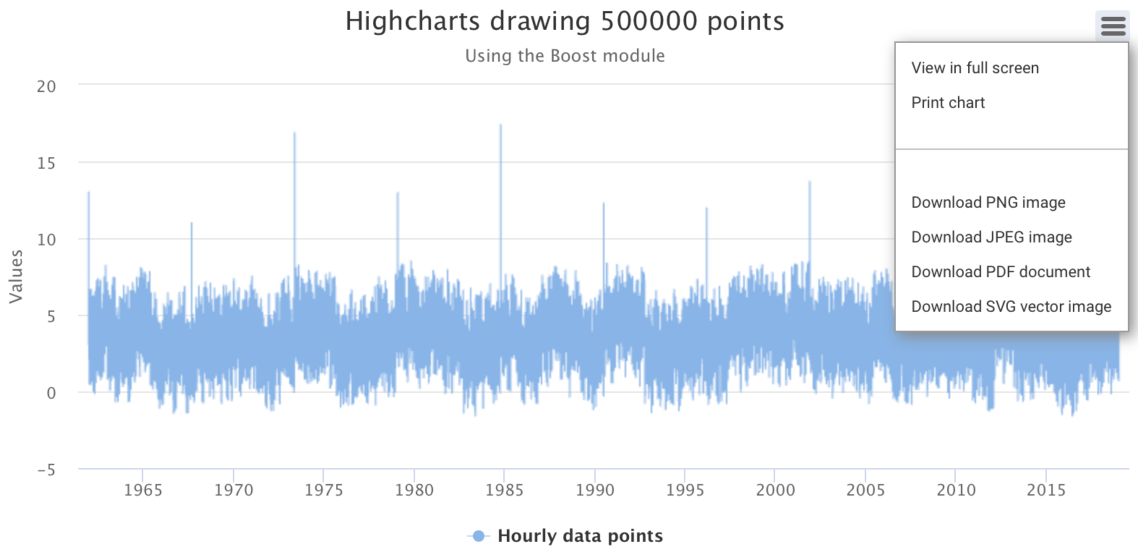


Obr. 3.6: Ilustrácia fungovania ORM systémov

[29] Stránka projektu Memcached https://memcached.org
[30] Popis publish-describe https://aws.amazon.com/pub-sub-messaging/
[31] Stránka projektu RabbitMQ https://www.rabbitmq.com
[32] Object relational mapping https://hibernate.org/orm/what-is-an-orm/
[33] NoSQL databázy - definícia https://www.mongodb.com/nosql-explained
[34] Stránka projektu Hibernate https://hibernate.org
[35] Stránka projektu Doctrine https://www.doctrine-project.org
[36] Stránka projektu Kdyby/Doctrine https://github.com/Kdyby/Doctrine

3.9 HighCharts

Projekt HighCharts[37] poskytuje viacero možností implementácie **grafov** vo webových aplikáciách. Veľmi dôležitý bod pri návrhu aplikácie, je voľba vhodných balíčkov. HighCharts je **plateným** riešením. Na rozdiel od iných implementácií, zobrazovanie grafov má veľmi dobré odozvy aj pri **veľkom počte dát**. Samotné použitie knižnice, ktorá je dostupná cez balíček, je veľmi jednoduché. Inicializácia grafu v základnom formáte nie je dlhšia ako pár desiatok riadkov. Všetky grafy je možné plno upravovať na mieru po stránke funkcionality aj vzhľadu. V prípade potreby je možné vykonať **export** priamo z grafu. Podporovaný je tabuľkový formát Excel alebo štandardizovaný formát CSV.



Obr. 3.7: Príklad využitia grafu knižnice HighCharts s 500 tisíc bodmi

3.10 Babel

Napriek tomu, že sa podarilo jazyk Javascript **štandardizovať**, existuje veľké množstvo interpretov, ktoré prekladajú tento jazyk na vykonávateľný kód priamo v prehliadači. S každou ďalšou verziu prehliadača je tento interpret aktualizovaný aby dokázal odrážať **veľkú rýchlosť vývoja štandardu** a funkcií. Pre zachovanie kompatibility aj na starších prehliadačoch alebo tých, ktorý nepodporujú všetky časti štandardu, existuje nástroj **Babel**[38], ktorý umožňuje preložiť webovú aplikáciu napísanú v jazyku Javascript tak, aby bola funkčná na predom zadanej **množine webových prehliadačov**. Spôsob fungovania nástroja Babel je jednoduchý. V konfiguračnom súbore sa nastavia jednotlivé definície podporovaných prekladačov. Možné je zvoliť konkrétne pokrytie koncových užívateľov v percentách. Na vstup sa privedie nekompatibilný JavaScript kód, ktorý je následne analyzovaný. Všetky nekompatibilné volania sú **substituované** tými kompatibilnými. V niektorých prípadoch je do súboru doplnená závislosť, ktorá nahradí volania. Výsledkom je kód, ktorý je možný spustiť na všetkých prehliadačoch, definovaných v konfiguračnom súbore.

[37]Stránka projektu HighCharts https://www.highcharts.com
[38]Stránka projektu Babel https://babeljs.io

```
class Test {
    constructor(name) { this.name = name; }
    logger () { console.log("Hello", this.name); }
}
```

Babel

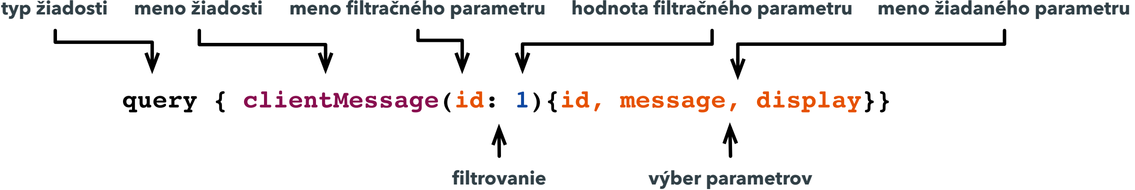
```
function _classCallCheck(instance, Constructor) {
    if (!(instance instanceof Constructor)) { throw new TypeError("Cannot call a class as a function"); }
}

var Test = function () {
    function Test(name) { _classCallCheck(this, Test); this.name = name; }
    Test.prototype.logger = function logger() { console.log("Hello", this.name); };
    return Test;
}();
```

Obr. 3.8: Príklad prekladu nekompatibilného kódu nástrojom Babel

3.11 GraphQL API

GraphQL[39] je framework na tvorbu **aplikačných rozhraní**, ktorý vyvinula firma Facebook v roku 2016 a následne vydala špecifikáciu spolu s referenčnou implementáciou. Ponúka alternatívu k rozhraniam založených na REST[40] notácii.

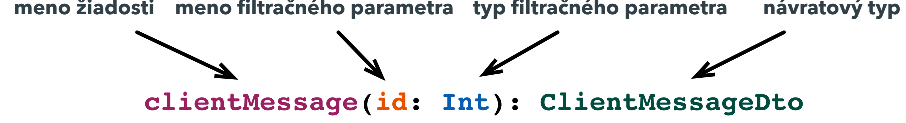
Základným stavebným kameňom je **jazyk požiadaviek**, ktoré framework využíva na komunikáciu so serverom. Dáta sú odosielané metódou POST vo formáte *JSON* na server. V prípade, že klient chce získať informácie z databázy servera, obsah tvorí žiadosť typu *query*. Druhou možnosťou je žiadosť o zmenu alebo zmazanie dát. V tomto prípade sa používa typ *mutation*. Štruktúra syntaxe žiadostí GraphQL sa skladá z dvoch hlavných častí. Pre typ query slúži prvá časť na filtrovanie výstupných dát, v prípade mutácii k zmene uložených dát. Druhá časť slúži na **výber parametrov objektu**, ktoré server vráti. Vďaka tejto možnosti je klient schopný požiadať o dáta ktoré naozaj potrebuje. Príklad takejto žiadosti ilustruje obrázok 3.9. [9]



Obr. 3.9: Príklad žiadosti GraphQL

Žiadosť je definovaná vlastnou **syntaxou GraphQL**. Táto syntax je tvorená podľa množiny pravidiel, ktoré sú nazývané *schemes*. Pomocou nich klient v požiadavke definuje objekty a ich parametre, ktoré ho zaujímajú a plánuje s nimi ďalej pracovať. Každé pravidlo obsahuje **návratový typ objektu**, ktorý je spätne zasielaný klientovi. [9]

[39] Stránka projektu GraphQL https://graphql.org
[40] Definícia REST API https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/



Obr. 3.10: Príklad pravidla GraphQL

Schémy definujú objekty, ktorých parametre sú nazývané *fields*. Každému parametru objektu je možné priradiť dátový typ označovaný *type*. Využívané sú buď preddefinované typy, ktoré sú obsiahnuté v samotnom systéme GraphQL alebo nové objekty, ktoré sú definované pri implementácii servera. [9]

meno typu objektu

definícia typu →

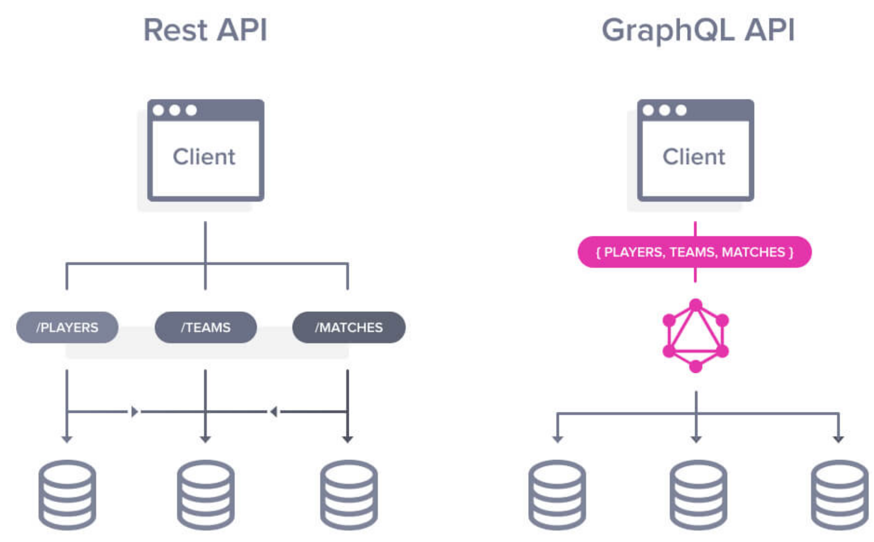
```
type ClientMessageDto {
    changed: LocalDateTime
    id: Int
}
```

← typ parametru objektu

parameter objektu →

Obr. 3.11: Príklad objektu GraphQL

Výhodou GraphQL rozhrania oproti známemu REST, je možnosť **zmeny požiadavky bez nutnosti úpravy na strane servera**. Rozdielom je aj samotná komunikácia, ktorá napriek tomu, že prebieha skrz HTTP požiadavky neodkazuje na rôzne prípojné body. Namiesto toho sa všetky žiadosti posielajú len na jednu URI a výsledok sa generuje podľa zaslanej žiadosti. Tieto žiadosti je nutné buď generovať alebo manuálne zapisovať. Lepšou voľbou je využitie SDK, ktoré umožní automatickú tvorbu takýchto požiadaviek a zabráni výskytu chýb, ktoré by mohli byť do kódu zanesené manuálnym zápisom.



Obr. 3.12: Rozdiel medzi REST a GraphQL rozhraním [17]

Kapitola 4

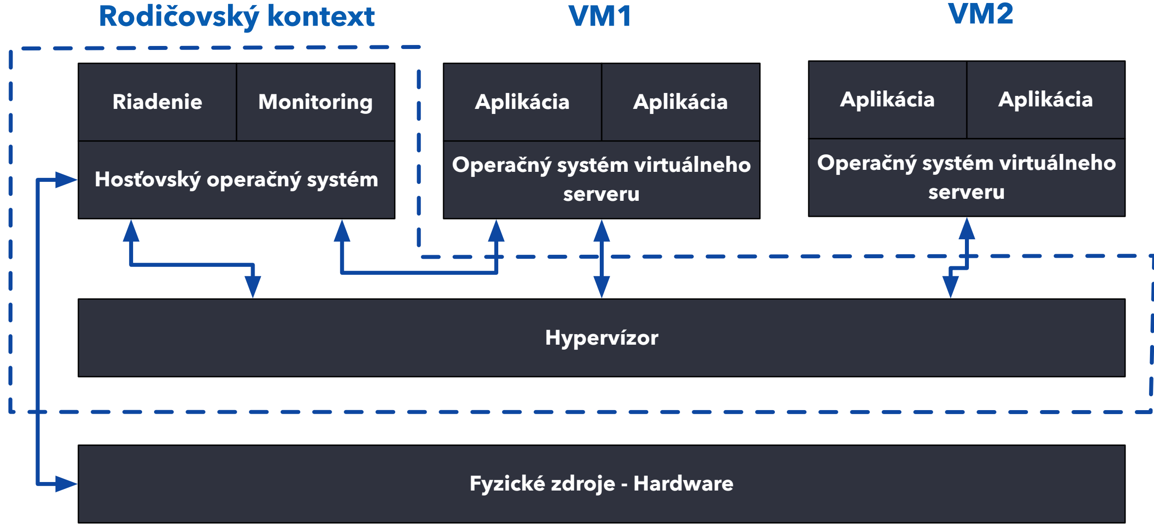
Virtualizácia

Virtualizácia je pojem používaný v prípade, keď je vytváraný **virtuálny obraz** servera, operačného systému, úložiska alebo sieťových zdrojov. Vo výsledku dosiahneme **viacnásobné použitie** týchto prvkov v rovnakom čase. Hlavným účelom virtualizácie je transformovanie využívania klasických fyzických serverov na **škálovateľné**, ekonomické a efektívne riešenie, ktoré prináša úsporu finančných nákladov a množstva spotrebovanej elektrickej energie. [13] Kapitola skúma proces virtualizácie ako taký a analyzuje jednotlivé **virtualizačné technológie**, pričom podrobne popisuje architektúru týchto systémov a jej jednotlivé komponenty, ktoré medzi sebou kooperujú.

4.1 Architektúra virtualizačných technológii

Klasickým modelom využívaným pri virtualizácii prostredia je systém s viacerými virtuálnymi zariadeniami využívajúcimi jeden procesor. Na riadenie poskytovaného výpočtového výkonu sa používa prvok nazývaný **hypervízor**. Ako u bežných procesov riadi operačný systém prideľovanie času procesora medzi jednotlivé aplikácie, plní rovnakú úlohu vo virtualizovaných systémoch práve tento element. Hypervízor zabezpečuje kontrolu nad **riadením fyzickej pamäte**, ktorá je dostupná pre jednotlivé zariadenia. Pre zabezpečenie ilúzie, že každé zariadenie disponuje vlastnou fyzickou pamäťou, poskytuje každému zariadeniu **virtualizovanú pamäť**. Túto pamäť následne mapuje na fyzickú, uloženú na hostiteľskom serveri. V prípade **I/O zariadení** sa musí postarať o zdieľaný prístup k zariadeniam z reálneho sveta. *Rodičovský kontext* v tomto prípade využíva vlastné ovládače zariadení a virtuálnym serverom poskytuje len akýsi všeobecný pohľad na tieto externé zariadenia a prístup k nim. Pri **virtualizovaní siete** rozdeľujeme komunikáciu na tú medzi dvomi servermi nachádzajúcimi sa v rovnakom *clustery*[1] alebo medzi virtuálnym serverom a tým dostupným cez internet. V prvom prípade je potrebné zabezpečiť, aby prenos dát prebiehal len v rámci virtualizácie. Táto vlastnosť je riadená pomocou softvérového *switch prvku* v *rodičovskom kontexte*. Vzdialená komunikácia je emulovaná cez virtualizovanú sieť. Hypervízor ponúka každému virtuálnemu zariadeniu pocit, že beží na vlastnom fyzickom stroji. **Riadiaci softvér** bežiaci na rodičovskom serveri dokáže s ním komunikovať a zabezpečiť vytváranie nových virtuálnych serverov alebo riadenie ich napájania, čím emuluje všetky aspekty bežného serveru. Umožňuje **bootovanie** jednotlivých virtuálnych serverov alebo rozhoduje o mieste, kde majú byť fyzicky uložené. Častokrát disponuje svojim vlastným **webovým rozhraním** a službou **migrácie**. [11]

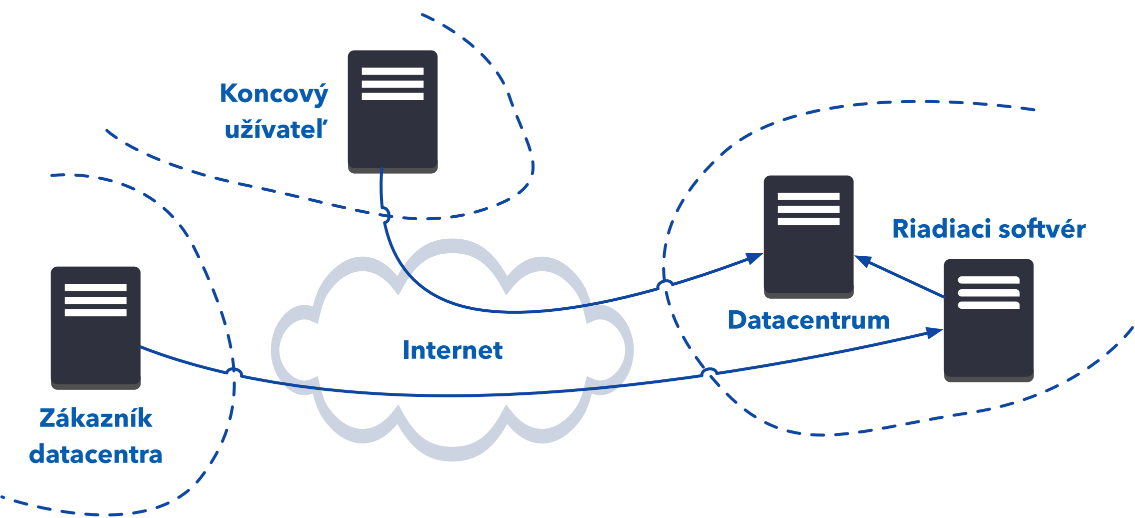
[1] Definícia clusteru https://it-slovnik.cz/pojem/cluster



Obr. 4.1: Architektúra virtualizovaného systému [11]

4.2 Pojem cloud computing

Cloud computing je pomerne nový koncept spájajúci v sebe webové služby, virtualizáciu, *SOA*[2] a *grid computing*[3]. Je založený na princípe virtualizácie, zdieľania a dynamického rozširovania zdrojov. Tieto zdroje sú následne poskytované ako **služba** prostredníctvom internetu. Za ich využívanie si poskytovateľ účtuje peniaze podľa konfigurácie, ktorú zákazník využíva. Medzi najpoužívanejšie virtualizačné nástroje patrí **VMware** popisovaný v kapitole 4.5 a **OpenStack** popisovaný v kapitole 4.4, ktorý je pomerne novým hráčom na trhu. Takto virtualizované prostredie poskytuje zdroje ako úložisko dát, výpočtový výkon alebo pripojenie do internetu. [19] [20]



Obr. 4.2: Architektúra konceptu cloud computing

[2] Popis SOA http://www.odbornecasopisy.cz/res/pdf/37516.pdf
[3] Grid computing https://azure.microsoft.com/en-in/overview/what-is-grid-computing/

4.3 Výhody použitia virtuálnych serverov

Zo strany zákazníka je výhodou možnosť prenajatia potrebných zdrojov bez nutnosti prevádzkovania celého fyzického serveru. Na rozdiel od neho je u virtuálnych serverov pomerne jednoduché zmeniť konfiguráciu. K zmene často dochádza bez nutnosti obnovy užívateľských dát alebo vypnutia operačného systému. Poskytovateľ má možnosť **zisku z prenájmu** takto poskytnutých zdrojov. Na rozdiel od fyzických serverov môže prenajímať výrazne vyšší výkon aké sú jeho dostupné zdroje. Táto výhoda je postavená na **rozdeľovaní výpočtového výkonu** podľa potreby medzi jednotlivé virtuálne servery. V prípade, že na niektorom zo serverov nedochádza k využitiu plnej kapacity, sú tieto zdroje uvoľnené a môžu byť poskytnuté ďalšiemu zákazníkovi. Pre zabezpečenie dostupnosti zdrojov poskytuje poskytovateľ zákazníkovi dokument *SLA*[4], ktorý udáva garantovanú **mieru dostupnosti**.

4.4 Virtualizácia technológiou OpenStack

OpenStack je **open-source systém**, ktorý je plno distribuovaný. Svoje služby sa snaží čo najviac oddeľovať a poskytovať čo najväčšie možnosti **konfigurácie** [19]. So systémom sa dá komunikovať prostredníctvom **príkazového riadka**, ktorý obsahuje skripty napísané v jazyku *Python*[5]. Ich použitím je možné ovládať prvky tejto architektúry, čo umožňuje plnú **automatizáciu procesov** bez potrebného zásahu ľudského prvku do systému.

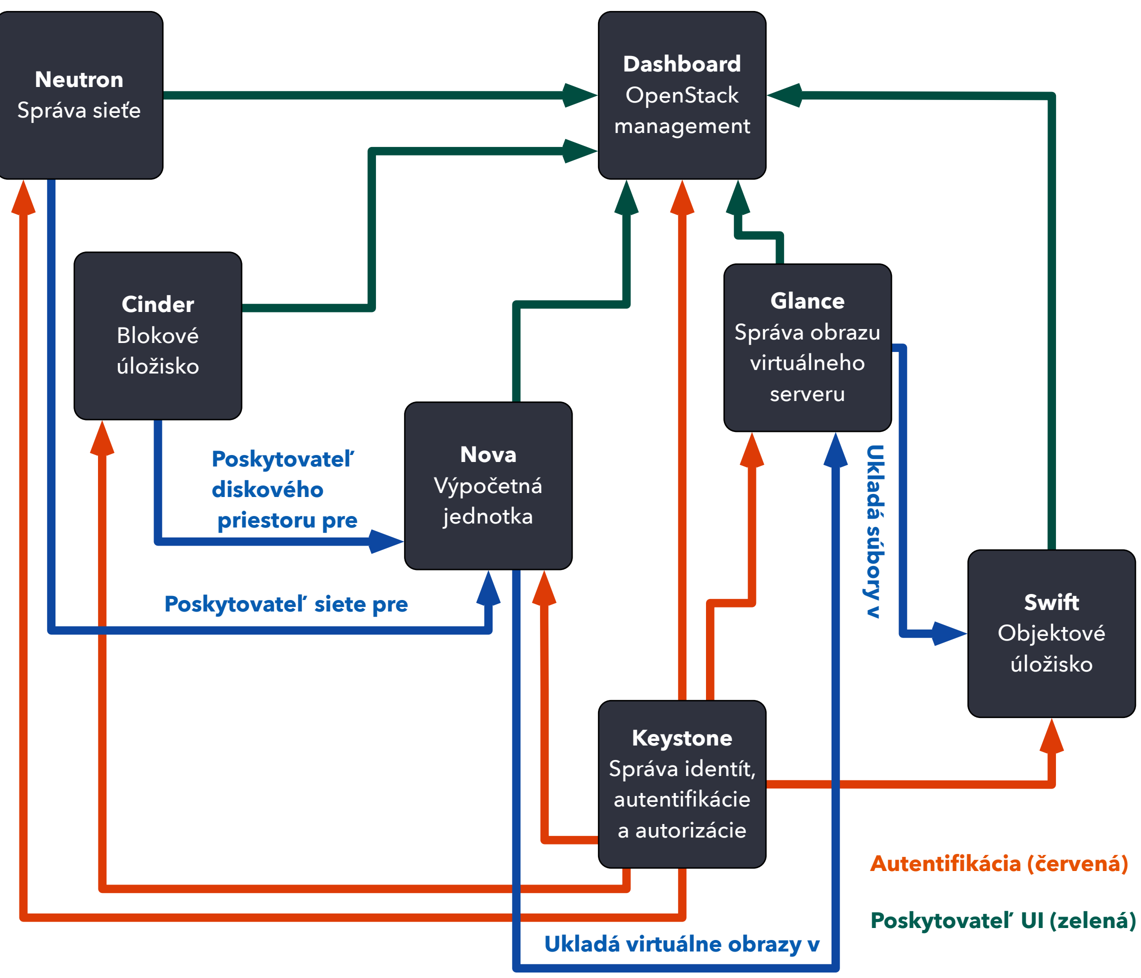
4.4.1 Architektúra clusteru

Beh systému zabezpečuje viacero riadiacich prvkov, ktoré medzi sebou navzájom komunikujú [19]. Ich podrobný popis sa nachádza v zozname nižšie a celkové grafické zobrazenie možno vidieť na obrázku 4.3. Informácie boli získané z citácie [16].

- **Horizon** - *dashboard*, poskytuje prístup k výpočtovým, diskovým a sieťovým zdrojom, ktoré využívajú virtuálne stroje. Dostupné je webové rozhranie so zoznamom všetkých prvkov architektúry, kde je možné nad nimi vykonávať rôzne operácie. Aplikácia je schopná generovať autentifikačné súbory, ktoré sú používané pri prístupe do systému.
- **Nova** - *compute*, poskytuje výpočtové služby pre vytváranie, mazanie, konfigurovanie alebo iné riadenie virtuálnych strojov. Stará sa o automatickú správu fyzických zdrojov, ktoré priraďuje virtuálnym serverom. Poskytuje hypervízor založený na technológiach *KVM, VMware, Xen, Hyper-V*[6] alebo technológii založenej na kontajneroch implementovaných priamo v linuxovom jadre, pomenovaných *LXC*[7].
- **Neutron** - *network*, poskytuje virtuálne sieťové služby *Nove*. Vďaka nemu je možné jednotlivým virtuálnym strojom pripájať vlastné siete. Je zodpovedný za správu IP adries, *DNS záznamov* alebo *DHCP servera*[8]. Umožňuje organizáciu a presúvanie záťaže nad sieťovými zdrojmi. Zabezpečuje bezpečnostné nastavenia siete ako *firewall*, *prístupové zoznamy*[9] a iné. Poskytovaný je *framework pre SDN*[10].

[4]Šablóna zmluvy SLA https://slatemplate.com
[5]Oficiálna stránka jazyka Python https://www.python.org
[6]Technológia Hyper-V https://www.cloudwards.net/hyper-v/
[7]LXC kontajnery https://linuxcontainers.org
[8]Popis protokolov DNS a DHCP https://study-ccna.com/dhcp-dns/
[9]Koncept prístupových zoznamov https://en.wikipedia.org/wiki/Access_control_list
[10]Definícia Software-Defined Networking (SDN) https://www.opennetworking.org/sdn-definition/

- **Glance** - *image*, ukladá virtuálne disky do objektového úložiska. Slúži pre ľudí, ktorí potrebujú organizovať veľký počet virtuálnych obrazov. V spolupráci s Nova a Swift poskytuje koncové riešenie pre správu obrazov diskov v cloude.
- **Cinder** - *block storage service*, je aplikácia a rozhranie, ktoré poskytuje trvalé blokové úložisko Nove. Navrhnuté je tak, aby poskytovalo tieto zdroje koncovým používateľom.
- **Swift** - *object storage service*, správa dát cloudu prostredníctvom technológie objektového úložiska. Tento systém je vysoko škálovateľný a podporuje redundanciu so zápisom rovnakých dát na viacero diskov. V prípade zlyhania je možné dáta obnoviť. Pri rozširovaní zdrojov stačí pridať nový server a zdroje sa automaticky rozšíria.
- **Manila** - *shared file system service*, správa dát cloudu z objektového úložiska. Poskytuje natívne a Amazon S3 kompatibilné API[11] pre prenos súborov.
- **Keystone** - *identity management*, autentifikuje a autorizuje všetky komponenty. Stará sa o správu autentifikačných tokenov a bezpečnú komunikáciu medzi administrátorom, systémom a medzi prvkami systému navzájom.



Obr. 4.3: Architektúra clusteru OpenStack

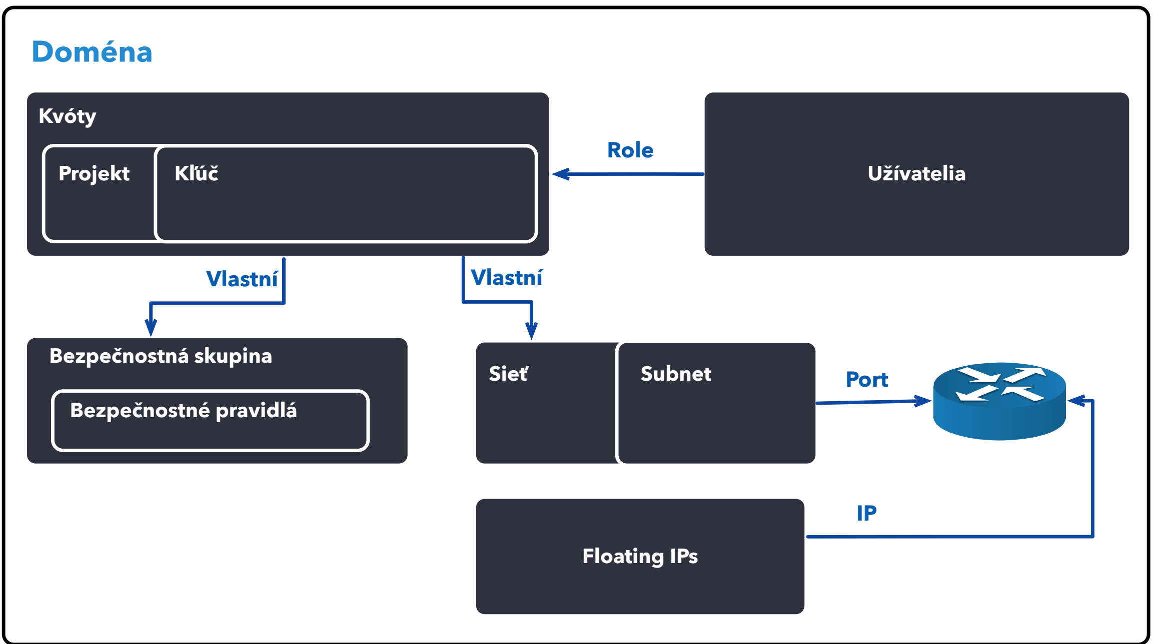
[11] Amazon S3 REST API https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

4.4.2 Zoznam entít systému

Okrem hlavných riadiacich objektov architektúry, ktoré sú pevne dané, sa v systéme nachádza viacero entít, ktoré sú priamo vytvárané alebo modifikované v prípade tvorby alebo iných úprav virtualizovaných serverov. Zoznam týchto entít nachádzajúcich sa v systéme OpenStack popisuje zoznam nižšie. Informácie boli opäť čerpané z citácie [16].

- **Domény** - *domains*, slúžia ako vysokoúrovňové kontajnery pre projekty, užívateľov a skupiny. Môžu sa používať na riadenie všetkých identifikačných komponent.
- **Projekty** - *projects, tentants*, je skupina žiadneho a viac užívateľov. V systéme *Nova* projekty vlastnia virtuálne zariadenia. V systéme *Swift* projekt vlastní vlastné *kontajnery*. Spolu s objektom užívateľa tvoria pár, ktorý môže mať priradenú *rolu*.
- **Role** - *roles*, je skupina oprávnení, umožňujúcich užívateľovi vykonávať dané operácie.
- **Bezpečnostné skupiny** - *security groups*, vystupujú v systéme ako firewall pre virtuálne servery a iné sieťové zdroje. Obsahujú zoznam jednotlivých pravidiel, ktoré špecifikujú sieťové prístupové pravidlá.
- **Kvóty** - *quotas*, bránia vyčerpaniu systémových zdrojov. Poskytujú operačné limity. Limitovať je možné rôzne zdroje ako napríklad maximálnu veľkosť projektu. Kvóty je možné nastavovať ako na jednotlivé projekty tak aj pár projekt-užívateľ.
- **Routery** - *routers*, je logická jednotka, ktorá posiela jednotlivé sieťové balíčky medzi sieťami. Poskytuje aj L3 vrstvu *OSI modelu*[12] a *NAT*[13] prekladania, pre poskytovanie prístupu do externej siete.
- **Siete** - *networks*, poskytujú izolovanú *L2 vrstvu OSI sieťového modelu*. Siete sa delia na tie, ktoré patria jednotlivým projektom a takzvané *provider networks*. *Projektové siete* sú plno izolované a nie sú zdieľané medzi inými projektami. *Provider networks* mapujú existujúcu fyzickú sieť a poskytujú prístup do externej siete pre servery. Jednotlivé siete môžu byť prepojované prostredníctvom *routerov*.
- **Podsiete** - *subnets*, reprezentujú zoznamy IP adries. Používajú sa na rezervovanie IP adries v prípade vytvorenia nových portov na sieti.
- **Porty** - *ports*, sú spojovacím bodom pre pripojenie jediného zariadenia. Port popisuje priradenú sieťovú konfiguráciu vo forme IP alebo MAC adresy použitej na porte.
- **Kľúčenky** - *keypairs*, tvoria *OpenSSH kľúče*[14] používané na pristupovanie k serverom. Kľúče je možné vytvárať priamo použitím príkazov OpenStack.
- **Konfigurácie** - *flavours*, definujú výpočtový výkon, veľkosť operačnej pamäte a úložiska inštancií vytvorených cez systém *Nova*.
- **Floating IPs** je zoznam dostupných verejných IP adries, ktoré sú dostupné v systéme a je ich možné priraďovať nad jednotlivými projektami.

[12] Definícia OSI modelu `https://www.ervin.sk/osi-model-a-siet/`
[13] Definícia technológie NAT `http://www.cs.vsb.cz/grygarek/TPS/projekty/0405Z/NAT/Nat.htm`
[14] Popis fungovania SSH kľúčov `https://help.ubuntu.com/community/SSH/OpenSSH/Keys`



Obr. 4.4: Entity systému OpenStack

4.5 Virtualizácia technológiou VMware

Konkurenciou oproti technológii OpenStack je technológia od spoločnosti VMware. Služba je ponúkaná pod licenciou a **nepatrí medzi otvorené platformy**. Firmám je poskytované riešenie *clusteru* spolu s technickou podporou. Po technickej stránke je systém rozdielny, ako na funkčnej, tak aj logickej úrovni. VMware ponúka viacero rozšírení oproti systémom OpenStack. Platforma predstavuje **robustné riešenie** v prípade veľkých počtov fyzických aj virtuálnych serverov, kde vyniká vynikajúcou možnosťou **škálovania** zdrojov.

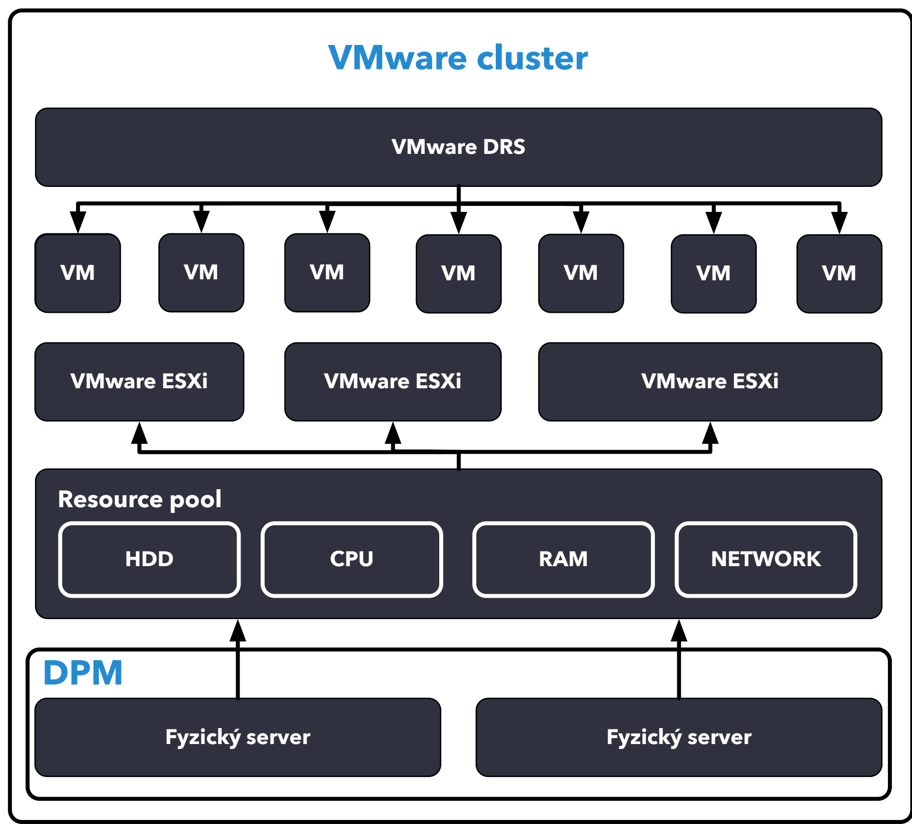
4.5.1 Architektúra clusteru

Cluster využíva *VMware's Distributed Resource Scheduler (DRS)*[15] na správu alokácie fyzických zdrojov, ktoré sú mapované na virtuálne stroje. Každý z týchto *hostov* využíva *VMware ESX hypervízor*[16]. *DRS* vykonáva *load balancing*[17] pre **zvýšenie výkonu** medzi jednotlivými virtuálnymi strojmi. *VMware Distributed Power Management (DPM)*[18] rozširuje funkcionalitu systému *DRS* s možnosťou **redukovania spotreby elektrickej energie**, tým že konsoliduje jednotlivé virtuálne stroje na menší počet hostiteľov. *DPM* v prípade zistenia, že niektorý hostiteľ nie je využívaný, ho vypne, čím zabezpečí **úsporu energie**. *DRM* ponúka aj iné funkcionality, ktoré umožňujú tvorbu **agilnej infraštruktúry**.[8]

Technológia *cluster abstraction* umožňuje **abstrakciu kolekcie jednotlivých hostov** tak aby bolo možné kombinovať ich zdroje. *Resource pool* abstrakcia ponúka **správu hierarchickej štruktúry** jednotlivých prvkov zdrojov na každej úrovni. Funkcia *initial placement* priradí špecifického hostiteľa v *clustery* danému VM pri jeho zapnutí. Dostupný

[15]VMware DRS https://www.vmware.com/products/vsphere/drs-dpm.html
[16]VMware ESX, oficiálna dokumentácia https://www.vmware.com/products/esxi-and-esx.html
[17]Load balancing technológia https://www.sectec.sk/bezpecnost/load-balancing
[18]DPM technológia http://www.waldspurger.org/carl/papers/drs-vmtj-mar12.pdf

je **režim údržby**, ktorý umožňuje jednoduché aktualizácie hostov, pričom aktuálne bežiace VM presunie na ďalší dostupný zdroj. **Zdieľanie zdrojov** je možné kontrolovať pomocou viacerých techník. *Reservations* zabezpečujú možnosť priradiť danému VM garantované množstvo zdrojov. Je úlohou administrátora aby potom zabezpečil že toto číslo nepresiahne ich celkový počet. *Limits* špecifikujú **maximálne množstvo skonzumovaných zdrojov**. *Shares* udávajú množstvo prostriedkov, ktoré môže VM skonzumovať proporcionálne s jeho zdieľanou alokáciou.[8]



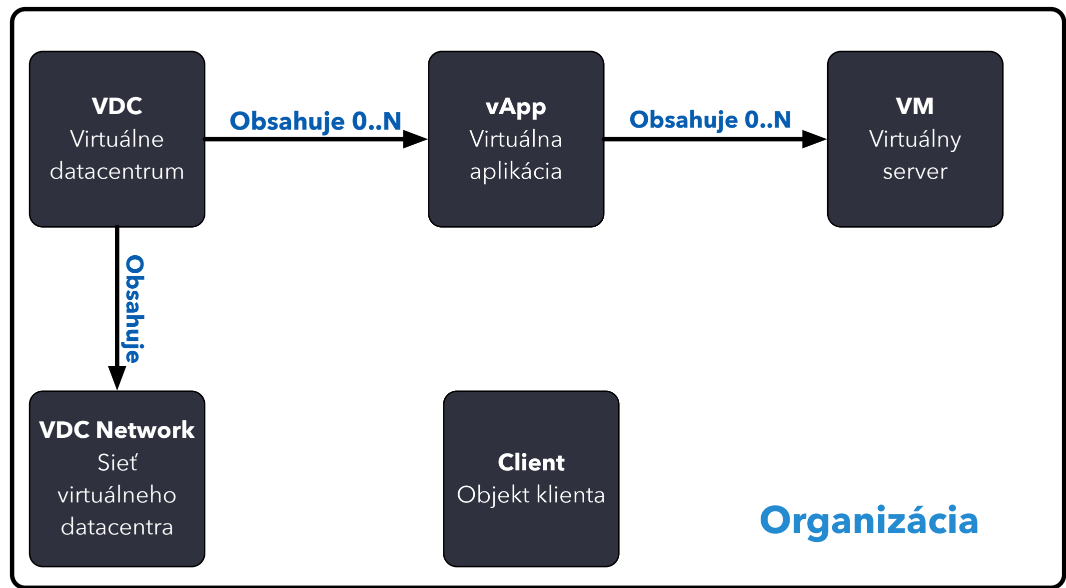
Obr. 4.5: Architektúra clusteru VMware

4.5.2 Zoznam entít systému

Ekosystém VMware je okrem blokov, ktoré zabezpečujú jeho funkčnosť zložený aj z logických entít, ktoré sú uložené v systéme. Tieto entity je možné vytvárať alebo nad nimi vykonávať rôzne akcie. Zoznam jednotlivých takýchto entít popisuje zoznam nižšie.

- **Client** - *client*, jednotka obsahujúce údaje o klientovi, ktorý spravuje zdroje. Prostredníctvom klienta je možná autentifikácia a autorizácia užívateľa do organizácie.
- **Organization** - *org*, slúži ako abstrakcia pre skupinu užívateľov, ktoré majú oprávnenia na manipuláciu so zdrojmi, ktoré sú jej pridelené. Obsahuje viacero inštancií virtuálnych serverov a reprezentuje logickú bezpečnostnú hranicu. Môže jej byť priradená sieť a obsahuje virtuálne dátové centrá.
- **Virtual datacentre** - *vdc*, abstrahuje virtuálne datacentrum. Obsahuje viacero aplikácii a je v ňom možné vytvárať sieťové služby, ktoré spadajú pod organizáciu.

- **Virtual application** - *vAPP*, *kontajner* pre *distribuované softwarové riešenie*. Aplikácie slúžia ako abstrakcia nad jednotlivými virtuálnymi strojmi, kde je ich možné prepájať do rôznych infraštruktúr.
- **Virtual machine** - *vm*, je virtuálny server, ktorý nahrádza ten fyzický.



Obr. 4.6: Entity systému VMware

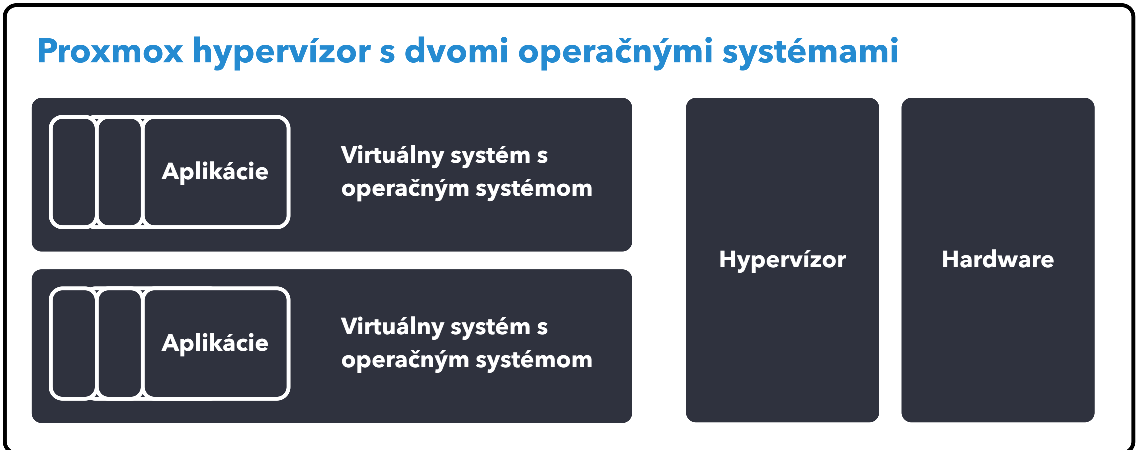
4.6 Virtuálne privátne servery

Jedná sa o **softwarové emulovanie fyzického serveru** pomocou technológie *hypervising*, ktorá beží na hosťovskom serveri. Jeden takýto hostiteľ môže virtualizovať viacero VPS serverov, pričom každý z nich má inú konfiguráciu a aplikácie. Server má pridelené zdroje vo forme procesorového výkonu, pamäte a veľkosti dátového úložiska. Pri úložisku sa výkon udáva v *IOPS* čo je počet vstupno-výstupných operácií za sekundu. Oproti VMware a OpenStack je VPS o niečo menej komplexné, avšak **flexibilné riešenie virtualizácie** a ďalšieho predávania služieb. Po vytvorení serveru je možné sa k nemu pripojiť a získať tak plný prístup nad jeho správou. [5]

4.6.1 Proxmox VE

Proxmox[19] je implementácia pre technológiu VPS. Nad fyzickým hardware je implementovaná vrstva *hypervízora*, ktorý spravuje fyzické zdroje. Na *hypervízor* sú napojené jednotlivé virtuálne počítače, pričom na každom z nich môže bežať **vlastný operačný systém**. Proxmox VE sprístupňuje možnosť tvorby viacerých typov virtualizovaných systémov. Jednotlivé komponenty architektúry tohto systému zobrazuje obrázok 4.7.

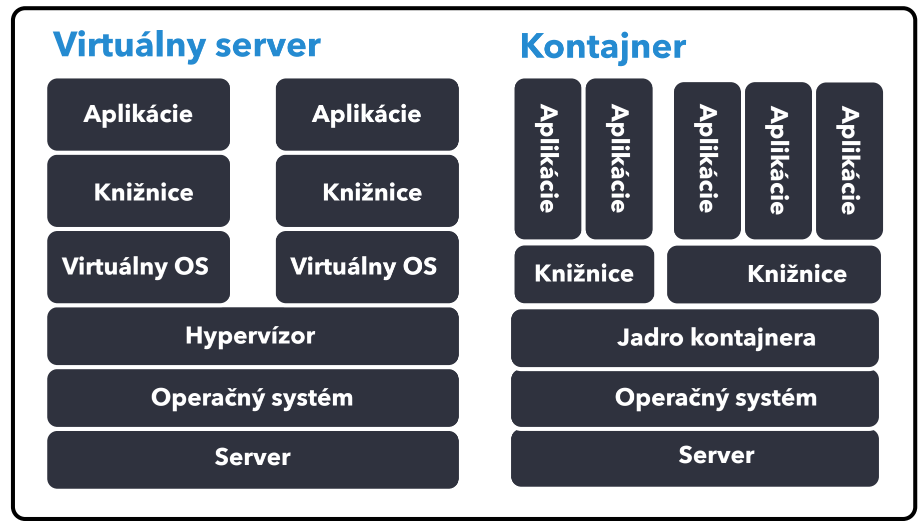
[19] Stránka projektu Proxmox VE https://www.proxmox.com/en/



Obr. 4.7: Architektúra Proxmox VE

Virtualizované systémy je možné tvoriť viacerými spôsobmi. *KVE - kernel based virtual machine* je technológia, ktorá bola zahrnutá do jadra linuxového systému už v roku 2007. Tento **modul kernelu systému** umožňuje užívateľom využívať výhod architektúry rozšírení **hardwarovej virtualizácie**. *QUEMU - quick emulator* poskytuje emuláciu a virtualizáciu rozhrania, ktoré môže byť **skriptované** alebo priamo kontrolované. Aj bez využitia Proxmox je možné upravovať tieto virtualizované inštancie a pridávať im virtuálne disky alebo meniť ich stav. Možná je **kombinácia QUEMU a KVE** virtualizácii. [6]

Technológia Proxmox umožňuje vytvárať *LXC kontajnery*. Na rozdiel od virtuálnych serverov **zdieľajú komponenty operačného systému** ako knižnice a spúšťateľné súbory s hosťovským operačným systémom. Pri virtuálnych serveroch nie je táto vlastnosť dostupná. Výhodou tejto technológie je **väčšia rýchlosť a menej spotrebovaných zdrojov**. Treba prihliadať na fakt, že pri dnešnom znižovaní cien komponentov stráca táto výhoda na dôležitosti. *Proxmox kontajner* je možné vytvoriť použitím nástroja *LXC*. [6]



Obr. 4.8: Rozdiel medzi kontajnerom a virtuálnym serverom

Kapitola 5

Technologické možnosti

Na základe teoretických poznatkov popísaných v sekciách 4.5 a 4.4 kapitoly 4 je možné definovať **metódy komunikácie s virtualizačnými technológiami**. Prostredníctvom týchto komunikačných kanálov je možné vytvorenie scenárov pre automatickú správu poskytovaných služieb. Každý scenár podrobne popisuje **vytváranie jednotlivých entít systému** a možnosti následného spájania týchto menších prvkov medzi sebou pre vykonanie zložitejšej úlohy ako je napríklad tvorba nového servera alebo zmena konfigurácie existujúcej inštancie.

5.1 Návrh automatizácie systému OpenStack

Táto sekcia nadväzuje na sekciu 4.4. Komunikácia so systémom prebieha prostredníctvom **aplikácie napísanej v jazyku Python** s menom `openstack`. Tá umožňuje riadenie všetkých komponentov systému popísaných v sekcii 4.4.1. Prvým argumentom aplikácie je *komponent clusteru*, ktorá má za danú funkcionalitu zodpovednosť. V jednotlivých príkladoch sú veľkými písmenami vyznačené vstupy, ktoré je potrebné doplniť podľa požiadaviek. Pred vytvorením každej entity definovanej v sekcii 4.4.2 alebo jej modifikovaním je potrebné overiť, či neexistuje v systéme. Na získanie zoznamu všetkých entít konkrétneho druhu sa použije príkaz `openstack ENTITA list`. Všetky príkazy, ktoré vytvárajú prvky v systéme, podporujú výstup vo formáte JSON po zadní parametru `-f json`. Pri každej entite je možné zadať popis, ktorý bližšie popisuje tento prvok pomocou parametru `description`.

5.1.1 Autentifikácia požiadaviek

Existujú dva druhy možnej autentifikácie do systému OpenStack. V prvom prípade je možné vykonať prihlásenie ako **administrátor systému**. Na autentifikáciu je využívaný *RC súbor*, ktorý je možné získať prostredníctvom *dashboardu*. Tento súbor je potom vykonaný v termináli spolu s každou požiadavkou. V prípade druhého spôsobu autorizovania ide o prihlásenie pod **užívateľským účtom** konkrétneho klienta. Spôsob je rovnaký ako v prvom prípade. Niektoré komponenty systému je lepšie vytvárať pod užívateľským prihlásením z hľadiska bezpečnosti. Výhodou je, že všetky zoznamy prvkov na výstupe príkazov obsahujú filtrované výsledky podľa prihláseného používateľa. Podrobnejší spôsob autorizácie popisuje sekcia implementácie 8.3. Príklad konfiguračného súboru vidno na obrázku 5.1.

```
export OS_AUTH_URL=https://mirantis.master.cz:5000/v3
export OS_PROJECT_ID=${PROJECT_ID}
export OS_PROJECT_NAME=${PROJECT_NAME}
export OS_USER_DOMAIN_NAME="Default"
if [[ -z "$OS_USER_DOMAIN_NAME" ]]; then unset OS_USER_DOMAIN_NAME; fi
unset OS_TENANT_ID
unset OS_TENANT_NAME
export OS_USERNAME=${USER_NAME}
export OS_REGION_NAME="RegionOne"
if [[ -z "$OS_REGION_NAME" ]]; then unset OS_REGION_NAME; fi
export OS_INTERFACE=public
export OS_IDENTITY_API_VERSION=3
export OS_PASSWORD=${PASSWORD}
```

Obr. 5.1: Príklad konfiguračného systému autentifikácie

5.1.2 Vytvorenie používateľa

Pri požiadavke na vytvorenie používateľa je potrebné nastaviť **prihlasovacie údaje**, ktoré bude klient používať pri pripojení na server alebo do dashboardu. Ako prvý parameter je prijímané užívateľské meno, parametrom `password` je možné zadať heslo pre prístup a parametrom `email` emailovú adresu pre zasielanie upozornení na mail. Výsledkom funkcie je **serializovaný objekt vytvoreného používateľa**.

```
openstack user create MENO --password HESLO --email EMAIL
```

5.1.3 Vytvorenie projektu

Pred vytvorením projektu je potrebné vytvoriť doménu prvým príkazom, ktorý na vstupe prijíma len názov vytváranej *domény*. Na ňu bude naviazaný konkrétny projekt. Jeho vytvorenie je možné inicializovať použitím druhého príkazu, ktorý prijíma názov projektu ako prvý argument, a parameter `domain` s názvom vytvorenej domény.

```
openstack domain create MENO && ... project create MENO --domain DOMÉNA
```

Vytvorenému projektu je potrebné **priradiť užívateľa a rolu**, ktorú bude mať v rámci projektu. Na priradenie užívateľa použijeme príkaz, ktorý má na vstupe názov *role*, ktorá by už mala byť definovaná v systéme, meno užívateľa cez parameter `user` a názov projektu cez parameter `project`. V príklade je použitá rola *_member_*.

```
openstack role add "_member_" --user UŽÍVATEĽ --project PROJEKT
```

Pripojenie na server je možné až po povolení *SSH a ICMP komunikácie*. Konfigurácia sa nastavuje pre *bezpečnostnú skupinu*, ktorú je možné získať z vytvoreného projektu. Po získaní *bezpečnostnej skupiny* je možné definovať *bezpečnostné pravidlá*. Každé pravidlo prijíma ako argument `project` názov projektu, `ingress` alebo `outgress` ako parameter smeru toku, `ethertype` definuje verziu IP protokolu, `protocol` povoľovaný protokol a `port` definuje názov portu. Vzdialenú IP adresu konfiguruje parameter `remote-ip`.

```
openstack security group rule create SKUPINA --project PROJEKT --ingress \
--ethertype IPv4 --protocol tcp --dst-port 22:22 --remote-ip 0.0.0.0/0
```

```
openstack security group rule create SKUPINA --project PROJEKT --ingress \
--ethertype IPv4 --protocol icmp --remote-ip 0.0.0.0/0
```

Posledným krokom je **nastavenie kvót projektu**, ktoré bude môcť užívateľ využívať. Príkaz prijíma parameter `instances`, ktorý udáva maximálny počet vytvorených virtuálnych serverov, `cores` maximálny počet jadier, `ram` maximálnu veľkosť operačnej pamäte, `floating-ips` maximálny počet verejných IP adries, `gigabytes` maximálnu veľkosť inštancie, `snapshots` maximálny počet zálohovacích obrazov, `volumes` maximálny počet diskových jednotiek, `secgroups` maximálny počet bezpečnostných skupín, `networks` maximálny počet sieťových entít, `routers` maximálny počet *routerov.* Potrebné je zadať aj názov projektu ktorému nastavujeme kvóty.

```
openstack quota set --instances 9 --cores 9 --ram 51200 --floating-ips 1 \
--gigabytes 1000 --snapshots 10 --volumes 10 --secgroups 10 --networks 3 \
--routers 1 PROJEKT
```

5.1.4 Vytvorenie siete

Vstupným bodom pri **vytváraní sieťového pripojenia** je vytvorenie *routera.* Nový router sa vytvára definovaním jeho názvu, `project` je parameter, ktorý definuje názov projektu ktorému router konfigurujeme. Na pripojenie *verejnej brány* je nutné použitie druhého príkazu, ktorý na vstupe očakáva názov už vytvoreného routera a meno verejnej brány, ktorá už v systéme existuje cez parameter `external-gateway`.

```
openstack router create NÁZOV --project PROJEKT
openstack router set NÁZOV --external-gateway BRÁNA
```

V ďalšom kroku je potrebné **vytvoriť objekt siete**, ktorý priradíme požadovanému projektu a uložíme ho pod názvom, ktorý si zvolíme.

```
openstack network create NÁZOV --project PROJEKT
```

Po úspešnom vytvorení siete sa tejto sieti **priradí subnet**, ktorý je možné pomenovať. Parameter `ip-version` definuje verziu IP protokolu, `dns-nameserver` nameserver, `subnet-range` rozsah. Príkaz nižšie zobrazuje príklad vytvorenia takéhoto *subnetu.*

```
openstack subnet create NÁZOV --network SIEŤ --ip-version 4 \
--subnet-range 192.168.1.0/24 --dns-nameserver "81.31.33.19"
```

Po vytvorení siete a subnetu nasleduje **vytvorenie prepojenia tejto siete na router** použitím mena portu a parametrov `network` s názvom siete a `device` s názvom routera. V poslednom kroku sa vytvorí *rozhranie* na routeri, ukazujúce na vytvorený subnet.

```
openstack port create NÁZOV --network SIEŤ --device ROUTER
openstack router add subnet ROUTER SUBNET
```

5.1.5 Vytvorenie inštancie

Vytvorením užívateľa, projektu a siete je možné vytvárať jednotlivé inštancie, ktoré budú spolupracovať s týmito prvkami. Na začiatku je potrebné **vytvorenie SSH kľúča**, ktorý umožňuje pripojenie k serveru cez štandardný príkaz *ssh.*

```
openstack keypair create NÁZOV
```

Vytvorenie inštancie je jednoduché a postačuje zadať *obraz*, z ktorého sa bude server klonovať parametrom `image` spolu s konfiguráciou predanou parametrom `flavor`. Konfigurácia definuje rezervovanie zdrojov a nastavenie obmedzení pri procesore, pamäti a úložisku. Vytvorenie je možné vyvolať zavolaním príkazov nižšie, pričom prvý na výstupe vypíše zoznam dostupných konfigurácií a v druhý vytvorí inštanciu.

```
openstack flavor list --quote all
openstack server create --image OBRAZ --flavor KONFIGURÁCIA --key-name KLÚČ
```

Vytvorená inštancia potrebuje pre prístup do verejnej siete IP adresu. Tá sa získa z alokačného zoznamu a následne je predaná na priradenie k inštancii.

```
openstack floating ip create SIEŤ
openstack server add floating ip INŠTANCIA IP
```

5.2 Návrh automatizácie systému VMware

Táto sekcia nadväzuje na sekciu 4.5. Interakcia s virtualizačnou technológiou VMware je umožnená prostredníctvom nástroja *VMware vCloud Director*, ktorý slúži pre poskytovateľov služieb a zabezpečuje manipuláciu s jeho zdrojmi. Director má implementované **jednoduché REST API**, ktoré zabezpečuje autentifikáciu požiadaviek. Architektúra sa skladá rovnako ako u OpenStack z viacerých prvkov. Podrobnejší popis prvkov popisuje sekcia 4.5.1. Automatizovaná komunikácia s vCloud Directorom **prebieha na úrovni REST API pomocou SDK**, ktoré je dodávané spoločnosťou. Táto vlastnosť umožňuje vytvorenie **automatických scenárov**, ktoré budú vytvárať servery alebo meniť ich konfiguráciu. Všetky objekty definované v sekcii 4.5.2 sú v systéme uložené pomocou takzvaných *refs*, ktoré predstavujú *hashované odkazy* na dané objekty v systéme.

Pred každou požiadavkou **je dobré skontrolovať, či v systéme neexistuje daný objekt**, na získanie kolekcie jednotlivých objektov, ku ktorým je možné v systéme pristupovať, slúži objekt *Query*, ktorému môžeme nastaviť stránkovanie alebo filter prostredníctvom objektu *Query_Params*. Príklad takéhoto volania je uvedený nižšie.

```
VMware_VCloud_SDK_Query_Params()->setFields('name');
VMware_VCloud_SDK_Query_Params()->setFilter('name==ORGANIZACIA');
VMware_VCloud_SDK_Query()->queryRecords('organization',
VMware_VCloud_SDK_Query_Params()->getRecord()
```

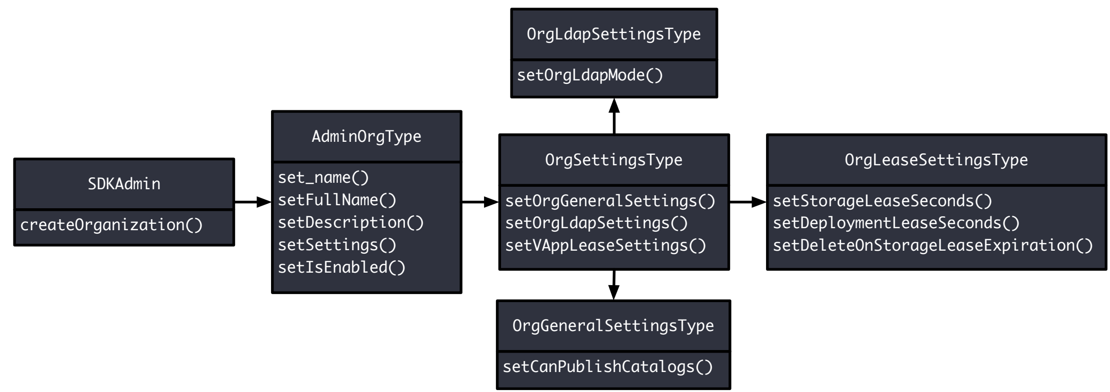
V prípade, že sa jedná o **asynchrónnu operáciu** a je nutné počkať na dokončenie predchádzajúcej akcie, existuje v SDK funkcia *getTasks*, ktorá získa zoznam aktuálne spracovávaných úloh. Po získaní konkrétnej úlohy, je možné zavolať funkciu *waitForTask*, ktorá zabezpečí čakanie. Pomocou tohto mechanizmu je možné vytvárať akési fronty požiadaviek.

5.2.1 Autentifikácia požiadaviek

Overovanie požiadaviek so systémom *vCloud Director* je realizované pomocou autentifikácie *basic auth*, ktorá obsahuje prihlasovacie meno a heslo. Takto formulovaná požiadavka sa pošle cez SDK funkciu *login* nad objektom *Service* v SDK. V odpovedi servera získame *token*, ktorý je možné ďalej využívať.

5.2.2 Vytvorenie organizácie

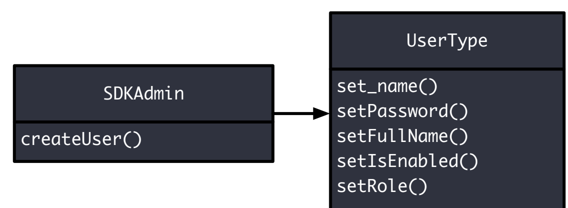
Vytvorenie začína inicializáciou viacerých objektov, ktoré sú spoločne zasielané na *vCloud Director API*. Nastavením objektu *OrgGeneralSettingsType*, je možné **zakázať možnosť publikovania zmien** do globálnych nastavení vCloud Directora, čo môže byť užitočné na zachovanie konzistentného stavu clustera. V prípade, že je používaný server *LDAP*, autentifikáciu určuje objekt *OrgLdapSettingsType*. **Uvoľňovanie zdrojov po určitom čase** je konfigurovateľné cez objekt *OrgLeaseSettingsType* a *OrgVAppTemplateLeaseSettingsType*. Všetky nastavenia sa zabalia do spoločného objektu *OrgSettingsType*, ktorý má nadradený koreňový objekt *AdminOrgType*, ktorý obsahuje informácie o mene organizácie, popisu a slúži aj na možnosť jej **aktivácie alebo deaktivácie**. Nakoniec volaním funkcie SDK začne proces vytvárania organizácie.



Obr. 5.2: Schéma objektov pre vytvorenie organizácie

5.2.3 Vytvorenie klienta

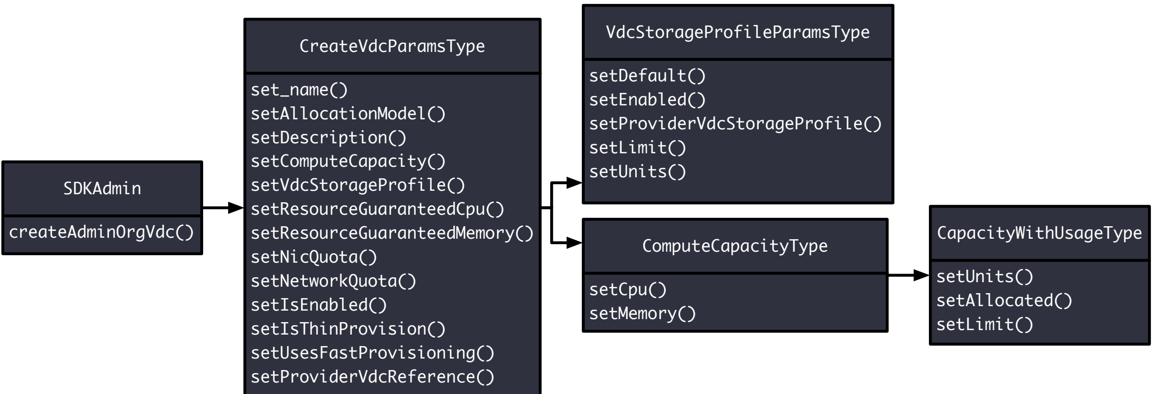
Pred vytváraním klienta je potrebné požiadať o **odkaz na rolu**, ktorá mu bude následne priradená. Odkaz je možné získať zavolaním SDK funkcie *getRoleRefs* s parametrom obsahujúcim meno role. Vytvorenie klienta začína inicializáciou objektu *UserType*, ktorý prijíma meno, heslo, celé meno a priezvisko. Pomocou neho je možné nastaviť užívateľa ako **predvoleného pre organizáciu**. Následne je možné **priradenie role**. Vytvorenie klienta sa uskutoční volaním SDK a funkcie *createUser*.



Obr. 5.3: Schéma objektov pre vytvorenie klienta

5.2.4 Vytvorenie virtuálneho datacentra

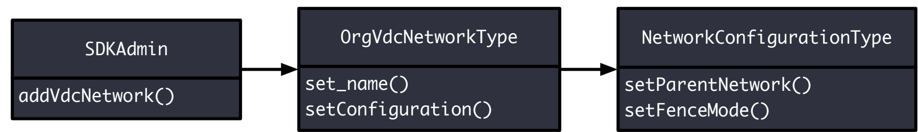
VDC potrebuje **závislosť na entitu poskytovateľa**. Odkaz je dostupný volaním API tentokrát funkciou *getProviderVdcRefs* a parametrom, ktorý obsahuje názov poskytovateľa. **Každé VDC potrebuje mať nastavený profil úložiska**, kam sa budú jeho virtuálne stroje fyzicky ukladať. Odkaz na profil poskytuje funkcia *getProviderVdcStorageProfileRefs*. VDC disponuje *kvótami zdrojov*, ktoré môže poskytovať svojím virtualizovaným serverom. Pre procesor a operačnú pamäť slúži objekt *CapacityWithUsageType*, ktorý obsahuje jednotku a hodnotu limitu. Tieto objekty sa zabalia do spoločného objektu *ComputeCapacityType*. Pre nastavenie limitov úložiska je potrebné využiť objekt *VdcStorageProfileParamsType*, ktorý odkazuje na profil úložiska a definuje limit veľkosti s jednotkou. Samotné VDC vytvára objekt *CreateVdcParamsType*, ktorému sa nastavuje názov, popis, alokačný model, limity procesora a operačnej pamäte, limity profilu úložiska a garantované hodnoty procesora a operačnej pamäte. **Umožňuje zapnúť alebo vypnúť funkcie** *ThinProvisioning* a *FastProvisioning*. Potrebné je **priradenie odkazu na poskytovateľa**. Vytvorenie VDC je dostupné zavolaním funkcie *createAdminOrgVdc*.



Obr. 5.4: Schéma objektov pre vytvorenie virtuálneho datacentra

5.2.5 Vytvorenie siete VDC

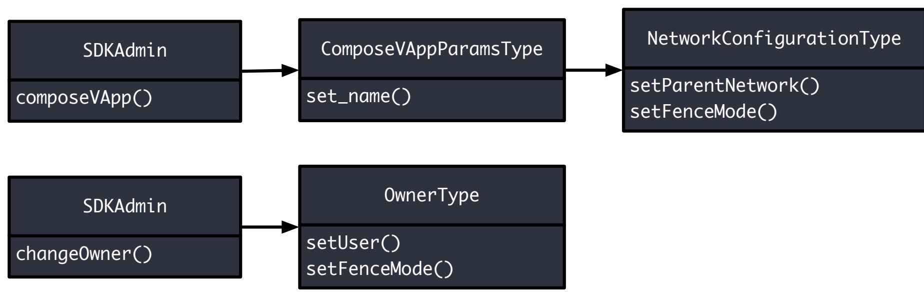
Proces vyžaduje odkaz na verejnú sieť. Verejná sieť je dostupná volaním SDK nad objektom *Extension* pomocou funkcie *getVMWExternalNetworks* s parametrom mena verejnej siete. Sieť sa konfiguruje prostredníctvom objektu *NetworkConfigurationType*, ktorého parameter rodičovskej siete odkazuje práve na verejnú sieť. Potrebné je nastavenie takzvaného *fence* módu. Nastavenia sa zabalia do objektu *OrgVdcNetworkType*, ktorému sa navyše priradí meno siete, ktorú vytvárame. Vytvorenie je sprostredkované funkciou *addvdcNetwork*.



Obr. 5.5: Schéma objektov pre vytvorenie siete VDC

5.2.6 Vytvorenie virtuálnej aplikácie

Pri vytváraní aplikácie sa **vytvorí kompozícia**, ktorú konfiguruje objekt *ComposeVAppParamsType*, ktorému je možné nastaviť meno novej vAPP. Požiadavka na vytvorenie kompozície sa odosiela cez funkciu *composeVApp*. Následne je možné upraviť vlastníka predaním parametra referencie na užívateľa funkcii SDK *changeOwner*.



Obr. 5.6: Schéma objektov pre vytvorenie virtuálnej aplikácie

5.2.7 Vytvorenie virtuálneho servera

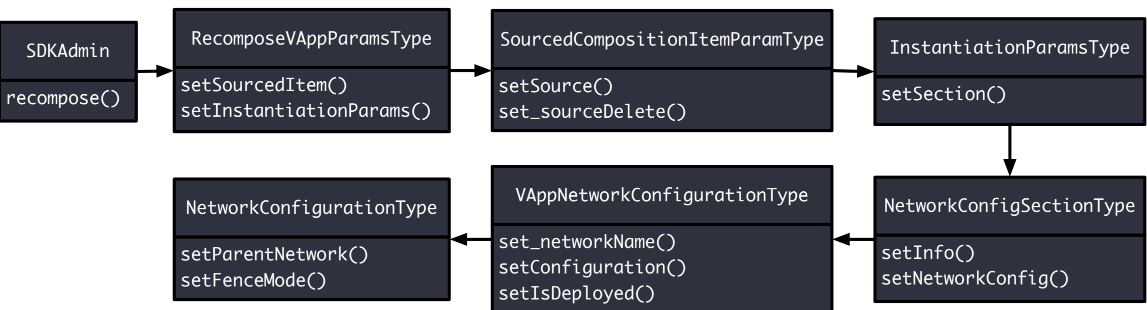
Nový virtuálny server sa dá vytvoriť pomocou objektu *šablóny*. **Zoznam šablón** je možné získať pomocou štandardnej funkcie nad objektom *Query*, tvoriacim výpis zoznamu jednotlivých objektov v systéme. Potrebné je definovanie typu objektu ako *vAppTemplate*. Metóda `getChildren()->getVm()` na výstupe vráti konkrétny virtuálny server, ktorý bude slúžiť ako predloha pre nový virtuálny server.

Nastavenie siete servera sa konfiguruje cez objekt *InstantiationParamsType*, ktorý nesie konfiguračnú sekciu *VMware_VCloud_API_NetworkConfigSectionType* s objektom typu *VMware_VCloud_API_VAppNetworkConfigurationType*, ktorému je možné nastaviť meno používanej siete spolu s vlastnosťou *FenceMode* a jej konfiguráciu objektom typu *VMware_VCloud_API_NetworkConfigurationType*. *SourcedCompositionItemParamType* je objekt, ktorý odkazuje na referenčnú šablónu klonovaného systému. Všetko zabalí do objektu *RecomposeVAppParamsType*. Rekonfigurácia sa volá nad objektom SDK vAPP funkciou *recompose*.

Úprava nastavení procesora a operačnej pamäte je dostupná volaním funkcií *getVirtualCpu* a *getVirtualMemory* nad SDK objektom entity VM. Po získaní objektu kvantity funkciou *getVirtualQuantity* a zmenou hodnoty parametra tohto objektu sa zmena vykoná volaním funkcií *modifyVirtualCpu* a *modifyVirtualMemory*. Pri nastavovaní úložiska je proces podobný ale zložitejší, pretože diskových jednotiek môže mať VM viacero. Nad objektom SDK entity VM sa zavolá funkcia *getVirtualDisks* a funkcia *getItem*, ktorá vráti konkrétnu jednotku disku. Funkcia *get_anyAttributes* slúži na získanie hodnôt disku ako aj kapacity. Tieto hodnoty je možné zmeniť a uložiť zavolaním funkcie *modifyVirtualDisks*.

Potrebné je upraviť parametre prihlasovania. Cez objekt *getGuestCustomizationSettings* je možné konfigurovať spôsob prihlasovania buď cez meno a heslo alebo cez server LDAP. Meniť je možné aj ďalšie parametre ako **počet chybných pokusov prihlásení** alebo **funkciu automatického prihlasovania do systému**. Po zmene hesla sa môže nastaviť meno počítača a následne funkcia *modifyGuestCustomizationSettings* uloží zmeny.

Poslednou fázou je vydanie nového VM objektom *DeployVAppParamsType*, ktorý **server zapne** a vykoná úpravy. Funkcia *deploy* nad SDK entitou VM vykoná akciu.



Obr. 5.7: Vytvorenie virtuálneho servera

5.3 Návrh automatizácie systému Proxmox VE

Konkrétne použitie Proxmox bude využívať *cluster* zložený z troch uzlov. Pre správnu korekciu prípadných chýb je potrebné, aby boli v prevádzke minimálne dva uzly. Je možné nastaviť rôzne *zóny* v ktorých sa budú nové servery ukladať. V prípade **migrácie** je potom možné server migrovať na iný bez výpadku pri použití zdieľaného úložiska. V prípade, *kontajnerov* je však potrebné počítať s krátkym výpadkom. Podporované súborové systémy sú *LVM, NFS, ZFS* a rôzne iné. Ako zdieľané úložisko je možné využiť *súborový systém NFS*. Zákazníkovi je možné prideliť prístup priamo do webového **rozhrania Proxmox.**

Pre automatizovanie serverov pomocou technológie a aplikácie Proxmox je potrebné využitie **REST API**[1] poskytované spoločnosťou Proxmox. Dostupné sú aj rôzne implementácie SDK, ktoré zjednodušujú komunikáciu, ale z hľadiska jednoduchosti API a prípadného budúceho škálovania je lepšie zvoliť vlastnú implementáciu.

5.3.1 Autentifikácia požiadaviek

Pre **zabezpečenie bezpečnej komunikácie** je potrebné najprv vytvoriť spojenie na prístupový bod, ktorý poskytuje takzvaný *ticket*. Ticket je entita, ktorá obsahuje *cookie*, s ktorým sa bude ďalej pristupovať k ďalším požiadavkám a kľúč pre zabránenie útoku *Cross-Site Request Forgery*[2]. Po získaní týchto parametrov je potrebné definovať SDK implementáciu, ktorá bude cookie spolu s tokenom zasielať na server pri každej požiadavke. Príklad získania autentifikačných údajov je popísaná cURL volaním nižšie.

```
curl --data "username=XXX&password=YYY"
 apiAddress/api2/json/access/ticket
```

5.3.2 Vytvorenie LXC

Servery Proxmox budú vytvárané pomocou **klonovania** pre možnosti rozšírenej konfigurácie po ich vytvorení. Schopnosť modifikácie dostupných obrazov nie je dostatočná, preto je potrebné vytvorenie vlastných šablón. Pri tvorbe novej inštancie klonovaním je definovaný **identifikátor** novej inštancie, ktorá bude vytvorená na výstupe scenára. Parameter *full*

[1] Dokumentácia Proxmox API https://pve.proxmox.com/pve-docs/api-viewer/
[2] Cross-Site Request Forgery útok https://www.veracode.com/security/csrf

umožňuje vytvorenie **hlbokej kópie** obrazu. Pri tomto spôsobe sa vytvorí kompletná kópia, ktorá je plne nezávislá na originálnom obraze. Požiadavkou však je aby výsledný server mal rovnakú veľkosť disku ako šablóna. Pri klonovaní cez **linkovanie** dochádza k úspore diskového priestoru, avšak nový server nemôže bežať bez prepojenia na pôvodný obraz. Server môžeme zaradiť do konkrétneho *poolu*, konfigurovať mu lokálne **úložisko** so súborovým systémom a **koncový server**, na ktorom bude po klonovaní vytvorený. V URI adrese koncového bodu je potrebné konkretizovať **zdrojovú inštanciu** a zdrojové **meno serveru**.

```
curl --cookie "$(<cookie)" --header "$(<csrftoken)" -X POST \
 -d newid=555 -d description=centosTest -d full=1 -d pool=1253 \
 -d storage=local-lvm -d target=d4422 \
 apiAddress/api2/json/nodes/d4422/lxc/101/clone
```

Konfigurovať je možné **počet jadier**, **veľkosť operačnej pamäte** a miesta **swap** na disku. V prípade LXC je potrebné definovať **nastavenia siete** pre pripojenie na internet a do lokálnej siete. Súčasťou koncového bodu je názov servera (napríklad d4422), na ktorom sa daná inštancia nachádza a identifikátor, pod ktorým je inštancia v systéme uložená.

```
curl --cookie "$(<cookie)" --header "$(<csrftoken)" -X PUT \
 -d cores=2 -d memory=2048 -d swap=1024 -d onboot=1 \
 -d net0=name=eth0,bridge=vmbr0,gw=194.145.183.1,ip=194.145.183.203/24 \
 apiAddress/api2/json/nodes/d4422/lxc/555/config
```

Zväčšenie disku je možné príkazom, ktorý pozostáva z názvu disku, ktorý chceme upravovať a hodnotou novej veľkosti (napríklad 45G). Koncový bod je podobne nastavený ako u konfigurácie parametrov. V prípade LXC bude upravovaný súborový systém *rootfs*.

```
curl --cookie "$(<cookie)" --header "$(<csrftoken)" -X PUT \
 -d disk=rootfs -d size=45G \
 storage=local-lvm apiAddress/api2/json/nodes/d4422/lxc/555/resize
```

5.3.3 Vytvorenie KVM

Najlepším spôsobom vytvorenia KVM je tiež **klonovanie**. Samotné volanie je veľmi podobné LXC ako aj nastavenie koncového bodu volania.

```
curl --cookie "$(<cookie)" --header "$(<csrftoken)" -X POST \
 -d newid=1 -d description=test1 -d full=1 -d pool=1253 -d \
 storage=local-lvm apiAddress/api2/json/nodes/d4422/qemu/100/clone
```

Editácia je tiež podobná ako u LXC. Rozdielom je, že pri konfigurovaní nie je zadávané nastavenie siete. Táto vlastnosť nie je dostupná, pretože sa jedná o iný typ virtualizačnej technológie. Konfigurácia bude prebiehať **inicializačným skriptom**, ktorý sa bude spúšťať pri štarte servera, ktorý bol naklonovaný zo šablóny.

```
curl --cookie "$(<cookie)" --header "$(<csrftoken)" -X POST \
 -d cores=2 -d memory=2048 -d onboot=1 \
 apiAddress/api2/json/nodes/d4422/qemu/555/config
```

Ak by bolo potrebné **rozšíriť disk**, je tak možné urobiť cez príkaz nižšie. V prípade KVM je potrebné ako názov diskovej jednotky zadať disk menom *scsi0* namiesto *rootfs*, ako to bolo u LXC. Po zväčšení disku a správnej inicializácii serveru bude dostupné pripojenie cez konzolu prostredníctvom pripojenia ssh.

```
curl --cookie "$(<cookie)" --header "$(<csrftoken)" -X PUT \
 -d disk=scsi0 -d size=45G \
 storage=local-lvm apiAddress/api2/json/nodes/d4422/qemu/555/resize
```

5.4 Fyzické servery

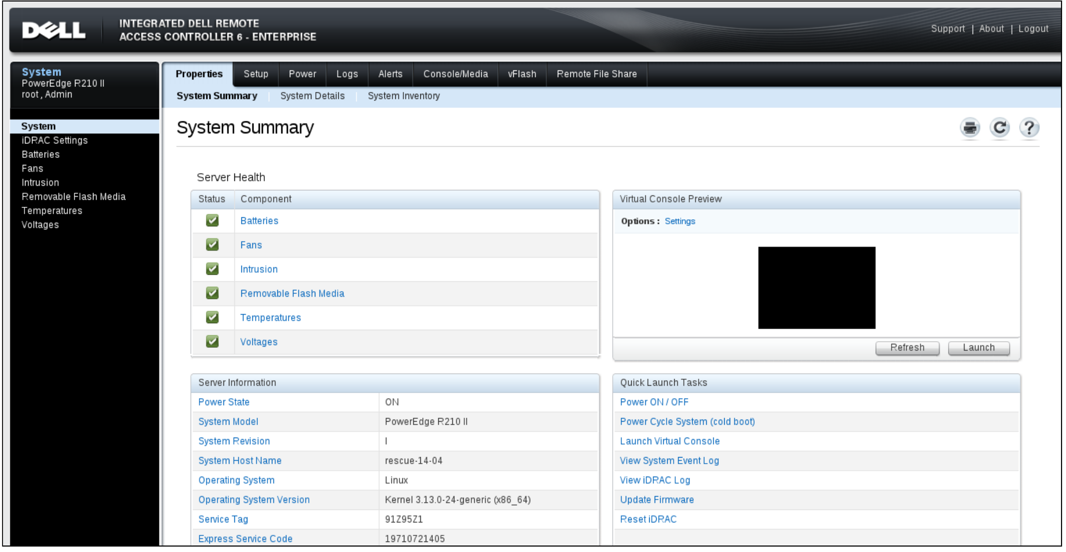
Fyzické servery **nie sú virtualizované**. Poskytovaný je priamo prístup na **fyzický server**. Server poskytuje svoj výkon na základe vlastnej **konfigurácie**. Na takomto serveri môže byť pomerne náročné zmeniť konfiguráciu. V prípade jej zmeny, môže prísť aj k strate užívateľských dát. Výhodou takýchto serverov je však ich vysoký výkon a plný prístup k administrácii celého servera.

5.4.1 Automatizácia predaja fyzických serverov

Fyzické servery poskytuje napríklad firma Dell. U väčších konfigurácií je dostupné rozhranie *iDrac*, ktoré umožňuje **správu servera** na vyššej úrovni. Dostupné je **migrovanie**, aktualizovanie, ale aj zapínanie servera a jeho vypínanie.

Automatizácie procesov je veľmi zložitá a nie je možné dodať zákazníkovi konfiguráciu v reálnom čase. Jednou z možností riešenia tohto problému je v prípade dostupnosti väčšieho počtu fyzických serverov pripraviť viacero **alternatívnych konfigurácií**, z ktorých sa bude pri vytváraní nového servera možné vyberať. Pomocou služieb administrácie ako je iDrac, je možné takýto pripravený server na diaľku zapnúť alebo vypnúť tak, aby len oprávnený klient získal prístup. **Prihlasovacie údaje je možné ukladať** v informačnom systéme.

Samotné scenáre nie je nutné popisovať. Komunikácia bude prebiehať čisto len na úrovni zasielania a prijímania dát zo servera GraphQL. Nie je potrebné interné vykonávanie skriptov alebo iných volaní priamo na fyzický server.



Obr. 5.8: Rozhranie iDrac

Kapitola 6

GraphQL SDK

Webová aplikácia umožňuje manipuláciu s užívateľskými dátami. Tieto zdroje poskytuje server, ktorý na komunikáciu využíva technológiu GraphQL popísanú v kapitole 3 a sekcii 3.11. Pri REST API sú požiadavky definované **prístupovým bodom**, na ktorý je požiadavka poslaná a jej obsahom. Výsledok má presne definovanú štruktúru. U GraphQL sú požiadavky aj štruktúra vrátených dát **variabilné**. Vďaka tejto vlastnosti, je možné komunikáciu prispôsobovať požiadavkám konkrétnej aplikácie. Na komunikáciu so serverom GraphQL bola v tomto projekte použitá **vlastná implementácie knižnice**, ktorá generuje požiadavky a spracováva prijímané dáta. Kapitola podrobne popisuje túto knižnicu po implementačnej a funkčnej stránke. Podrobne je rozobraná aj architektúra riešenia.

6.1 Architektúra knižnice

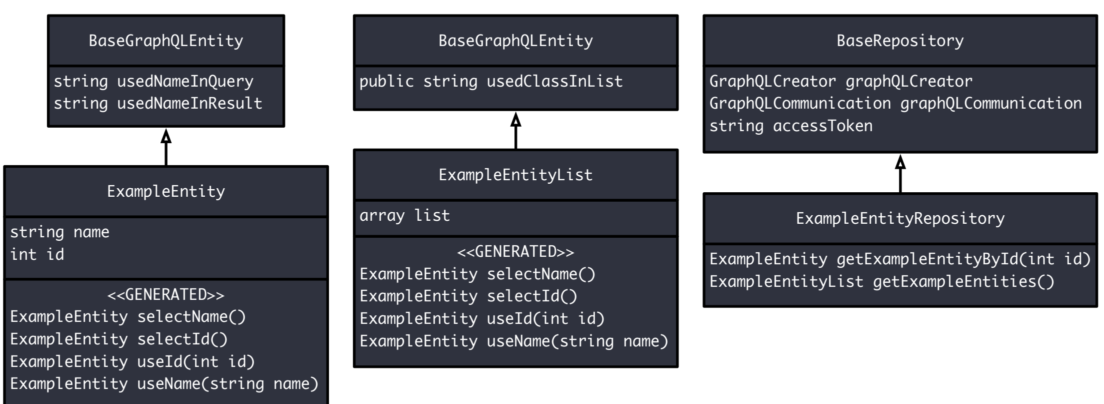
Architektúra je zložená z viacerých tried, ktoré poskytujú rôznu funkcionalitu. Všetky tieto triedy sú implementované ako **rozšírenie pre framework Nette**[1] popísaný v sekcii 3.2 pomocou *dependency injection* a *kontajnera DI* opísaného v sekcii 3.2.3. Toto rozšírenie je možné jednoducho pridať do existujúceho projektu a konfigurovať ho prostredníctvom *kontajnera*. Dostupné je nastavenie prístupového bodu servera, na ktorý sú požiadavky zasielané. Je potrebné definovať názov adresára, kam sa bude komunikácia a jej záznamy ukladať. V prípade, že je požadované zasielanie chybových hlásení do služby **Mattermost**[2], je nutné definovať *hook*, na ktorý sa zasielajú správy. Pomocou parametra *timeout* je možné zvoliť časovú odozvu, po ktorú bude komunikácia otvorená a čakať na odpoveď. Možné je aj **nastavenie prostredia** buď to ako produkčného, testovacieho alebo vývojového.

Samotná knižnica je rozdelená na triedu, ktorá **generuje požiadavky** zaslané ako žiadosť na server, triedu, ktorá **odosiela požiadavky** a ukladá komunikáciu do záznamových súborov, triedu, ktorá **spracováva odpovede servera** a prevádza ich na objekty DTO a pomocné triedy, ktoré definujú **nové dátové typy** využívané v knižnici alebo **rozhrania a rodičovské triedy**, z ktorých dedia objekty využívané ďalej v časti projektu. Dostupné sú funkcie na komunikáciu so službou Mattermost.

[1] Tvorba rozšírení Nette `https://doc.nette.org/cs/2.4/di-extensions`
[2] Tvorba hooks v Mattermost `https://docs.mattermost.com/developer/webhooks-incoming.html`

6.2 Použitie SDK vo webovej aplikácii

Aplikácia využíva tri druhy tried nachádzajúcich sa vo **vrstve modelu projektu**. *Entity* priamo **mapujú objekty** definované v schéme GraphQL servera, doplnené sú o entity definujúce **kolekcie** týchto objektov. Každá entita predstavuje DTO, ktoré obsahuje dva druhy metód a parametrov. Prvý druh s prefixom *use* umožňuje definovanie **filtračných požiadaviek**, druhý druh s prefixom *select* definuje **štruktúru vrátených dát** serverom. Entity kolekcie obsahujú len jediný parameter s názvom *list* typu pole, ktorý je plnený kolekciou objektov požadovaného typu. *Repozitáre* definujú samotné **skladanie objektov**, tvorbu výsledného query, zaslanie žiadosti na server a spracovanie dát z odpovede servera. Metódy repozitárov mapujú základné *queries* a *mutations* a ich vstupné parametre a výstupné objekty. Príklad jednotlivých tried zobrazuje obrázok 6.1.



Obr. 6.1: Príklad entity, entity kolekcie a repozitára

Každá entita je potomkom rodičovskej triedy **BaseGraphQLEntity**, ktorá implementuje základné operácie. V prípade vytvorenia objektu takejto triedy dochádza k inicializácii všetkých *set* a *get* metód pomocou funkcie *__call* pre nastavovanie a získavanie dát z objektu. Všetky parametre objektu sú definované a inicializované cez funkciu *initProps*. Implementuje aj metódy, ktoré nastavia parametre využívané pri komunikácii, napríklad meno požiadavky a meno výsledného objektu na výstupe servera. Pomocná funkcia *cleanupResultObject* umožňuje filtrovanie objektu po naplnení výslednými dátami od nepotrebných inštančných premenných. Funkcia *getResultAsArray* vráti výsledný objekt ako slovník.

Trieda **GraphQLCommunication** obsahuje funkcie, ktoré slúžia na komunikáciu so serverom GraphQL. Implementuje autorizáciu požiadaviek a ich zasielanie na server. Kontroluje výstupné dáta zo servera a vracia výnimky podľa ich povahy. Súčasťou je podpora ukladania a mazania výsledkov v pamäti servera *Redis* popísaného v sekcii 3.7. Vďaka implementácii všetkých týchto funkcií sú ďalšie volania v *repozitároch* abstrahované.

Objekty rodičovskej triedy **GraphQLDates** definujú dátový typ času, ktorý je používaný pri tvorbe požiadaviek a spracovaní dát. Implementujú rozširujúce metódy nad rodičovským objektom *Datetime*.

6.3 Implementácia tried repozitárov

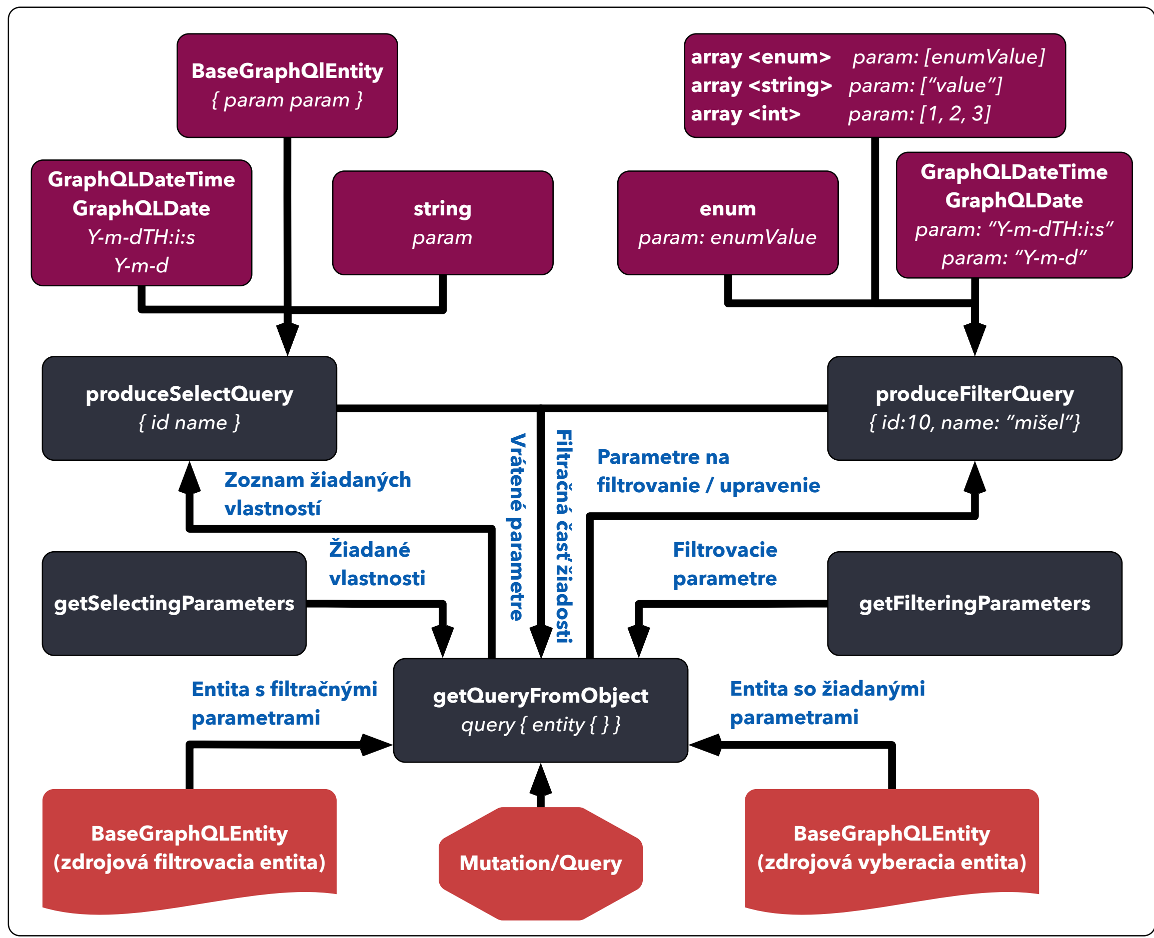
Repozitáre **spájajú celý proces** generovania, komunikácie a získavania odpovedí do jediného celku. Všetky triedy sú potomkami triedy *BaseRepository*. Táto trieda obsahuje metódy na nastavenie **autentifikácie** používateľa. Samotný repozitár mapuje funkcie definované v schéme GraphQL. Príklad metódy zobrazuje blok kódu 6.2. Na začiatku je potrebné **definovanie jednotlivých entít** použitých v požiadavke a odpovedi. Tieto entity je potrebné hierarchicky poskladať. Následne sa zavolá **funkcia na generovanie požiadavky** a jej odoslanie na server. Možné je nastaviť ukladanie do dočasnej pamäte servera *Redis* spolu s exspiráciou a zneplatnením iných dát. Po získaní odpovede zo servera sa **inicializuje nový objekt**. Tento objekt je naplnený a vrátený na výstupe metódy. Vďaka vysokej abstrakcii funkcionality je tento proces veľmi jednoduchý a obsahuje málo redundancie, čím prispieva k výraznému zvýšeniu čistoty kódu.

```
public function getServicesByName(string $name) : ServicesList {

    $select = new ServiceBreakerDto();
    $select->selectName()->selectId();
    $filter = new ServiceBreakerDto();
    $filter->useName($name);
    $request = $this->graphQLCreator->getQueryFromObject($select,$filter);
    $result = $this->communicator->sendRequestGraphQuery($request,'POST', 28800);
    $outList = new ServicesList();
    $outList->constructCollectionFromInputArray($result,ServicesDto::class);
    return $outList;
}
```

Obr. 6.2: Príklad implementácie metódy repozitára

6.4 Generovanie požiadaviek

Generovanie zabezpečuje objekt triedy **GraphQLCreator**. Vstupom do inicializačnej funkcie sú entity a príznak, ktorý rozhoduje o type požiadavky. Podporované sú *mutations* aj *query* popísané v sekcii 3.11. V prípade *mutations* sa filtrovacie parametre zmenia na tie, ktoré nesú **modifikačné dáta**. Pred generovaním požiadavky *query* je potrebné z predaných entít získať **zoznam filtračných parametrov**. Pri oboch typoch požiadaviek sa získa zoznam parametrov, ktoré budú definovať **štruktúru dát** na výstupe servera. Na základe týchto zoznamov sa prehľadajú jednotlivé parametre a podľa ich typu sú dosadzované do výslednej požiadavky. V prípade filtračných parametrov je možné predať parametre **rôznych dátových typov** ako *enum* alebo *GraphQLDates*. V prípade parametrov definujúcich štruktúru výstupných dát, je možné definovať parametre bez použitia zoznamov. Opäť je možné predať čas cez dátové typy dediace od triedy *GraphQLDates*, jednoduchý reťazec alebo inú entitu, ktorá vyvolá rekurzívne spracovanie. Tento proces nie je možno generalizovať, pretože pri každom type je výstup definovaný inak. Na konci celého procesu sa poskladajú jednotlivé časti a vrátený je **výsledný reťazec požiadavky**, ktorá sa po zakódovaní do JSON formátu môže odoslať na server. Proces generovania žiadosti vizuálne zobrazuje obrázok 6.3.

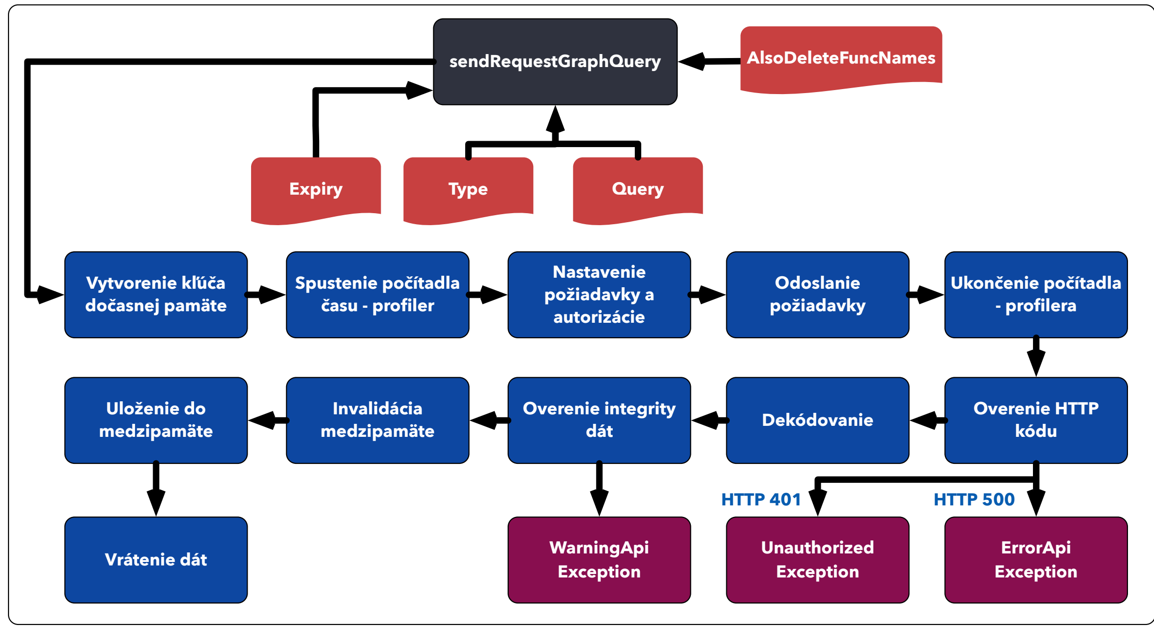


Obr. 6.3: Schéma procesu tvorby požiadavky

6.5 Zasielanie žiadostí na server

Po vytvorení požiadavky je potrebné jej zaslanie na GraphQL server. Všetku funkcionalitu spojenú s odosielaním a prijímaním dát spravuje trieda **GraphQLCommunicator**. Stará sa aj o správu **dočasného úložiska** na serveri *Redis* popísanom v sekcii 3.7. Vstupnú žiadosť enkóduje do formátu JSON a nastaví **autorizačné hlavičky**. Pred odoslaním inicializuje kľúč, pod ktorým budú dáta dostupné v pamäti v prípade opätovného volania. Kľúč sa generuje z identifikačného reťazca klienta spolu s obsahom žiadosti a umiestnením aplikácie pre zachovanie jedinečnosti. Výsledný tvar je pre bezpečnosť zakódovaný *SHA512 hashovacím algoritmom*[3]. V prípade, že je aplikácia spustená vo vývojovom režime, prebehne inicializácia **počítadla času** odozvy servera. Po prijatí dát zo servera, je skontrolovaný *HTTP kód*[4]. V prípade, že bola vrátená chyba je tento incident uložený a nahlásený aj do služby **Mattermost**. Po uložení do dočasnej pamäte a ukončení počítadla času je na výstupe vrátený **dekódovaný výsledok** alebo príslušná výnimka podľa podstaty chyby. Proces názornejšie ilustruje obrázok 6.4.

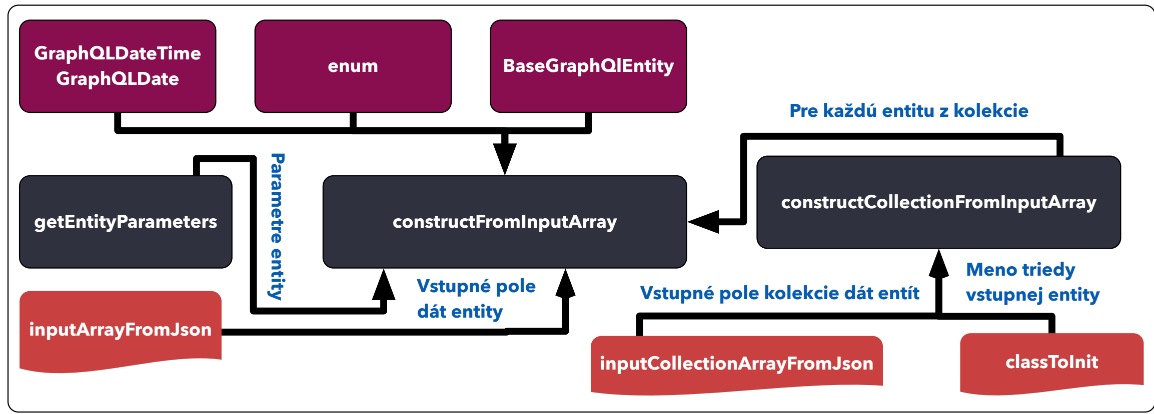
[3]SHA hashovací algoritmus http://www.quadibloc.com/crypto/mi060501.htm
[4]Zoznam HTTP kódov https://www.restapitutorial.com/httpstatuscodes.html



Obr. 6.4: Schéma procesu komunikácie so serverom

6.6 Spracovanie odpovede servera

Namiesto vytvorenia poľa z dekódovaných výstupných dát systém používa prepracovanejší systém. **Generované sú priamo entity**, ktoré sa podieľali na tvorbe požiadavky. Funkcia je inšpirovaná ORM systémami popísanými v sekcii 3.8. Výhodou je prehľadnosť a presná štruktúra očakávaných dát. Vývoj v editore ponúka možnosť automatického dopĺňania názvov parametrov. Systém na vstupe môže prijímať dva druhy požiadaviek. Buď je vrátená informácia o **jednej entite** alebo je vrátená **kolekcia týchto entít**. V prípade kolekcie sa jedná o prechod kolekcie týchto entít a volanie rovnakej funkcie ako pri získaní jednej entity. Táto entita je generovaná pomocou pomocných parametrov, ktoré sa nastavujú pri tvorbe entity a určujú aký typ objektu bude v kolekcii alebo v odpovedi. Na základe tohto typu systém spracováva vstupné dáta a napĺňa parametre entít podľa definovaných dátových typov. **Výsledok je vrátený ako entita alebo entita kolekcie**.



Obr. 6.5: Schéma procesu spracovania odpovede servera

Kapitola 7

Webová aplikácia

Súčasťou práce je návrh a implementácia webovej aplikácie, ktorá by umožňovala správu užívateľských dát a služieb. Na začiatku kapitola **špecifikuje požiadavky**, ktoré by mal systém implementovať. Popísaný je **návrh užívateľského rozhrania** a architektúry systému. Detailne je rozobraný spôsob zasielania **asynchrónnych požiadaviek** a oneskoreného načítavania obsahu stránok. Záver kapitoly je venovaný **automatickému testovaniu** a procesu nasadzovania a integrácie systému na rôzne prostredia v ktorých aplikácia beží.

7.1 Špecifikácia požiadaviek

Pred návrhom je potrebné špecifikovať požiadavky, ktoré by mali byť vo výsledku splnené. Na základe týchto požiadaviek je následne možné definovať problémy, ktoré bude potrebné riešiť na úrovni užívateľského rozhrania a výslednej implementácie. Zoznam nižšie popisuje konkrétne problémy vyplývajúce zo zadania.

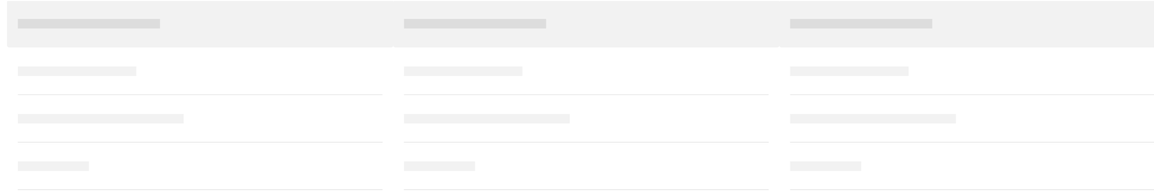
- Zobrazenie **základného pohľadu** pre používateľa, kde by mohol získať informácie o celom systéme. Pri návrhu bude najlepšie vychádzať z dashboardu použitom v systéme popísanom v sekcii 2.2. Dostupné by mali byť položky nezaplatených faktúr, aktuálne riešených problémov s podporou a zoznam požiadaviek čakajúcich na spracovanie. Vhodné by bolo implementovanie bloku s aktuálnymi správami.
- Poskytnutie informácií o všetkých **službách** zakúpených poskytovateľom vo forme stromu a kategorizovaných sekcií.
- **Asynchrónny návrh systému** tak, aby sa dáta získavali až po zobrazení celého obsahu stránky. Potrebné je vyriešiť grafiku pre zobrazenie stavu načítania položiek.
- Implementácia **grafov a štatistík** u služieb, ktoré poskytujú tieto informácie. Možnosť voľby časových okien u grafov. Filtrovanie a stránkovanie položiek štatistík tam, kde by ich počet mohol spôsobovať neprehľadné zobrazenie.
- **Lokalizovanie systému** do aspoň dvoch jazykových variácií bez nutnosti odhlásenia.
- Umožniť **konfiguráciu služieb**, ktoré to podporujú v samotnom GraphQL API. Možnosť rozšíriť tieto možnosti na automatický systém implementovaný touto prácou.
- Implementovať **zobrazenie informácií**, možnosti konfigurácie, dostupné rozširujúce operácie, grafy a tabuľkové štatistiky. Vytvoriť zobrazenia základných parametrov pre všetky **služby** popísané v prílohe D.

7.2 Návrh užívateľského rozhrania

Grafický návrh využíva koncept jednoduchosti a *flat designu*[1]. Design stránky je rozdelený do logických celkov podľa ich funkcie. **Horná lišta** je tvorená správou základných užívateľských dát ako sú faktúry, tikety alebo správy. Pravá časť lišty slúži na správu dát užívateľa a po rozbalení umožňuje meniť nastavenia profilu, odhlásiť sa alebo spravovať platobné informácie. **Ľavé menu** obsahuje zoznam všetkých služieb. Pre prehľadnosť sú zaradené do kategórií podľa ich typu. Možné je zobrazenie kompletného zoznamu všetkých služieb v stromovej štruktúre (viď. príloha B.7). **Zvyšnú časť** stránky tvoria zobrazované dáta.

7.2.1 Oneskorené načítanie dát

Zobrazenie stránok musí byť čo najrýchlejšie. Dáta je potrebné načítavať oneskorene po načítaní základnej štruktúry stránky. Takýto spôsob načítavania budí dojem lepšej odozvy systému. Rozhranie je potrebné navrhnúť inak ako v prípade klasických systémov. Dôležité je zobrazenie prvkov, ktoré sú schopné **indikovať načítavanie asynchrónnej operácie**. Musí byť vyplnený priestor na stránke, ktorý v momente načítavania **nezobrazuje žiadne informácie**. Stav načítavania sa zobrazí prostredníctvom prvku, ktorý je animovaný a nachádza sa v časti stránky, ktorá zobrazuje dáta. Prázdne miesto bude vyplnené elementom, ktorý pripomína buď zobrazenie grafu alebo tabuľky. Element bude vykreslený v neutrálnych farbách. Príklad vidno na obrázku 7.1.



Obr. 7.1: Návrh prvku zobrazujúceho načítavanie obsahu v prípade tabuľky

7.2.2 Responzívne zobrazenie

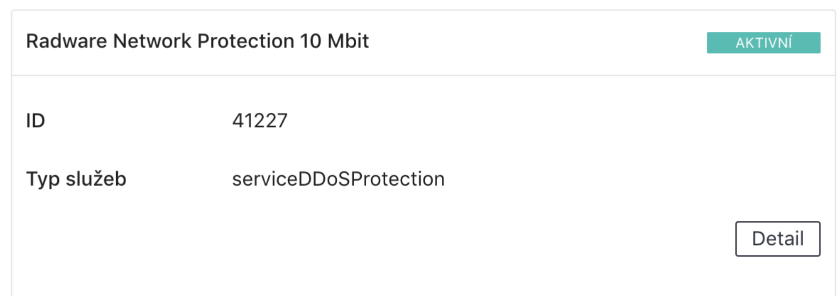
Dnešným trendom je zobrazovanie webových stránok na mobilných zariadeniach. Automatické prispôsobenie v prípade menších obrazoviek nie je dostačujúce a je potrebné navrhnúť a implementovať mobilný design. Väčšina informácií naprieč systémom sa zobrazuje vo forme **tabuliek**. Hodnoty, ktoré udávajú stav položiek sú vykreslené ako farebné **štítky** tak, aby podľa farieb užívateľ dokázal rozpoznať prípadný problém. Položke je možné priradiť **tlačidlo**, ktoré vykoná potrebnú akciu. Príklad tabuľky vidno na obrázku 7.2.

Název služby	**ID**	**Typ služeb**	**Stav**	**Akce**
Radware Network Protection 10 Mbit	41227	serviceDDoSProtection	AKTIVNÍ	Detail
SSL certifikát - *.safesignatures.com	55248	serviceSsl	AKTIVNÍ	Detail

Obr. 7.2: Zobrazenie informácií o službe na väčšej obrazovke

[1] Popis konceptu flat designu https://www.interaction-design.org/literature/topics/flat-design

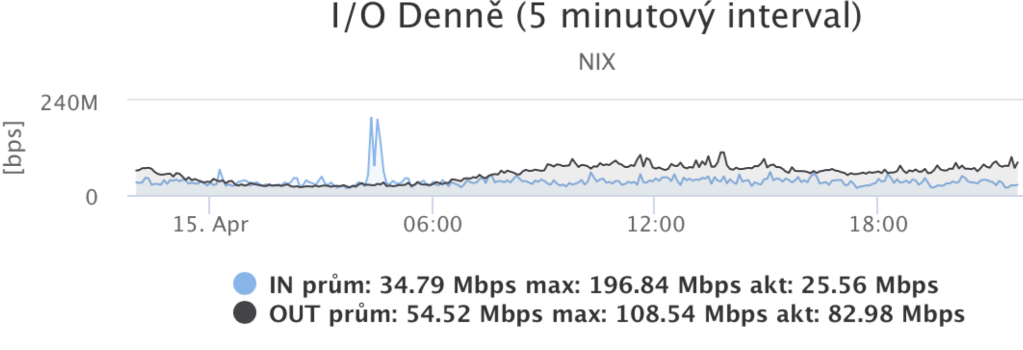
V prípade menších obrazoviek by mohol nastať problém, keď by sa do zobrazenej tabuľky nezmestili všetky informácie. Riešením by bolo buď zmeniť veľkosť písma a elementov alebo odstránenie niektorých stĺpcov tabuľky tak, aby zobrazené informácie nepresahovali veľkosť zobrazenia. Ani jeden z týchto spôsobov však nie je plne vyhovujúci. Systém má zabudované riešenie, kedy sa tabuľky v kóde vytvárajú abstraktne. V prípade, že webová aplikácia zistí, že sa zobrazuje na menšom displeji, **zmení zobrazenie tabuľky na blokové**. Blokové vykresľovanie nezobrazuje informácie v stĺpcoch tabuľky ale pod seba. Tento spôsob zabezpečí responzívnosť pri zobrazení ľubovoľného počtu položiek. Výslednú transformáciu vidno na obrázku 7.3.



Obr. 7.3: Zobrazenie informácií o službe na mobiloch

7.2.3 Zobrazenie grafov

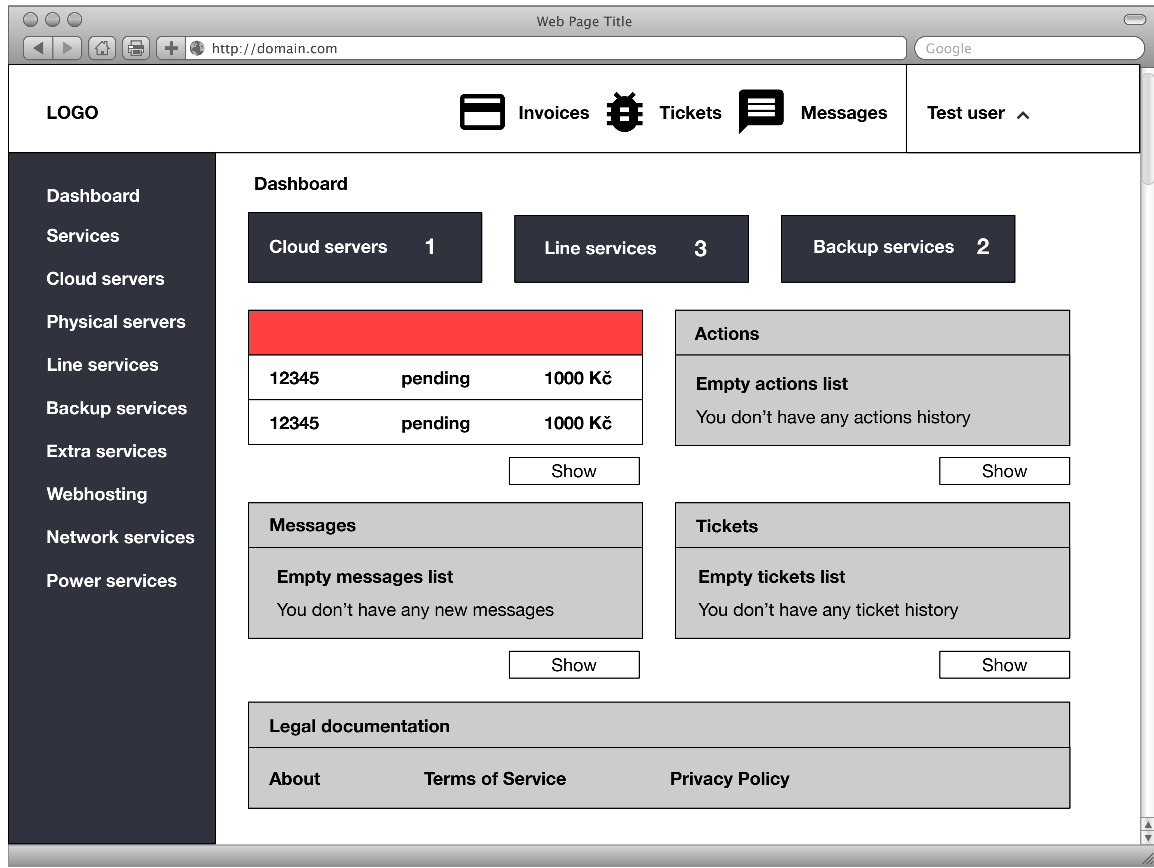
Zobrazenie grafov je implementované cez knižnicu *HighCharts* popísanú v sekcii 3.9, ktorá rieši problematiku od spracovania vstupných hodnôt až po výsledné zobrazenie a jeho štylizovanie. Na zobrazenie štatistík jednotlivých služieb je najlepšie použiť **klasický graf s 2 osami**, pričom jedna bude zobrazovať časový rozmer a druhá hodnotu parametru podobne ako u služby *Digital Ocean* preberanej v sekcii 2.1. Problém môže vzniknúť v prípade, že je načítané veľké množstvo dát, čo znemožní prehľadné zobrazenie. Na riešenie tohto problému ponúka knižnica **možnosť zoom**, ktorá urobí výrez časového okna a detailnejšie zobrazí informácie. Pretože v niektorých grafoch sa môže zobrazovať viacero parametrov na jednej časovej osi, je súčasťou návrhu možnosť **skrývať jednotlivé parametre** zobrazované v grafe. Knižnica automaticky počíta **konverzie jednotiek** na vyššie, aby neboli zobrazované príliš veľké čísla. Pod každým grafom by malo byť možné vidieť aktuálnu, priemernú a maximálnu hodnotu, aby zákazník nemusel tieto hodnoty manuálne odčítavať z grafu.



Obr. 7.4: Zobrazenie grafu štatistík služby

7.2.4 Dashboard

Pohľad dashboardu je inšpirovaný systémom Rackspace popisovanom v sekcii 2.2. Zobrazený je hneď po úspešnom prihlásení užívateľa do systému. Poskytuje základné informácie o službách a prihlásenom používateľovi. Pomocou jednoduchých blokov v hornej časti stránky je užívateľ informovaný o **aktívnych službách** na jeho účte. Zobrazované sú ich kategórie a počet. Po kliknutí na požadovaný blok kategórie prebehne presmerovanie na detail konkrétnej služby. Pod zobrazenými kategóriami dostupných služieb sú zobrazované tabuľky. Prvý tabuľkový blok zobrazuje **faktúry**, ktoré momentálne čakajú na zaplatenie alebo sú po lehote splatnosti. Ak existujú takéto faktúry, zobrazí sa tabuľka v červenom bloku. Ak má zákazník všetky faktúry uhradené, zobrazí sa informácia, že nie je potrebné venovať tejto sekcii pozornosť. V prípade tabuľky akcií sa zobrazujú **udalosti**, ktoré čakajú na schválenie obchodníkom alebo automatickým systémom spracovávania úloh. V prípade, že nie je zoznam udalostí prázdny, zobrazený je ich stav, čas vytvorenia a identifikácia. Tabuľka správ obsahuje **aktuality a správy** zaslané marketingovým oddelením spoločnosti. Posledná tabuľka zobrazuje zoznam **aktuálne riešených problémov**. Pri všetkých tabuľkách je možné pomocou tlačidlá prejsť na konkrétny zoznam všetkých položiek. Spodná časť stránky sprístupňuje odkazy na **dokumenty**. Načítavanie všetkých položiek dashboardu je indikované rovnako ako v sekcii 7.2.2.



Obr. 7.5: Návrh užívateľského rozhrania dashboardu

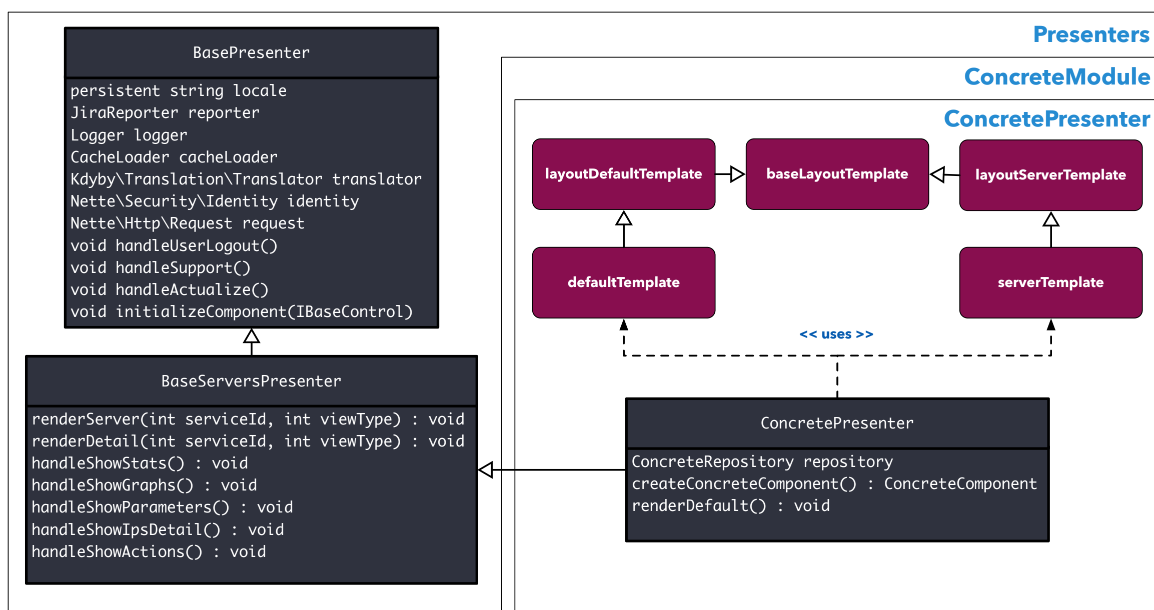
7.3 Objektová architektúra

Pri implementácii dochádza častokrát k problémom, ktoré bránia ďalšiemu **rozširovaniu** zdrojového kódu. Väčšina týchto problémov je spôsobená nesprávnym návrhom architektúry aplikácie. Jedným z najpoužívanejších návrhových vzorov využívaných pri návrhu architektúry aplikácie je *MVP*, rozdeľujúci aplikáciu na samostatné logické vrstvy, ktoré medzi sebou navzájom komunikujú. Detailný pohľad na túto techniku popisuje sekcia 3.2.1.

Systém využíva rozdelenie jednotlivých prvkov podobné technológii *React*[2]. Všetky hlavné entity stránok sú spojené do *modulov*. **Moduly** obsahujú ľubovoľný počet stránok - *presenterov*. Tie obsahujú menšie logické prvky - *komponenty*. Každá **komponenta** disponuje vlastnou šablónou, JavaScript podporou a štýlom. Vďaka tomuto návrhu je možné jednotlivé komponenty kedykoľvek znovu použiť. Všetka logika spracovania požiadaviek a získavania dát z modelov je obsiahnutá v komponentoch a tak nedochádza k prílišnému naťahovaniu rozsahu tried samotného presentera.

7.3.1 Vrstva presenterov

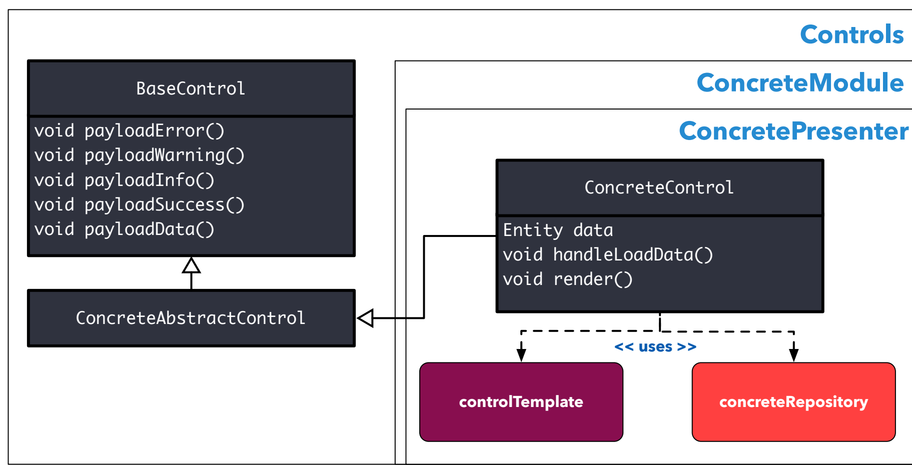
Každý **presenter** je potomkom rodičovskej triedy *BaseServersPresenter* alebo inej, ktorá poskytuje základné funkcie pre daný typ služieb. **Rodičovská trieda** dedí od abstraktnej triedy *BasePresenter*, ktorá uchováva objekty služieb, využívané v presenteroch. Implementuje akcie na udalosti, ktoré môžu byť volané zo všetkých zobrazení aplikácie. Namiesto implementácie celej riadiacej logiky stránky v jednej triede presentera, obsahujú tieto triedy len inicializačné metódy komponent, ktoré obsahujú. Presenter je závislí na šablóne, ktorá sa použije pri výslednom zobrazení stránky. **Šablóny** sú navrhnuté tak, aby boli hierarchicky pospájané a zabránili redundancii HTML kódu. **Layout rodičovskej šablóny** obsahuje definície používaných JavaScript a CSS súborov. **BaseLayout** šablóna definuje kostru celého webu a závislosť na zdrojoch používaných naprieč celou webovou aplikáciou.



Obr. 7.6: Architektúra vrstvy presentera

[2]Stránka projektu React https://reactjs.org

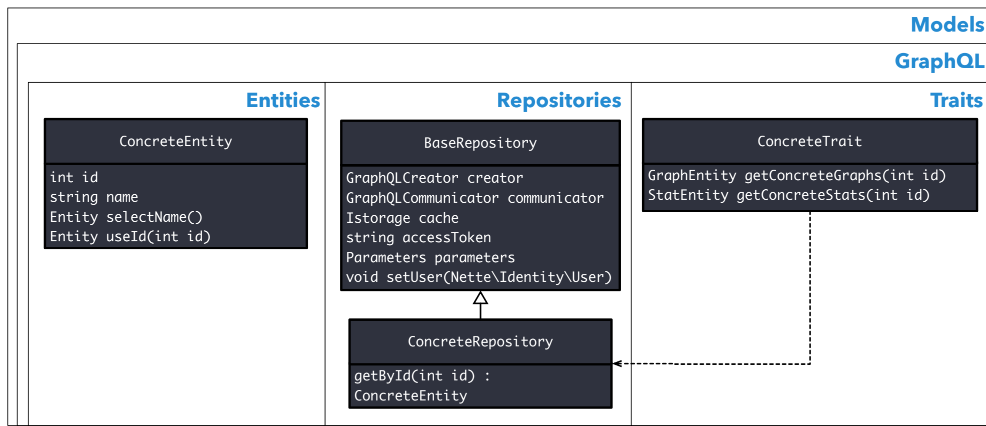
Komponenty sú súčasťou vrstvy presentera pre konkrétnu logickú časť stránky. Predstavujú modulárny systém. Každý komponent je potomkom rodičovskej *BaseControl*, ktorá obsahuje funkcie typické pre všetky komponenty. Rodičovská trieda implementuje metódy, ktoré definujú štruktúru odpovedí servera (*payloadError*). Komponenty vlastnia premennú dátového typu entity, ktorej informácie sa vo výsledku zobrazujú, metódy vykreslenia šablóny a akcie, ktoré implementujú odpovede na jednotlivé signály zasielané aplikáciou. Implementujú metódy pre asynchrónne načítanie dát (*handleLoadData*).



Obr. 7.7: Architektúra vrstvy komponentov

7.3.2 Implementácia modelov

Väčšina komponentov je závislá na **modeloch**. Základnou entitou reprezentujúcou model je *repository*, vid. sekciu 6.2. Je potomkom *BaseRepository*, ktorá implementuje funkcie na nastavenie základných parametrov pre komunikáciu s GraphQL serverom. Repozitáre sú závislé na **entitách**, ktoré sa skladajú za účelom tvorby požiadavky. Niektoré repozitáre je potrebné navrhnúť abstraktne s možnosťou ich rozširovania o ďalšie vlastnosti. Riešením takéhoto objektového prístupu sú **traity**. Predstavujú prvok označovaný aj *extension*. Umožnia do objektu repozitára vložiť všetky metódy implementované v traite. Možné je vytvoriť závislosť na viacero traitách. Využívané sú pri štatistikách a grafoch.



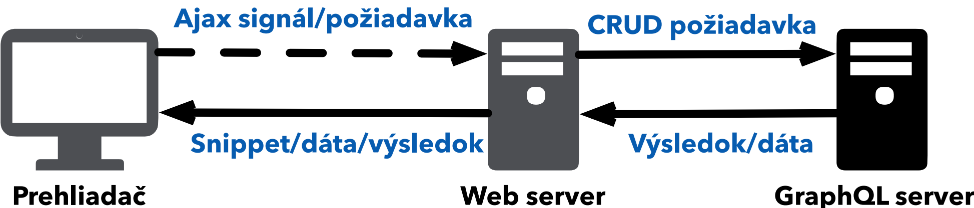
Obr. 7.8: Architektúra vrstvy modelov

7.4 Implementácia asynchrónnosti

Trendom pri tvorbe informačných systémov je prechod od tradičného konceptu ku chovaniu podobnému **aplikáciám na chytrých telefónoch**. Takéto riešenia sa chovajú responzívne a namiesto vykonávania celkových prechodov medzi stránkami, dochádza len k **obnovovaniu jednotlivých blokov**. Na dosiahnutie princípu je potrebné využiť technológiu *Ajax*[3].

Akákoľvek stránka po načítaní zobrazuje len **základnú štruktúru** spolu s prvkom, ktorý oznamuje priebeh načítavania. V tomto momente sa na stránke nevykresľujú žiadne dáta, ktoré by bolo potrebné získavať z GraphQL API. Tieto dáta sú požadované **asynchrónne** až po jej inicializácii podľa potreby. Výhodou takéhoto systému je, že samotný prechod na novú stránku má veľmi **malú odozvu**.

Po inicializácii DOM stromu sa zavolá **funkcia na prekreslenie** časti stránky. Túto požiadavku spracuje *TypeScript* modul. Vytvorí sa nové Ajax **volanie na webový server**. Ten na signál reaguje zaslaním **GraphQL požiadavky** a získa dáta z GraphQL servera. GraphQL server zašle dáta webovému serveru, ktorý doplní *snippety*[4] šablóny, ktorú zašle klientskemu prehliadaču. Vykoná **nahradenie DOM** prvkov a zobrazí výsledok. V prípade že sa jedná o **signál** ktorý bude modifikovať dáta alebo vykonávať inú operáciu, je priebeh rovnaký. Rozdiel je vo formáte zasielaných a prijímaných dát.



Obr. 7.9: Diagram zasielania Ajax požiadaviek

7.4.1 Modul responzívnosti

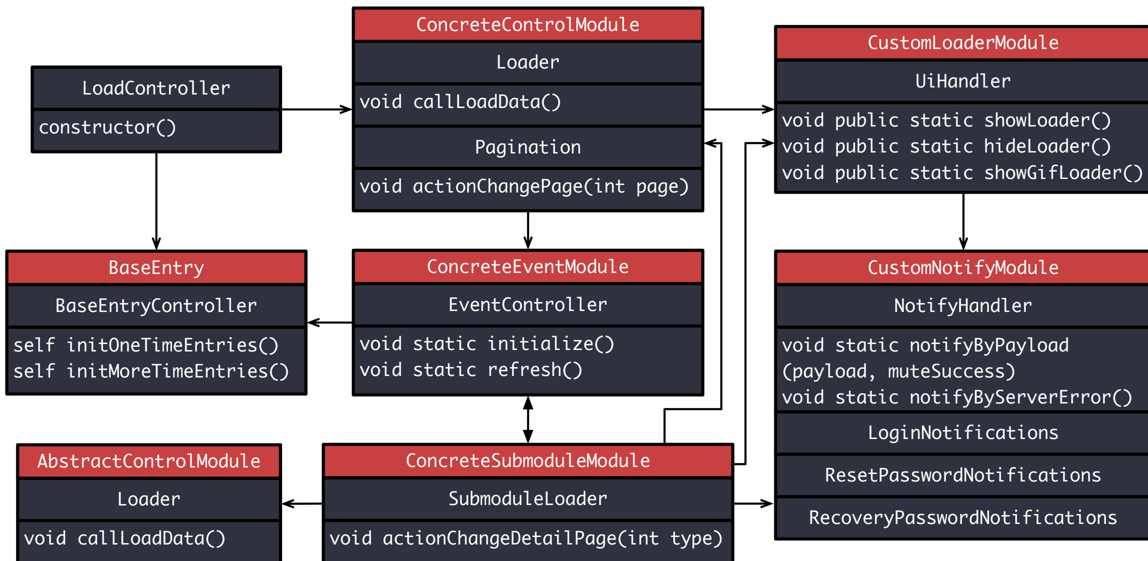
Aby mohla responzívnosť stránky fungovať, je nad aplikáciou postavená nadstavba napísaná v jazyku **TypeScript** popísanom v sekcii 3.4. Táto vrstva sa stará o spracovávanie požiadaviek stránky a úpravy DOM stromu. Po načítaní stránky sa vytvorí objekt triedy *LoadController*. Obsahuje len *konštruktor*, ktorý inicializuje závislosti, pripojí udalosti k jednotlivým prvkov stránky a zavolá prvú inicializáciu, ktorá vykreslí dáta.

Niektoré stránky obsahujú **podstránky** pre štatistiky, grafy alebo akcie. Je potrebné správne odpovedať na signály pri prechode medzi nimi a prekresľovať zobrazenie. O funkcionalitu sa stará trieda *SubmoduleLoader*. V prípade že je jedna z podstránok abstraktná z dôvodu, že sa vyskytuje na viacerých stránkach, trieda *SubmoduleLoader* inicializuje takéto prvky z abstraktných závislostí. O **komponenty stránok** sa starajú moduly *Control* a triedy typu *Loader* a *Pagination*. **Trieda Loader** definuje funkcie, ktoré vytvárajú Ajax volania. **Trieda Pagination** sa stará o implementáciu stránkovania.

Pri prekreslení stránky sa deaktivujú **udalosti** naviazané na DOM strom, ktorý bol nahradený. O obnovenie udalostí sa postará modul *Event* a trieda *EventController*. Funkcia *initialize* sa používa pri prvom načítaní stránky a *refresh* pri každom prekreslení stránky. Všetky volania využívajú generalizované triedy modulu *CustomLoader* a *CustomNotify*,

[3]Technológia Ajax `https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started`
[4]Nette snippety `https://doc.nette.org/cs/3.0/ajax`

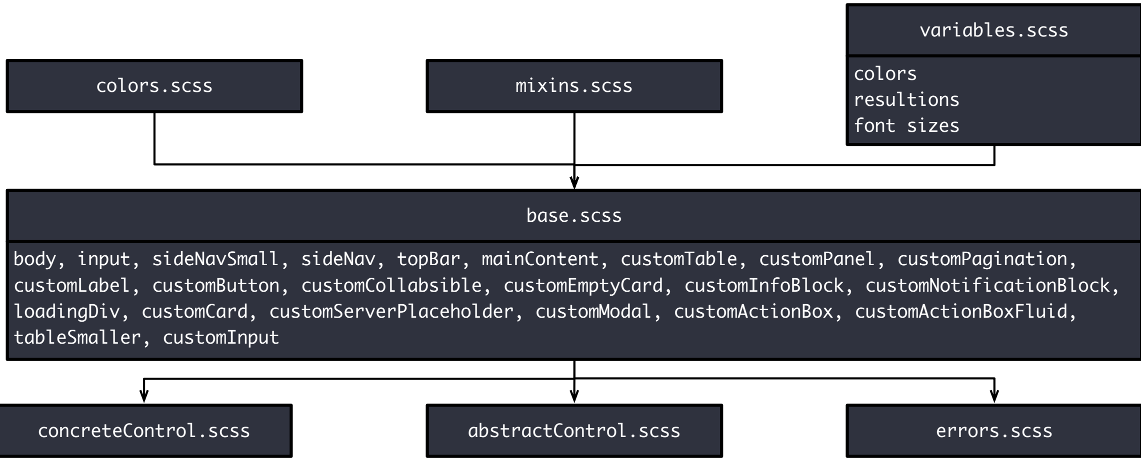
ktoré definujú funkcie zodpovedné za zobrazenie prvku indikujúceho načítavanie a funkcie zobrazujúce chyby servera vo forme hlásení.



Obr. 7.10: Schéma modulov a tried modulu responzívnosti

7.5 Implementácia designu aplikácie

Design aplikácie je implementovaný využitím technológie SASS popísanej v sekcii 3.3. **Architektúra** je rozdelená na hlavný modul *base.scss*, ktorý obsahuje definície všetkých hlavných prvkov stránky. Abstrahované sú aj hodnoty **farebnej palety** v súbore *variables*. **Komponenty** Nette implementované modulárnou architektúrou popísanou v sekcii 7.3.1 majú vlastnú definíciu štýlu v súbore s rovnakým názvom ako je meno komponenty. V prípade **abstraktných komponentov** je tento systém hierarchie ekvivalentný.



Obr. 7.11: Schéma architektúry štýlov

7.6 Správa dočasného úložiska

Najpomalšia časť aplikácie je napojenie na GraphQL server, ktorý musí spracovávať dáta z databázy a posielať ich webovému serveru. Pre urýchlenie tejto časti je využitá dočasná pamäť implementovaná cez server *Redis* popísaný v sekcii 3.7. Táto implementácia je obsiahnutá v knižnici SDK pre GraphQL. Každému volaniu tejto knižnice je možné priradiť hodnotu exspirácie položky. Pre jednotnosť sprostredkováva aj časové hodnoty popísané v tabuľke 7.1. Podrobnejšie nastavenia exspirácie možno nájsť v prílohe F.

Funkcia	Hodnota	Použitie
getSecondsRemoveASAP	minúta	prihlasovanie, profil, stav portov, doménový súbor, tikety
getSecondsHour	hodina	denné, DDoS, ip, štatistiky meračov a podpory
getSecondsWorkDay	4 hodiny	zoznam a detaily služieb, kreditné karty, strom služieb
getSecondsToDayEnd	1 deň	zobrazenie faktúr a detailu faktúr
getSecondsDay	1 deň	zoznam portov a smerov, mesačné štatistiky, správy, osoby
getSecondsMonth	1 mesiac	technické informácie, jazyky

Tabuľka 7.1: Definície časových hodnôt exspirácie

7.7 Kompatibilita TypeScript modulu

TypeScript modul je vo výsledku spracovaný kompilátorom *Babel*, ktorého spôsob fungovania popisuje sekcia 3.10. Podporované sú prehliadače Edge, Firefox, Chrome a Safri v posledných dvoch verziách. Toto nastavenie definuje súbor *.babelrc*. Kompilácia použitím nástroja Babel je súčasťou nastavenia aplikácie Webpack.

7.8 Automatické testovanie GraphQL API

Aplikácia je napojená na GraphQL API, z ktorého prijíma dáta. Pretože tento server nemá vlastné testy, môže sa stať, že niektoré používané štruktúry sa zmenia bez upravenia ich definície v aplikácii. Takýto problém je veľmi ťažké odhaliť a môže sa objaviť až v prípade, že užívateľ bude konkrétne dáta požadovať. Písanie a zahrnutie **unit testovania** do procesu vývoja aplikácie je preto úplnou samozrejmosťou.

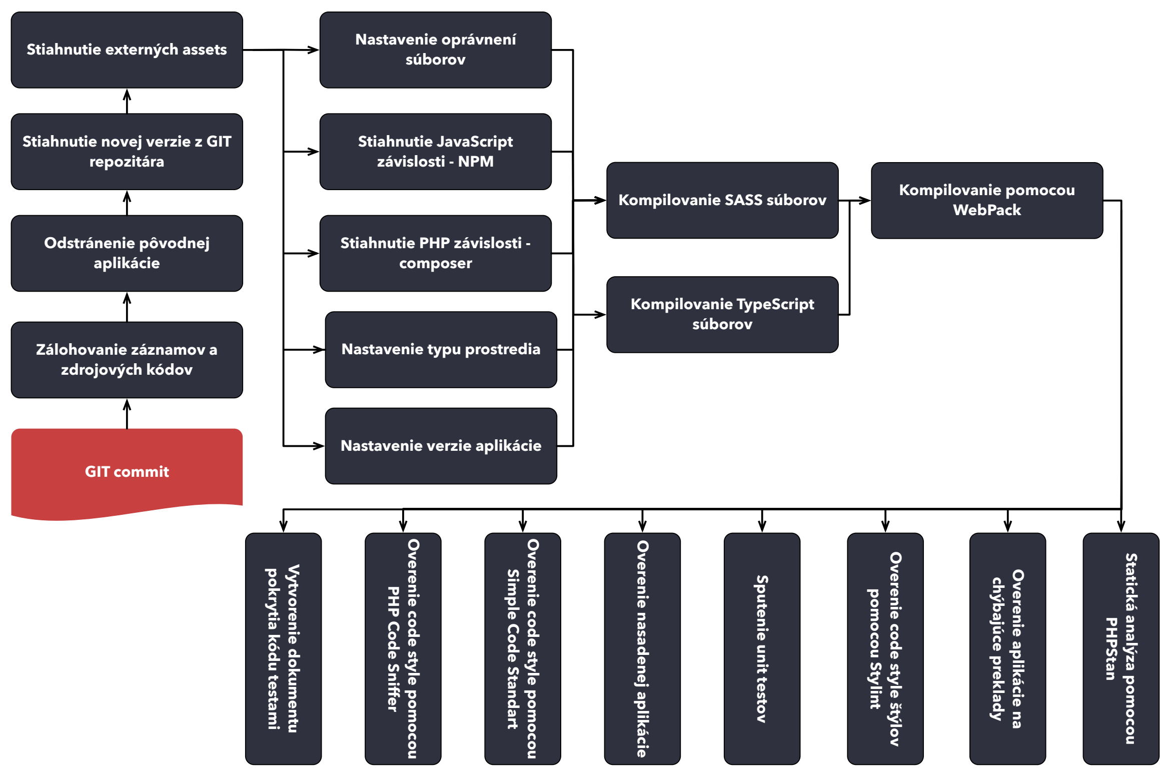
Testy sú implementované použitím knižnice *Nette Tester*[5], ktorá funguje podobne ako *PHPUnit*[6] avšak niektoré veci uľahčuje, čím skracuje čas potrebný na písanie testov. Každý test overuje funkcionalitu konkrétnej triedy *repository*. Po získaní dát, overuje typy vrátených hodnôt a chyby vo forme výnimiek. Po spustení testov je vygenerovaný dokument, ktorý zobrazuje **aktuálny stav pokrytia kódu testami**, takzvaný *coverage report*[7]. Ten zobrazuje percentuálnu hodnotu pokrytia a rozlišuje časti kódu, ktoré sú pokryté od tých, ktoré nie sú a je potrebné dané testovanie doplniť. Spúšťanie testovania je súčasťou procesu **integrácie aplikácie**. Všetky testy by malo byť možné spustiť vo viacerých vláknach.

[5]Dokumentácia Nette Tester https://tester.nette.org/cs/
[6]Projekt PHPUnit https://phpunit.de
[7]Code coverage https://confluence.atlassian.com/clover/about-code-coverage-71599496.html

7.9 Systém priebežnej integrácie

Nahrávanie zdrojových súborov priamo na server nie je efektívne a umožňuje vznik rôznych chýb z dôvodu nedodržania presného postupu. Lepším spôsobom je tvorba **automatizačného systému**, ktorý pri niektorých zmenách vo verzovacom systéme aplikáciu automaticky integruje a to aj v prípade, že je potrebné nasadzovanie na viacero serverov.

Systém bol implementovaný použitím systému *GitLab pipelines*, ktoré umožňujú tvorbu **automatických scenárov**. **Konfigurácia** je možná cez konfiguračný súbor, ktorý sa nachádza v rodičovskom adresári projektu s názvom *.gitlab-ci.yml*. Na začiatku systém **zálohuje** aktuálne integrovanú verziu a všetky jej zdrojové súbory vrátane záznamov. **Zmazaná** je stará verzia aplikácie. **Aktuálna verzia** je získaná cez verzovací systém. Potrebné je stiahnuť **externé závislosti**, ktoré sa nakopírujú do verejnej zložky. Pre beh systému je dôležité **nastavenie správnych oprávnení** na jednotlivé adresáre. Programy *NPM* a *Composer* popísané v sekcii 3.5 stiahnu všetky knižnice potrebné pre **kompiláciu** TypeScript modulu a beh webového servera. Aplikácia má možnosť bežať v rôznych prostrediach. Skript musí definovať aktuálne **prostredie behu aplikácie**. Pri integrácii na testovacie a produkčné prostredia je potrebné upraviť aktuálne bežiacu **verziu**. Ďalším krokom je **kompilácia zdrojových súborov** *SCSS* popísaných v sekcii 3.3 a *TypeScript* v sekcii 3.4. **Výsledné súbory** spracuje aplikácia *Webpack* opísaná v sekcii 3.6 a vytvorí z nich jediný súbor pre každý *presenter*. V predposlednom kroku sa **overí zdrojový kód** pomocou techník popísaných v sekcii 9.4 a statickej analýzy kódu. Spustia sa aj unit testy s generovaním *code coverage*, ktoré sú popísané v sekcii 7.8.



Obr. 7.12: Schéma procesu integrácie aplikácie

Kapitola 8

Automatická správa serverov

Implementovanú webovú aplikáciu je potrebné napojiť na systém, ktorý bude automaticky spracovávať požiadavky užívateľov. Kapitola popisuje návrh systému z hľadiska **databázového modelu** a využitia technológie ORM popísanej v 3.8. Navrhnutá je **architektúra** umožňujúca implementáciu všetkých scenárov popísaných v 5. Popisuje spôsoby automatického **testovania a validácie**. Koniec kapitoly popisuje **automatizáciu** celého systému a prepojenie implementovaných komponentov do spoločne fungujúceho ekosystému.

8.1 Konfigurácia prostredia

Pred implementáciou je potrebné zabezpečiť aby systém fungoval na **viacerých prostrediach**. Iné požiadavky sú v prípade, že je systém spustený na vývojovom prostredí a iné na produkcii. Prechod medzi jednotlivými konfiguráciami je zabezpečený cez **nastavenie konštanty** *ENV* v *Bootstrap*[1] súbore aplikácie. V prípade nastavenia požadovanej hodnoty sa využije príslušný konfiguračný súbor, ktorý definuje zvyšné premenné používané *kontajnerom DI* popísanom v 3.2.3. Pretože niektoré nastavenia sú dôverné a zabezpečujú prístup do konkrétneho systému, sú tieto nastavenia vylúčené mimo štandardné konfiguračné súbory a nie sú zdieľané cez repozitár projektu.

8.2 Objektovo relačný model

Všetky úlohy, ktoré je potrebné spracovať, sa nachádzajú v **relačnom databázovom úložisku** *MariaDB*[2]. Aby nevznikalo veľké množstvo nedefinovaných hodnôt pri návrhu, je výhodnejšie použitie **odlišných tabuliek pre každú operáciu**. Každá z týchto tabuliek definuje informácie potrebné pre jej vykonanie a je možné definovať **integritné obmedzenia**, ktoré zabezpečia správne načítanie každej úlohy systémom. Na implementáciu databázy je využitá knižnica *Doctrine*, popísaná v sekcii 3.8, ktorá umožňuje objektové definovanie databázového modelu na strane systému nazývané aj **ORM**. Pri využití tejto technológie je potrebné definovanie jednotlivých entít formou tried. Každému parametru triedy je možné definovať typ ako aj integritné obmedzenia. Rovnako sa definujú aj vzťahy medzi jednotlivými tabuľkami vo väzbách *ManyToMany, OneToMany, ManyToOne*. Po implementácii modelu na strane systému je databázový model **automaticky generovaný**

[1] Bootstrap súbor https://doc.nette.org/cs/3.0/bootstrap
[2] Projekt MariaDB https://mariadb.com

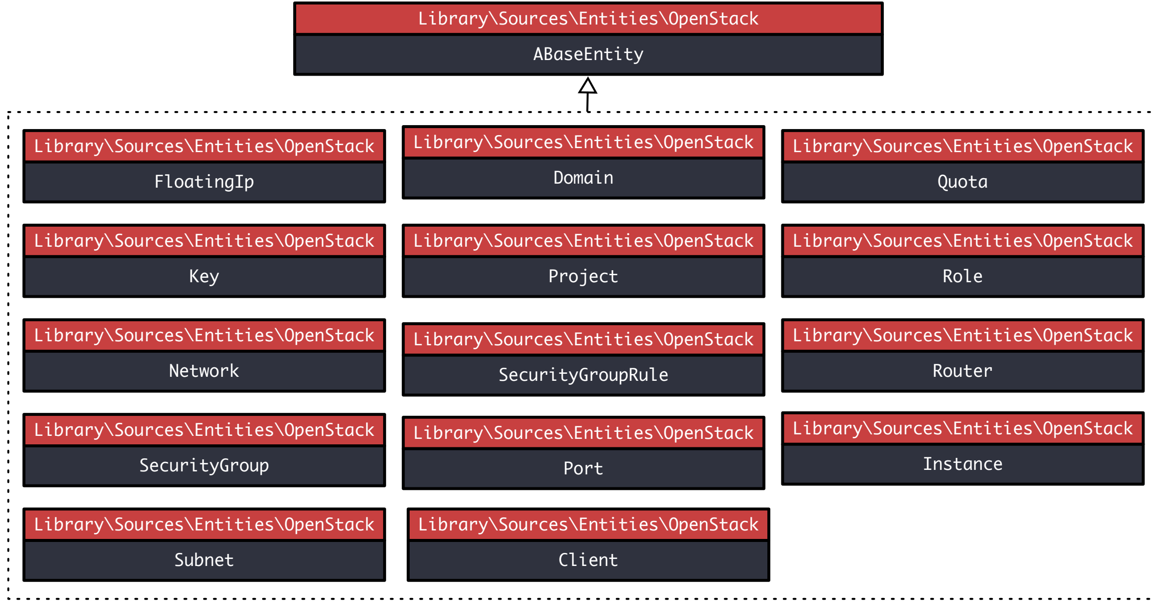
knižnicou. V prípade zmeny entít v projekte je možné vytvoriť *migráciu databázy*[3] ktorá zaručí **aktualizáciu databázového modelu** bez potreby straty dát.

8.3 Modul automatizácie systému OpenStack

Systém je rozčlenený do logických častí, pričom jednotlivé triedy medzi sebou kooperujú. Kooperáciu zabezpečuje **kontrolér**, ktorý implementuje scenár tvorby serveru popísaný v 5.1. **Komunikácia** so systémom OpenStack prebieha prostredníctvom aplikácie napísanej v jazyku Python. Jej spúšťanie prebieha volaním na konzole, ktorá je virtualizovaná zo systému PHP[4]. Všetky odpovede z tejto konzoly majú **výstup vo formáte JSON**, ktorý je následne spracovávaný a ukladaný do DTO entít systému. **Autentifikácia** požiadaviek je popísaná v sekcii 5.1.1. Podrobnejší diagram systému sa nachádza v prílohe C.

8.3.1 Definícia DTO entít

Entity priamo **mapujú jednotlivé logické prvky**, ktoré tvoria architektúru virtuálneho serveru popísanú v 4.4.2. Všetky triedy dedia metódy z triedy *ABaseEntity*, ktorá poskytuje základné **inicializačné funkcie**. Funkcia *initializeEntityFromArray* implementuje funkciu, pri ktorej zo vstupného poľa, ktoré je výsledkom dekódovaného JSON výstupu, **vytvorí DTO entitu** objektového systému aplikácie. V prípade, že výsledkom je kolekcia entít, je potrebné vytvoriť objekt kolekcie. Funkcia *initializeCollectionFromArray* implementuje **tvorbu dynamických entít kolekcie**, ktoré obsahujú jediný parameter a to pole jednotlivých entít. Vďaka tvorbe takýchto tried nie je potrebná ich ďalšia definícia v systéme, čo šetrí značnú časť kódu. Každá entita obsahuje **parametre**, ktoré ju definujú. Tieto parametre rovnako mapujú parametre entít virtualizačného systému. Obrázok 8.1 zjednodušene ilustruje architektúru entít.

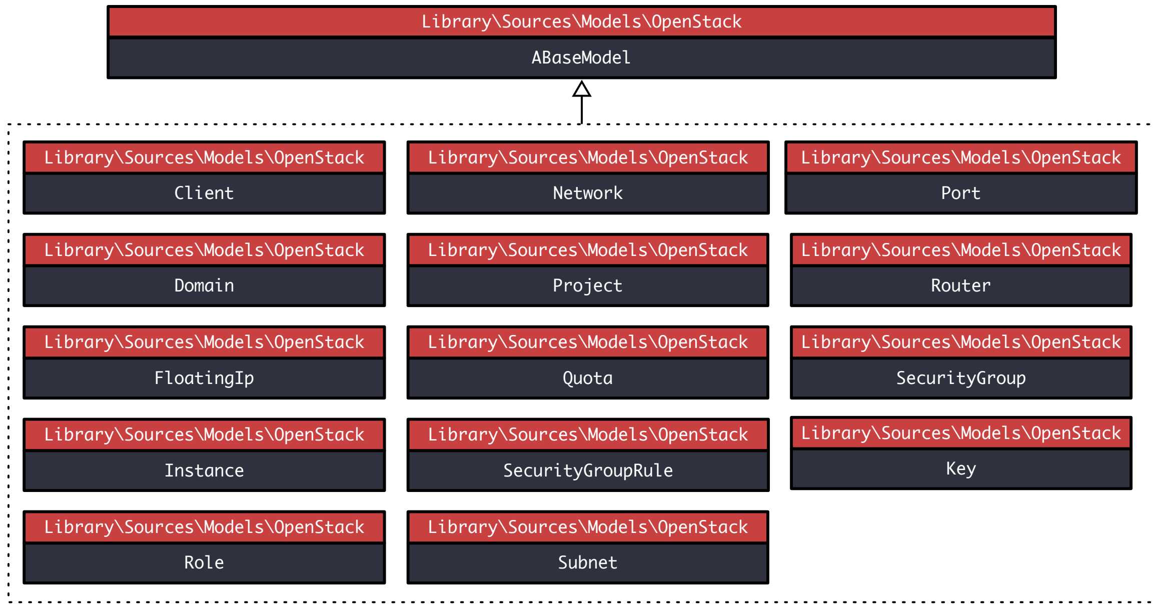


Obr. 8.1: Diagram tried pre entity

[3]Doctrine migračný modul https://www.doctrine-project.org/projects/migrations.html
[4]Virtuálny terminál PHP https://www.php.net/manual/en/function.exec.php

8.3.2 Implementácia aplikačnej logiky

Modely implementujú **operácie**, ktoré je možné vykonávať nad entitami. Všetky modely sú potomkami rodičovskej triedy *ABaseModel*. Táto trieda obsahuje metódy na **vykonávanie konkrétnych operácii** nad aplikáciou virtualizačného systému. Trieda obsahuje konštanty definujúce uloženie *RC súborov* popísaných v 5.1.1 pre zabezpečenie **autentifikácie**. Táto autentifikácia sa dá vyvolať funkciou *setAuthorization*. V prípade, že je potrebné používať daný model cez účet administrátora, stačí túto funkciu nevolať. Pre lepšiu abstrakciu je **pridávanie parametrov** do príkazu, implementované funkciou *addParamerToCommand*. Ak je parameter *boolean* hodnotou, je možné parameter pripojiť príkazom *addBooleanParameterToCommand*. Jednotlivé entity obsahujú metódy, ktoré **mapujú operácie** nad komponentami clusteru. Najčastejšie sa jedná o funkcie, ktoré danú entitu vyhľadajú alebo ju vytvoria. Zjednodušený diagram tried zobrazuje obrázok 8.2.



Obr. 8.2: Diagram tried pre modely

8.3.3 Implementácia kontroléra systému

Kontrolér implementuje **jednotlivé scenáre** popísané v 5.1. Všetky kontroléry sú potomkami rodičovskej triedy *ABaseController*. Táto trieda definuje metódu na inicializáciu užívateľskej **autentifikácie modelov**. Jednotlivé kontroléry obsahujú závislosť na *ORM entitách*, z ktorých čerpajú informácie potrebné pre beh scenára. Funkcia *processByTaskId* **vykoná úlohu** so zadaným ID zavolaním ORM funkcie, ktorá vráti entitu *OpenStackCreateNewServer*. Implementovaný bol základný scenár tvorby serveru, ktorého zaradenie do architektúry zobrazuje obrázok 8.3.

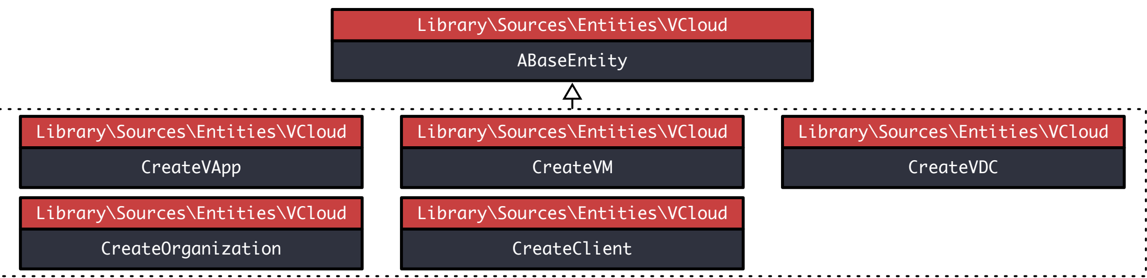
Obr. 8.3: Class diagram pre kontrolér

8.4 Modul automatizácie systému VMware

Modul VMware využíva podobnú architektúru ako OpenStack. **Entity** mapujú stavebné prvky systému, popísané v sekcii 4.5.2. **Model** využívajúci entity implementuje operácie, ktoré sú nad nimi dostupné vo virtualizačnej technológii systému vCloud Director. **Kontrolér** vytvára abstrakciu nad modelmi a implementuje scenáre popísané v 5.2. Autentifikácia je podrobne popísaná v 5.2.1. Podrobnejší diagram systému sa nachádza v prílohe C.

8.4.1 Definícia DTO entít

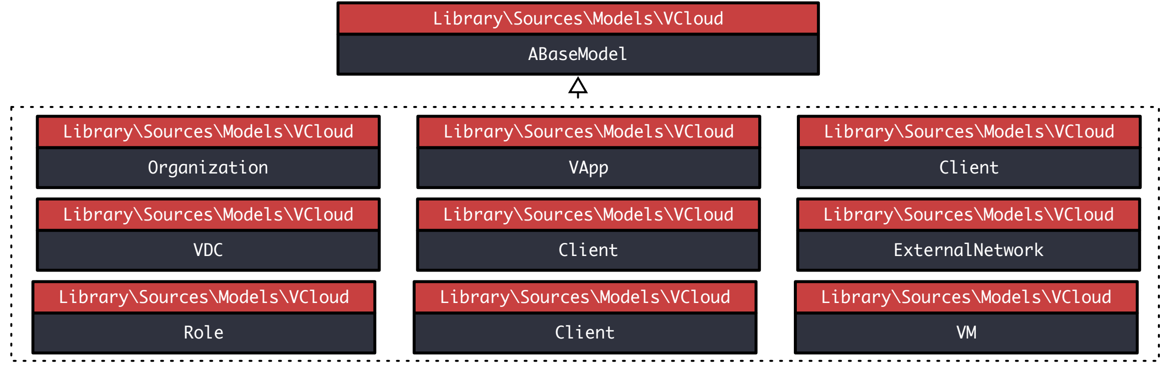
Entity sú **alternatíva** voči jednoduchým dátovým typom. Na rozdiel od týchto dátových štruktúr, objekty majú presne definované svoje **parametre**. V prípade, že funkcia vo svojom parametre vyžaduje namiesto poľa alebo slovníku objekt, je možné presne určiť všetky hodnoty, ktoré je možné nastaviť. Každý parameter môže byť nepovinný a vlastniť **základnú** hodnotu. Všetky entity sú potomkami triedy *ABaseEntity*, ktorá neobsahuje žiadne funkcie alebo parametre, je však lepšie tento vzťah do budúcna definovať. Jednotlivé entity definujú objekty, ktoré sú predávané metódam tried modelu ako parametre.



Obr. 8.4: Diagram tried pre entity

8.4.2 Implementácia aplikačnej logiky

Modely komunikujú priamo s VMware API na ktorého abstrakciu je využitá knižnica *VCloud SDK*. Tá sprístupňuje komunikáciu s **REST API**. Všetky modely sú potomkami triedy *ABaseMode*, ktorá implementuje metódy pre získanie informácií o objektoch virtualizačného systému. Implementované sú metódy, ktoré umožňujú **konverziu** medzi typmi využívanými v SDK. Pre každý typ objektu systému popísaného v 4.5.1 je definovaný model, ktorý nad ním implementuje operácie zo scenára popísaného v sekcii 5.2.



Obr. 8.5: Diagram tried pre modely

8.4.3 Implementácia kontroléra systému

Kontrolér je prvok **abstrakcie**. Nad viacerými modelmi systému implementuje scenáre popísané v sekcii 5.2. Určuje, ako budú tieto entity medzi sebou kooperovať. Postupne volá jednotlivé metódy modelov a spája ich do výsledného objektu. V prípade, že niektorá operácia je asynchrónna, implementuje **čakanie** na dokončenie operácie. Každý kontrolér je potomkom triedy *ABaseController*, ktorá sprístupňuje základné objekty SDK a metódu čakania na asynchrónne úlohy. Momentálne je implementovaný len kontrolér umožňujúci tvorbu nového virtuálneho serveru, postupne je možné definovať ďalšie scenáre a kontroléry.

Obr. 8.6: Diagram tried pre kontrolér

8.5 Modul automatizácie systému Proxmox VE

Architektúra systému Proxmox popísaného v sekcii 4.6 nie je natoľko zložitá ako v prípade OpenStack alebo VMware. Pri implementácii bol však využitý **rovnaký návrh** ako v prípade týchto služieb pre dodržanie jednotnosti implementovaného kódu. Na **komunikáciu** so systémom Proxmox analyzovanom v 4.6.1 je využité REST API. Toto API je v prípade implementácie abstrahované cez **vlastné SDK**, ktoré vykonáva zasielanie žiadostí, spracovanie dát na výstupe a autentifikáciu popísanú v sekcii 5.3.1. Celý systém je klasicky rozdelený na entity, modely a kontroléry. Rozdiely medzi entitami LXC a QEMU vysvetlenými v sekcii 4.6.1 sú minimálne. Veľmi podobná je aj ich ďalšia konfigurácia. Podrobnejší diagram systému sa nachádza v prílohe C.

8.5.1 Definícia DTO entít

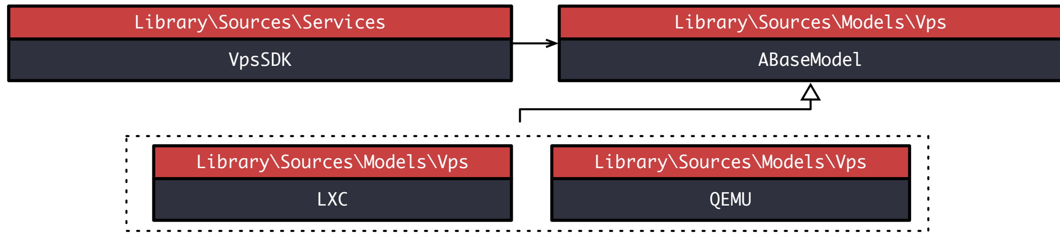
Entity u systému Proxmox namiesto toho aby priamo mapovali jednotlivé objekty systému musia mapovať **jednotlivé operácie** z dôvodu rozdielnych parametrov použitých pri každom volaní. Vo vlastnej implementácii komunikačného SDK je spracovávaný priamo celý objekt DTO entity. Všetky entity pritom dedia od jednej triedy *ABaseEntity*. Tá obsahuje inicializačnú funkciu *initializeEntityFromObject* na naplnenie objektu z výstupných dát zaslaných z rozhrania virtualizačnej technológie Proxmox. Ak je to možné, je každej entite nastavená **preddefinovaná hodnota**.

Obr. 8.7: Diagram tried pre entity

8.5.2 Implementácia aplikačnej logiky

Na implementovanie funkcionality tvorby serverov je potrebné využiť len dve triedy modelov, ktoré dedia od spoločnej *ABaseModel*. Táto rodičovská trieda obsahuje funkcie na **autentifikáciu modelov** použitím *tokenu* a *cookie*. Princíp autentifikácie je popísaný v sekcii 5.3.1. Jednotlivé modely volajú SDK a zasielajú požiadavky na API Proxmox.

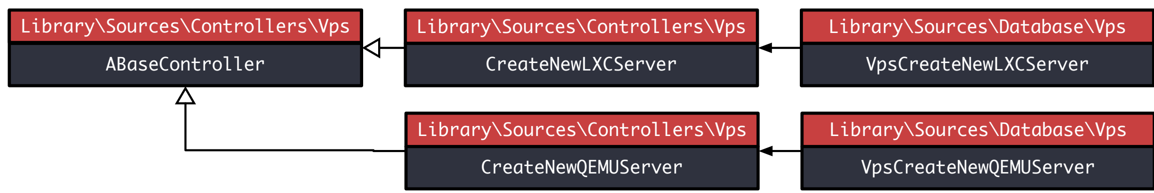
Vlastná implementácia SDK je postavená na knižnici *Guzzle*[5]. Služba obsahuje základné parametre na konfiguráciu spojenia. Poskytuje dve funkcie. *getTicket*, ktorá vráti autentifikačné údaje a *sendRequestWithDataFromObject*, ktorá zasiela požiadavky na rozhranie technológie Proxmox. V prípade chýb sú vrátené výnimky.



Obr. 8.8: Diagram tried pre modely

8.5.3 Implementácia kontrolérov systému

Kontroléry tvoria istú mieru **abstrakcie** nad modelmi. Na vstupe prijímajú z ORM objektovo relačného úložiska **konkrétnu úlohu** pre spracovanie daného scenára. Scenáre sú bližšie popísané v sekcii 5.3. Pretože niektoré operácie sú **asynchrónne**, je zahrnuté do procesu kontroléra čakanie. Všetky kontroléry dedia vlastnosti od *ABaseController*. Táto trieda obsahuje inicializáciu všetkých služieb. Implementované sú scenáre pre vytvorenie *QEMU* a *LXC* inštancie. Kontroléry budú ďalej volané zo systému automatizácie popísaného v 8.8.



Obr. 8.9: Diagram tried pre kontroléry

8.6 Automatické testovanie systému

Na testovanie celého systému automatickej správy serverov bola využitá knižnica *Nette Tester* ako v prípade webovej aplikácie. Pre každý **model** je implementovaný jeden *unit test*, ktorý overuje funkcionalitu všetkých metód daného modelu. Triedy testov majú v konštantách definované parametre, s ktorými sú spúšťané pri testovaní. Pretože nie je možné unit testy medzi sebou prepájať a tak overiť funkčnosť celého systému, je potrebné implementovať takýto test aj nad samotnou triedou **kontroléra**. V prípade overenia funkčnosti celého systému by malo byť postačujúce spustenie práve tohto testu, ktorý by mal vykonať príslušný scenár a verifikovať tak funkčnosť celého systému naraz.

[5]Projekt Guzzle http://docs.guzzlephp.org

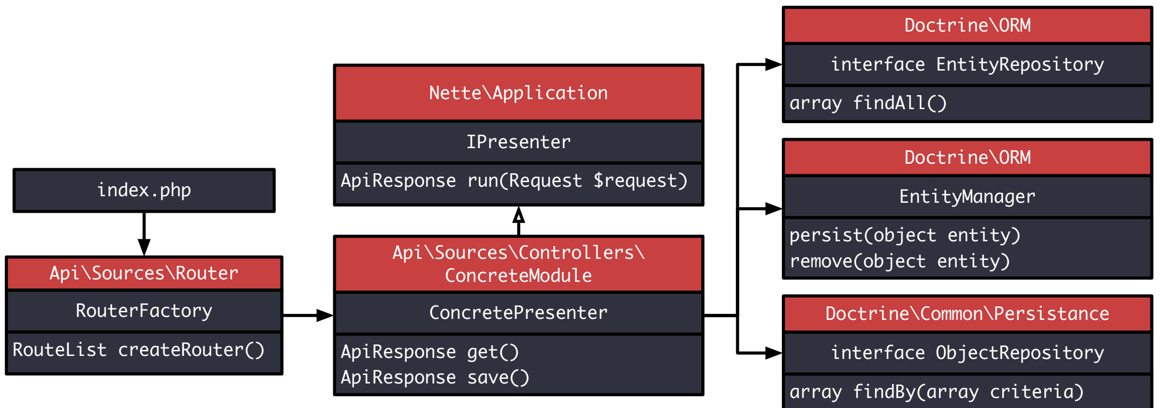
Ďalšia **verifikácia kódu** prebieha ako u webovej aplikácie cez systém automatickej integrácie a nasadenia cez GitLab. Proces tejto implementácie nie je nutné popisovať a je zhodný s procesom nastaveným u aplikácie. V prípade potreby overenia všetkých komponentov je možné nastaviť **automatické spúšťanie** unit testov v daných časových intervaloch.

8.7 Rozhranie automatickej správy serverov

Možnosť prepojenia automatického systému s inými systémami je implementovaná pomocou REST API. Toto rozhranie je napísané v jazyku PHP popísanom v sekcii 3.1. Využíva rovnaký systém dependency injection, popísaný v 3.2.3 ako webová aplikácia. **Štruktúra projektu** je síce rozdielna, ale využíva konceptu aplikácií napísaných v Nette, viď. 3.2.1. Komunikácia s **databázou** je zabezpečená systémom ORM vysvetleným v sekcii 3.8. Rozhranie bolo previazané s implementovaným systémom na automatizáciu úloh. Takéto riešenie zabráni prípadným rozdielom medzi definíciami entít, ktoré sú využívané v technológii ORM obidvoch modulov. Spôsob je vhodný aj z hľadiska spoločného procesu integrácie. Zmeny v implementácii modulu správy serverov musia byť okamžite premietnuté do rozhrania REST API a naopak.

8.7.1 Implementácia rozhrania

Rovnako ako pri všetkých ostatných moduloch je základným bodom implementácie využitie kontajnera DI. Spôsob jeho fungovania detailnejšie popisuje sekcia 3.2.3. Prostredníctvom neho je možná ďalšia konfigurácia modulu API. Pri prijímaní požiadavky sa najprv zavolá inicializačný súbor *index.php*. Ten vytvorí kontajner a inicializuje všetky služby. Najdôležitejšou službou je *router*[6]. Router zabezpečuje premenovávanie požiadaviek na rozhranie do jednotlivých riadiacich kontrolérov. V prípade, že nie je požiadavka konkretizovaná, vykoná sa presmerovanie na štandardný kontrolér *DefaultPresenter*. Pre každý typ virtualizačnej technológie je dostupný jeden kontrolér. Pretože systém využíva technológiu ORM vysvetlenú v sekcii 3.8 nie je potrebné implementovať vrstvu modelu. Táto technológia sa postará o všetky operácie nad databázou v už abstrahovaných volaniach. Kontrolér generuje na výstupe odpoveď typu *ApiResponse*. Obsahuje návratový kód, zoznam chýb, ktoré nastali pri spracovaní a dátový výstup. Formát je detailnejšie popísaný v 8.7.2.



Obr. 8.10: Architektúra rozhrania API

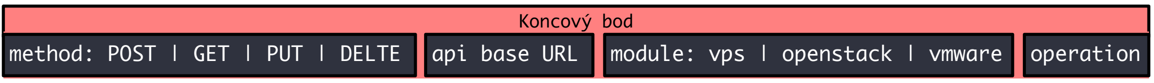
[6]Smerovania v Nette aplikácii https://doc.nette.org/cs/2.4/routing

8.7.2 Komunikačný protokol API

Zasielanie požiadaviek do systému automatickej správy serverov je realizované cez **koncové body** v URI. Koncept *REST*[7] využíva viacero techník, pre zabezpečenie čo najlepšej definície **komunikačného protokolu**. Rozhranie by malo disponovať **jednotným formátom** požiadaviek a odpovedí. Všetky požiadavky sú implementované cez koncept *CRUD*[8] akcií, ktoré umožňujú **manipulovať** so záznamami uloženými v databázy. V prípade *POST* operácií sa dáta posielajú po zakódovaní konkrétnych entít DTO do serializačného formátu JSON. **Výstup rozhrania** má tiež definovaný jednotný formát odpovedí klientskemu serveru. Každá odpoveď obsahuje návratovú hodnotu vo forme číselného kódu, kolekciu chýb, ktoré sa vyskytli pri spracovaní a dáta poskytujúce odpoveď servera na žiadosť klienta.

Obr. 8.11: Komunikačný formát odpovedí API

Jedným z kritérií konceptu REST je, že všetky volania neuchovávajú **stav** daného systému. Táto požiadavka je splnená priamo cez princípy fungovania PHP servera, ktorý nie je schopný udržiavať stav mimo *session alebo cookies*[9]. Ukladanie **výsledkov** nie je pri tomto API dôležité. Túto vlastnosť je možné implementovať priamo v aplikácii, ktorá rozhranie využíva. Volania prebiehajú vždy cez **klient - server** komunikáciu, kde klientom je webová aplikácia alebo užívateľ priamo komunikujúci s API a rozhranie API je serverom poskytujúcim funkcie. Samotný formát koncového bodu ilustruje obrázok 8.12.



Obr. 8.12: Definícia koncových bodov

Filtrovanie výstupných výsledkov je sprístupnené prostredníctvom URI. Ak chceme filtrovať podľa parametra DTO entity (viď. prílohu E), s ktorou požiadavka pracuje, stačí na konci URI zadať GET parameter s názvom parametra a hodnotou, ktorú chceme použiť ako filtračnú. Nižšie je možno vidieť príklad takéhoto koncového bodu v prípade filtrovania podľa čísla klienta nad požiadavkou tvorby nového VMware servera. V prípade potreby by bolo možné použiť akýkoľvek iný parameter z definície DTO objektu entity.

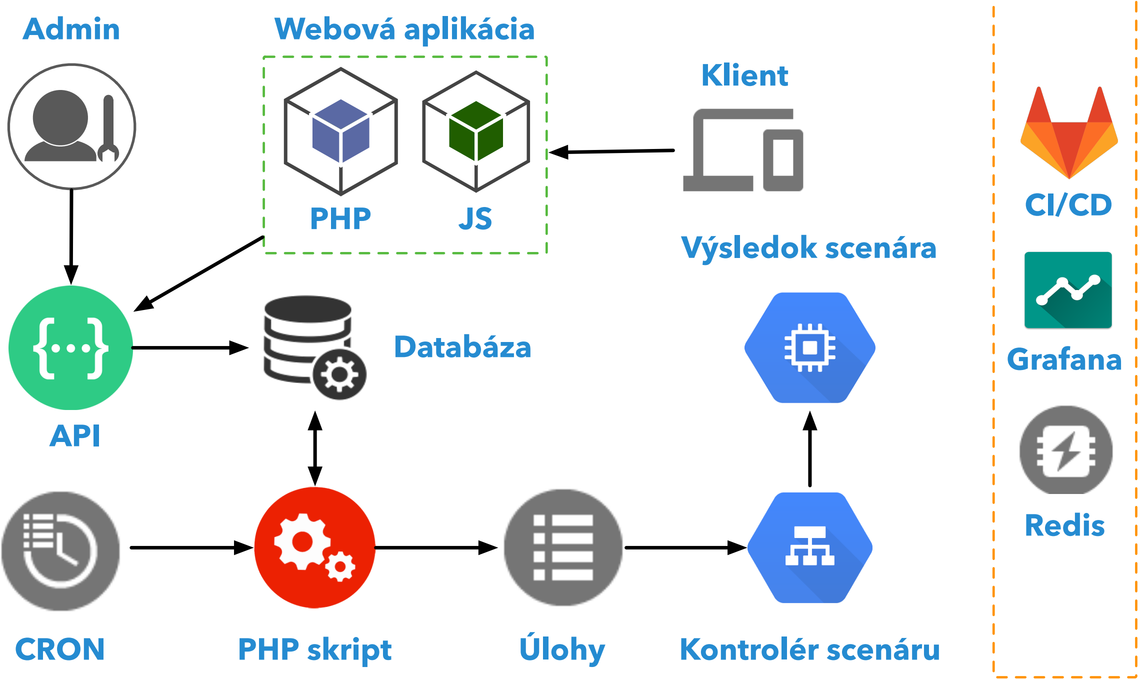
```
apiAddress/vmware/v-mware-create-new-server?clientId=2433
```

Pre možnosti validácie systému API modulu je potrebné **zaznamenávanie komunikácie**, ktorú vykonáva trieda *Logger*. Podobne ako u ostatných komponentov, ukladá výsledky a rozdeľuje ich medzi úspešné a neúspešné. V prípade chyby je zasielané hlásenie do komunikačnej služby Mattermost. Tento systém bližšie popisuje práca v sekcii 9.4. Príklady komunikácie s API je možné nájsť v prílohe E.

[7] Požiadavky na REST API https://www.restapitutorial.com/lessons/whatisrest.html
[8] Definícia CRUD https://www.codecademy.com/articles/what-is-crud
[9] Ukladanie stavu v PHP https://docstore.mik.ua/orelly/webprog/php/ch07_06.htm

8.8 Automatizácia procesov systému

Existuje viacero možností komunikácie s API rozhraním systému automatizácie. V prípade, že klient žiada cez webovú aplikáciu vytvorenie nového servera, interakciu inicializuje modul TypeScript popísaný v 7.4.1. Ten komunikuje so serverom PHP, ktorý vytvorí novú požiadavku cez rozhranie API. V prípade potreby vytvorenia servera mimo webovú aplikáciu, môže administrátor vytvoriť takúto požiadavku priamo na systém. Celý systém môže obsahovať ľubovoľný počet scenárov, ktoré je potrebné **automaticky spúšťať**. Tento proces je implementovaný pomocou *CRON úloh*[10], ktoré je možné spúšťať s ľubovoľnou frekvenciou pre každý scenár. Na začiatku procesu sa zavolá **PHP skript**, ktorý cez argument na svojom vstupe identifikuje o ktorý scenár sa bude jednať. Na základe neho vyberie z databázy všetky úlohy, ktoré boli verifikované, potvrdené a neboli ešte spracované. Na identifikovanie stavu úlohy slúži stĺpec status, ktorý obsahuje enum hodnotu. Po načítaní zoznamu, začne systém automatickej správy serverov sériovo spracovávať **jednotlivé úlohy**. Paralelne vykonávanie nepripadá do úvahy pretože operácie nad systémami virtualizácie nie sú atomické. Počas vykonávania úlohy dochádza k **zmene stavu** hodnoty v stĺpci *progress*. Táto hodnota označuje počet dokončených aktivít spracovávanej úlohy. Hodnotu je možné využiť na zobrazenie priebehu spracovávania. Po dokončení úlohy sa zmení jej stav a pokračuje sa na ďalšiu úlohu v poradí. Nakoniec je celý proces ukončený. Nad celým systémom funguje úložisko dočasnej pamäte Redis, viď. 3.7 a monitoring, ktorý bude bližšie popísaný v sekcii 9.4. Automatické testovanie webovej aplikácie a systému automatizácie je implementované cez proces v službe GitLab. Tento proces podrobnejšie popisuje 7.9.



Obr. 8.13: Proces automatického spracovania úloh

[10]Použitie CRON úloh https://www.ostechnix.com/a-beginners-guide-to-cron-jobs/

Kapitola 9

Testovanie systému

Táto kapitola sa zoberá samotným **testovaním** výsledného riešenia systému a jeho overenia. V sekcii 9.1 je popísaný spôsob, akým bolo testovanie medzi užívateľmi vykonávané. Časti 9.2 a 9.3 popisujú konkrétne testovanie z hľadiska užívateľa a **odozvy systému** na jeho reakcie. Časť 9.4 popisuje spôsoby, akými je možné **verifikovať**, že výsledná aplikácia neobsahuje chyby a ktorým častiam je potrebné venovať budúci vývoj. Možnosti ako riešiť **výpadky** a čo najlepšie informovať klientov o prípadných problémoch a odstávkach sú analyzované v sekcii 9.4.4. Záver kapitoly 9.2.6 popisuje získané **nápady na zlepšenie**, ktoré boli výstupom testovania systému.

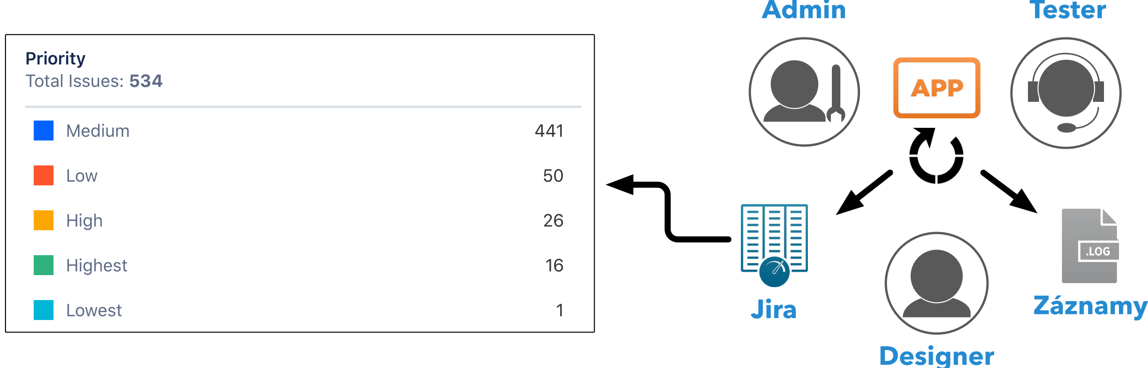
9.1 Spôsob testovania

Testovanie webovej aplikácie prebiehalo na rôznych **prehliadačoch**, ktoré by mali pokrývať čo najväčšiu skupinu zákazníkov a ktorých podpora je zahrnutá v systéme kompatibility jazyka Javascript Babel, ktorý je popísaný v kapitole 3.10. Jednalo sa o prehliadače *Safari, Chrome* a *Mozilla Firefox*, ktoré sú najpoužívanejšími v Českej republike. Medzi testovanými **operačnými systémami** bol zatiaľ *Microsoft Windows* a *MacOS* od spoločnosti Apple. Do budúcna budú v rámci ďalšieho testovania responzie zahrnuté predovšetkým **mobilné prehliadače** a mobilné systémy *iOS* a *Android*. Vhodné by bolo aj testovanie na systéme *Linux*, ktorý je využívaný veľkým počtom systémových a serverových administrátorov. Problémom je však **časová náročnosť**. S každým operačným systémom a prehliadačom, prípadne možným rozlíšeným násobne vzrastá časová náročnosť testovania výsledného riešenia. Aktuálny testovaný rozsah je preto dostačujúci.

9.1.1 Segmentácia testovacích užívateľov

Testovanie bolo rozdelené na dve časti. V prvej prebiehalo testovanie na úrovni **zamestnancov** v tíme testerov spoločnosti *MasterApp s.r.o.*. Títo zamestnanci nemali žiadne predchádzajúce znalosti o implementovanom systéme a tak vykonávali bežné užívateľské testovanie a scenáre. Kvôli nedostatku skúseností v danej problematike neboli schopní systém otestovať z technického hľadiska. V druhej časti testovania bol systém zaslaný tímu **administrátorov**, ktorí denne pracujú so všetkými službami a virtualizačnými technológiami. Priebežne boli tak zhromažďované problémy, ktoré boli nájdené u oboch tímov. Po identifikácii chýb a problémov boli realizované opravy a vydania nových verzií webovej aplikácie. Tento cyklus sa opakoval až do doby, kedy neboli zistené žiadne ďalšie chyby, ktoré by mohli brániť vydaniu aplikácie na produkčné prostredie. V priebehu testovania bolo viackrát vykonané

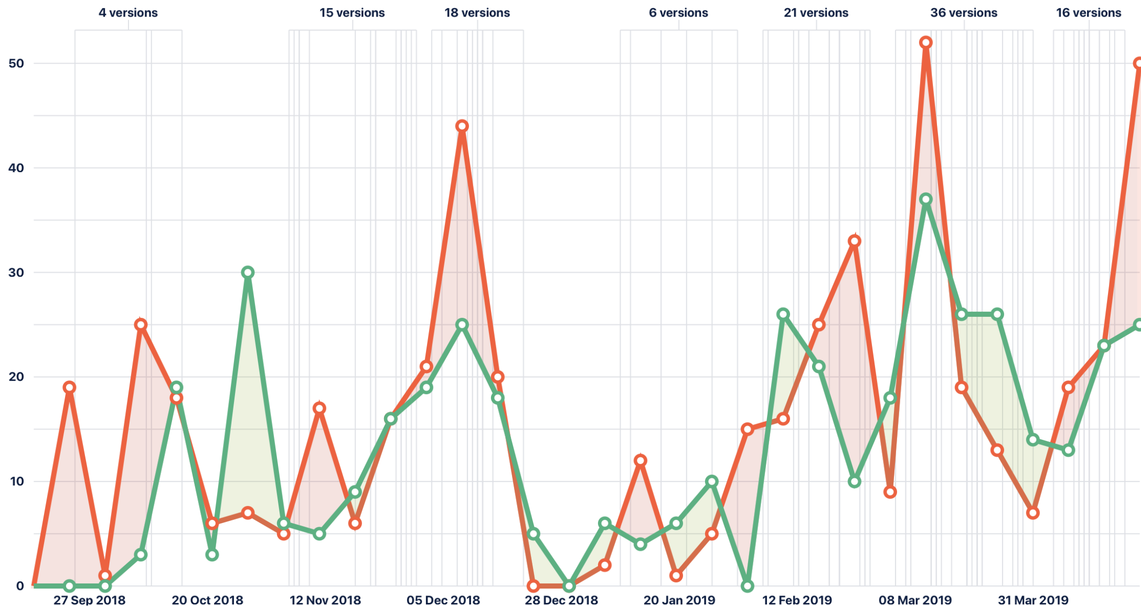
overenie užívateľského rozhrania a jeho korekcia v spolupráci s **tímom grafikov**. Takýmto testovaním by mali byť pokryté všetky časti aplikácie spolu s veľkou časťou konečných užívateľov a zákazníkov firmy.



Obr. 9.1: Segmentácia testovacích užívateľov a problémov podľa priority

9.2 Testovanie prototypu

Testovanie prototypu bolo nutné rozčleniť na viacero funkčných častí z dôvodu značného rozsahu celého projektu. Na začiatku prebehlo testovanie zobrazenia **jednotlivých služieb**. Postupne bolo potrebné overiť všetky **štatistiky a grafy**. Aby bolo možné zapnúť na produkčnom prostredí **dočasnú pamäť** pomocou nástroja Redis, viď. 3.7 muselo tomu predchádzať overenie, či jednotlivé položky majú správne nastavenú exspiráciu. Nakoniec prebehlo interné testovanie **automatického systému**. Postupne boli testované všetky časti systému a následne vydané **nové verzie**. Rozdiely medzi vytvorenými a vyriešenými problémami v čase zobrazuje obrázok 9.2. Nad grafom sa zobrazujú počty vydaných verzií v danom období.



Obr. 9.2: Vytvorené (červeným) a uzavreté problémy (zeleným) v čase

9.2.1 Validácia služieb

Služby sú logicky rozdelené podľa ich typu. Zoznam všetkých typov služieb, ktoré boli v systéme implementované zobrazuje príloha D. Pri službách bolo potrebné overiť informácie, ktoré boli zobrazované v **tabuľkovom formáte**. V tejto časti sa najčastejšie vyskytovali problémy s chýbajúcimi informáciami, jednotkami alebo špatnými **prekladmi** v českej alebo anglickej variácii stránky. Problémom tohto testovania bol veľký počet služieb a tabuliek s množstvom rôznych informácií.

9.2.2 Overenie štatistík

Štatistiky boli rozdelené na tie, ktoré zobrazovali **aktuálne hodnoty** v mesiaci a tie, ktoré poskytovali **historické hodnoty** za uplynulé mesiace v posledných rokoch. U všetkých štatistík je potrebné overiť správnosť hodnôt, zobrazované jednotky a intervaly, za ktoré boli hodnoty zobrazované. V prípade, že bolo možné niektoré štatistiky **filtrovať** na základe zadaných filtrov alebo bolo možné zobrazovať stránky pomocou stránkovania, bolo potrebné vykonať rozsiahlejšie testovanie aj týchto funkcií. Najčastejšími problémami boli chybné jednotky a intervaly zobrazovaných položiek. Pri filtrovaní bolo značne náročné overiť, či výsledné hodnoty sú skutočne správne filtrované.

9.2.3 Analyzovanie grafov

Asi najzložitejším bolo testovanie zobrazenia **grafov** pomocou knižnice HighCharts popísanej v sekcii 3.9. Problémom bolo overenie hodnôt priamo z grafu. Jednotlivé zobrazenia sa líšili od typu zobrazovaného grafu a služby, ku ktorej bol graf priradený. V prípade zobrazenia využitia **zdrojov virtuálnych serverov** bola potrebná validácia, či hodnoty neprekračujú tie, ktoré zasiela virtualizačný nástroj. V prípade grafov zobrazujúcich **dátové toky** bola potrebná hlavne validácia zobrazených jednotiek a hodnôt. Bolo potrebné pokryť aj neštandardné situácie ako napríklad situáciu, keď graf neobsahoval žiadne hodnoty alebo hodnoty v grafe mali byť nulové. V prípade, že server hodnoty za daný časový interval neposlal, systém musel tieto hodnoty generovať. Ťažké bolo rozlíšenie, či sa jedná o zobrazenie hodnôt mimo interval, alebo je potrebné tieto hodnoty prezentovať ako nulové.

9.2.4 Testovanie dočasného úložiska

Testovaným prvkom bola aj **dočasná pamäť** implementovaná prostredníctvom serveru Redis, viď. 3.7, používaná pri načítavaní dať z GraphQL servera. Tá slúži na urýchlenie odozvy stránky. V prípade jej použitia môžu vzniknúť problémy s **exspiráciou** zmenených údajov. V prípade, že užívateľ odošle zmenu nejakej entity, je potrebné takéto dáta vymazať z pamäte a pri prekreslení zobraziť už upravené dáta. V niektorých prípadoch je možné dáta editovať externe, pričom je nemožné mazať takto zmenené hodnoty. Potrebné je nastaviť správne **exspiráciu dát** v pamäti.

9.2.5 Automatizácia tvorby serverov

Posledným krokom bolo testovanie automatického systému tvorby serverov. Testovanie zatiaľ prebiehalo len na úrovni implementácie, ale v budúcnosti bude systém podrobený testovaniu administrátorským tímom. Vyžadované bude overenie systému a jeho chovania pri rôznych konfiguráciách serverov, ktoré sa majú vytvoriť. Dôraz treba klásť na validáciu vstupných hodnôt, ktoré by mohli spôsobiť inkonzistenciu virtualizačných nástrojov.

9.2.6 Podnety z testovania systému

Na základe testovania podali oba tímy svoje návrhy, ako by bolo možné aplikáciu do budúcna **vylepšiť** po stránke funkcionalít a užívateľského rozhrania. Nasledujúce body zobrazujú najčastejšie požiadavky na budúce zapracovanie do systému.

- **Reorganizácia** kategórií a položiek, ktoré sa v danej kategórii zobrazujú. Momentálne existuje niekoľko typov zobrazovaných služieb, avšak ich zaradenie nie je vždy správne a veľa služieb je označovaných ako nadštandardné služby.
- Celková **úprava užívateľského rozhrania** tak, aby toto zobrazenie bolo priamo navrhnuté pre zobrazenie na menších obrazovkách. Potrebné je aplikovať režim blokového zobrazenia namiesto tabuliek pre všetky pohľady.
- Pridanie nových funkcií do **grafov**. Potrebné bude implementovať export do tabuľkových formátov a zladiť ich design so zvyškom aplikácie. Všetky tieto funkcie by mala poskytovať knižnica HighCharts popísaná v sekcii 3.9.
- Implementácia **užívateľských rolí**. Je potrebné, aby napríklad účtovníčka vo firme mohla zobraziť faktúry, ale nebola schopná manipulovať so servermi.
- **IP adresy** by sa mali zobrazovať agregované s ich celkovým počtom.
- Vylepšenie odozvy **DDoS štatistík**. Momentálne sú získavané z externého systému na strane GraphQL API. Výsledky je treba sťahovať s väčším časovým rozsahom a ukladať ich do dočasnej pamäte, pre rýchlejšie zobrazenie stránkovania.

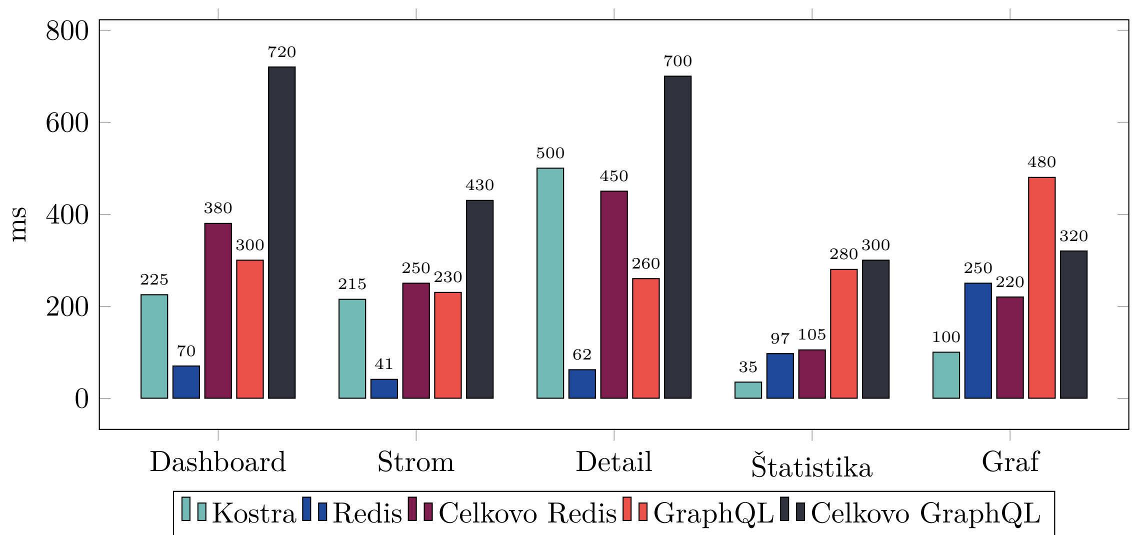
9.3 Testovanie výkonnosti

Rýchlosť je jeden z najdôležitejších parametrov systému. V prípade **webových aplikácií** je potrebné zabezpečiť čo najrýchlejšie odozvy aplikácie a načítania dát. Problém pomalšieho načítavania je možné z časti eliminovať prostredníctvom asynchrónneho načítavania stránok. Tento systém popisuje sekcia 7.4. V prípade načítavania dát z API rozhrania je možné tento proces výrazne zrýchliť použitím dočasnej pamäte servera Redis, viď. 3.7. Problém popisuje sekcia 9.3.1. Komunikácia s **rozhraním API** automatického systému tvorby serverov môže spomaľovať načítavanie dát vo webovej aplikácii. Je potrebné sa snažiť tento čas minimalizovať. Analýzu rozhrania API popisuje sekcia 9.3.2. Pri **virtualizácii** nie je príliš potrebné vynakladať snahu na optimalizáciu rýchlosti. Nižší celkový čas potrebný na tvorbu nových serverov môže umožniť rýchlejšie spracovávanie požiadaviek klienta. Celkový čas pre jednotlivé virtualizačné technológie je možné nájsť v časti 9.3.3.

9.3.1 Optimalizácia rýchlosti aplikácie

Pri testovaní rýchlosti je potrebné hľadieť na viaceré aspekty. V prípade **asynchrónneho načítavania** stránok popísaného v sekcii 7.4, je po načítaní stránky užívateľovi zobrazená len základná štruktúra webu. Toto načítanie by malo byť čo najrýchlejšie. **Načítanie dát** sa vykoná až po zobrazení kostry. Získanie informácií môže prebiehať dvoma spôsobmi. V prvej časti bola meraná časová náročnosť v prípade vypnutej dočasnej pamäte. Komunikácia prebiehala so **serverom GraphQL**. Následne bola dočasná pamäť zapnutá a meraný bol čas potrebný na získanie dát pri využití **Redis servera**, viď. 3.7. Na záver bol zmeraný

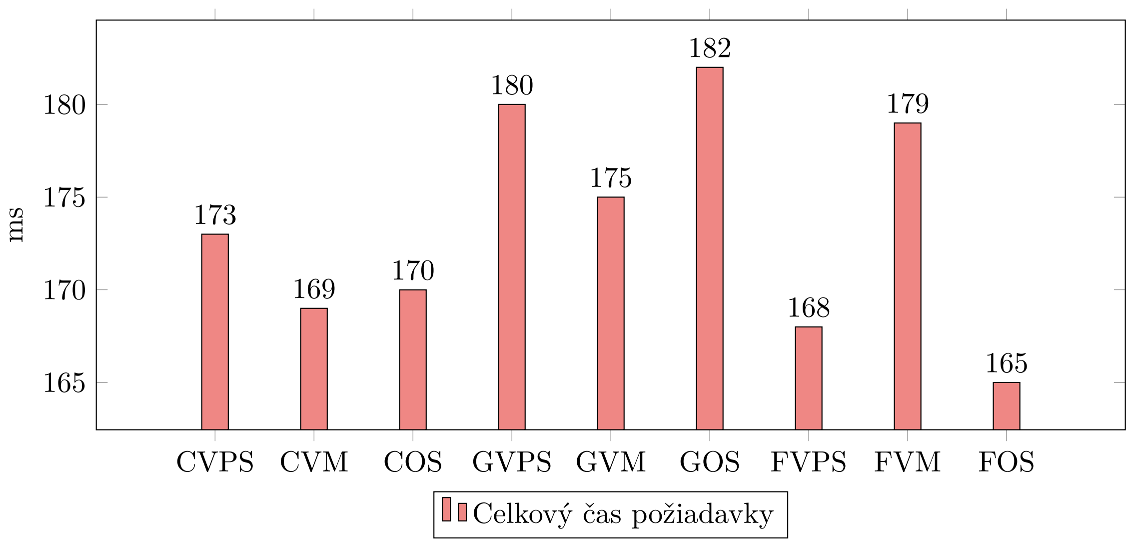
celkový čas zobrazenia stránky. Pri testovaní boli vybrané niektoré hlavné zobrazenia, na ktorých boli merané hodnoty. Jednalo sa o zobrazenie dashboardu, detailu služby OpenStack servera, štatistiky a grafu, ktorý ilustroval využitie virtualizovaných zdrojov. V prípade grafov prebiehalo súbežné načítavanie, celkový čas je preto nižší.



Obr. 9.3: Rýchlosť načítavania stránok webovej aplikácie

9.3.2 Rýchlosť spracovávania požiadavkov cez API

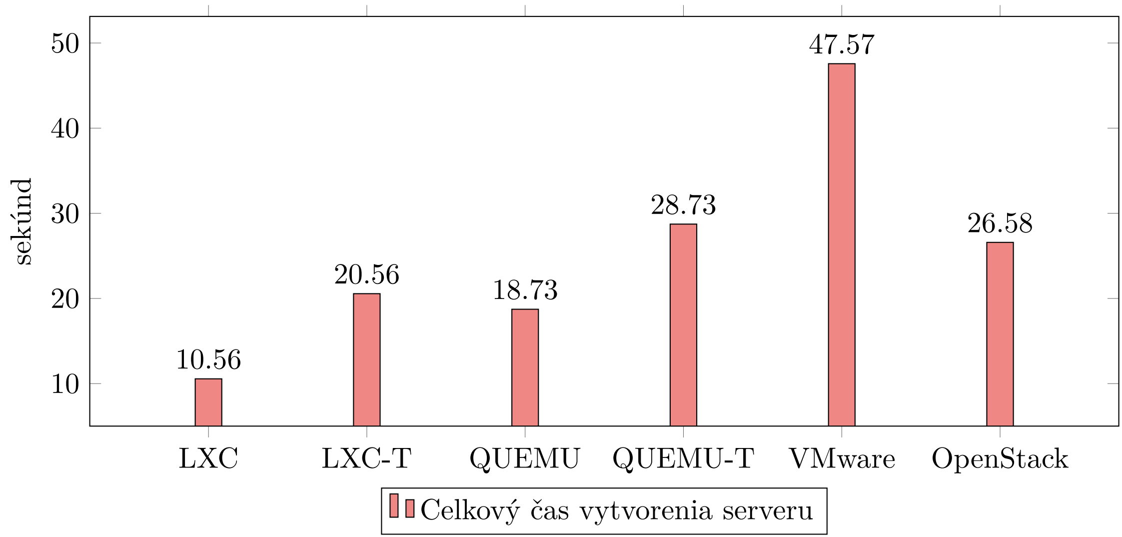
Rozhranie API komunikuje primárne s **webovou aplikáciou**. Rýchlosť načítavania dát priamo ovplyvňuje jej odozvy. Testovanie rozhrania prebiehalo nad každým typom. Graf 9.4 popisuje časovú náročnosť tejto komunikácie. V prípade typov označených písmenom C sa jednalo o **vytvorenie servera**, G popisuje **načítanie záznamov** a F označuje použitie **filtrovania**. Z grafu je vidieť, že táto komunikácia nie je závislá na type požiadavky a celkový čas komunikácie sa pohybuje okolo 170-180 milisekúnd.



Obr. 9.4: Test rýchlosti spracovania požiadaviek API

9.3.3 Časová náročnosť tvorby serverov

V tejto časti sa testovanie zameriavalo na spracovanie požiadaviek na **tvorbu serverov**. V prípade žiadostí označených písmenom T sa jednalo o zahrnutie **pasívneho čakania**. Pretože služba VMware je značne komplexná, tvorba serveru trvala dlhšie ako u ostatných riešení. Najkratšie bolo vytvorenie virtuálnej inštancie prostredníctvom technológie Proxmox, viď. 4.6.1 a kontajneru LXC. U technológie Proxmox je do budúcna potrebné nájsť riešenie, ako nahradiť pasívne čakanie volaním po skončení predchádzajúcej operácie.



Obr. 9.5: Čas spracovania požiadaviek automatického systému

9.4 Verifikácia vydanej aplikácie

Po nasadení webovej aplikácie a systému automatickej správy serverov je potrebné mať implementované nástroje, ktoré by umožňovali **verifikovať** bezchybný beh nasadených systémov. Údržba aplikácie vo väčšine IT projektov zaberá podstatnú časť kapacít vývoja. V systéme bol implementovaný nástroj zasielania **notifikácií** o chybách priamo do firemnej komunikačnej aplikácie *Mattermost*[1]. Na **dohľad** nad samotným serverom a aplikáciou *Apache*[2] bolo postavené riešenie prostredníctvom aplikácie *Grafana*[3].

9.4.1 Spätná väzba od zákazníka

Ak **chybu v systéme** nájde samotný používateľ, je priamo v aplikácii implementovaná možnosť hlásiť tento problém priamo do systému riadenia projektu *Jira*[4]. Vyvolanie dialógu je možné naprieč celou aplikáciou kliknutím na odkaz z ľavého menu. Informácie sa následne vyplnia v zobrazenom modálnom okne. Systém Jira vytvorí nový objekt problému, ktorý môže byť priradený do *sprintu*[5], v ktorom bude vyriešený. Táto funkcia bude v systéme len dočasne a v prípade jej nevyužívania bude odstránená.

[1]Projekt Mattermost https://mattermost.com
[2]Projekt Apache https://httpd.apache.org
[3]Projekt Grafana https://grafana.com
[4]Aplikácia Jira https://jira.atlassian.com
[5]Pojem sprint pri riadení projektu https://www.scrum.org/resources/what-is-a-sprint-in-scrum

9.4.2 Hlásenie chýb aplikácie

Na získanie hlásení o chybách bol v systéme implementovaný **notifikačný systém**. V prípade, že v aplikácii došlo k chybe, alebo nastane problém pri získavaní dát z GraphQL servera, je vygenerovaná notifikácia. Hlásenie obsahuje všetky podrobnosti potrebné pre opravenie problému. Odoslanie smeruje priamo do komunikačného systému, kde je na ňu možné reagovať. Nie je tak potrebné posielanie veľkého počtu mailov, ktoré nie sú príliš prehľadné. Príklad zobrazenia takejto notifikácie zobrazuje obrázok nižšie.

Obr. 9.6: Zobrazenie notifikácie o chybe

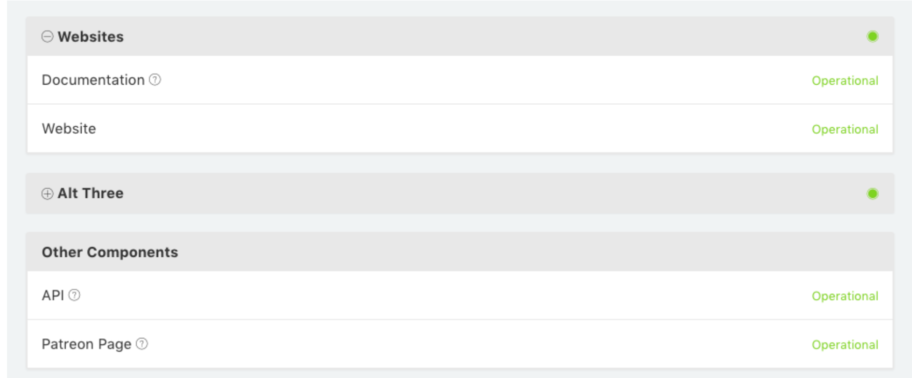
9.4.3 Monitorovanie servera

Počas behu servera môžu nastať rôzne technické problémy. Vhodným riešením týchto rizík je implementovanie systému, ktorý umožní hlásenie takýchto porúch bez zbytočnej odozvy. Nad webovou aplikáciou je vybudovaný systém monitoringu pomocou systému **Grafana**. Grafana je webová aplikácia, ktorá umožňuje získanie metrík priamo zo servera a zobrazenie **grafov** využitím týchto hodnôt. V prípade potreby je možné zapnúť **notifikácie**, ktoré je možné zasielať mailom alebo SMSkou. Obrázok 9.7 zobrazuje aktuálnu konfiguráciu dashboardu nad produkčným serverom.

Obr. 9.7: Zobrazenie dashboardu produkčného serveru Grafana

9.4.4 Podpora pri výpadkoch a údržbe

V prípade chyby systému, ktorá nemôže byť odstránená v krátkom čase, je potrebné problém hlásiť zákazníkovi. Najlepšou možnosťou je **zaslanie upozornenia** o zmene stavu riešenia výpadku. Najlepším spôsobom riešenia takýchto situácií je implementácia *status page* do ekosystému projektu. Takáto stránka zobrazuje jednotlivé komponenty a stav ich fungovania. Pri implementácii status page bol využitý open-source projekt *Cachet*[6]. Umožňuje **manuálne zadávanie stavu** komponentov a hlásení, ktoré by mali informovať užívateľa o probléme alebo odstávke. Dostupné je plánovanie **budúcich udalostí** systému. V prípade, že si užívateľ želá byť informovaný o zmene stavu komponentov, systém Cachet ponúka možnosť **prihlásiť sa na odber** hlásení. Status page je nasadená na inom serveri ako aplikácia, aby v prípade výpadku jedného servera bolo možné na túto stránku pristupovať.



Obr. 9.8: Implementácia status page pomocou Cachet

[6]Projekt Cachet https://cachethq.io

9.5 Metriky kódu

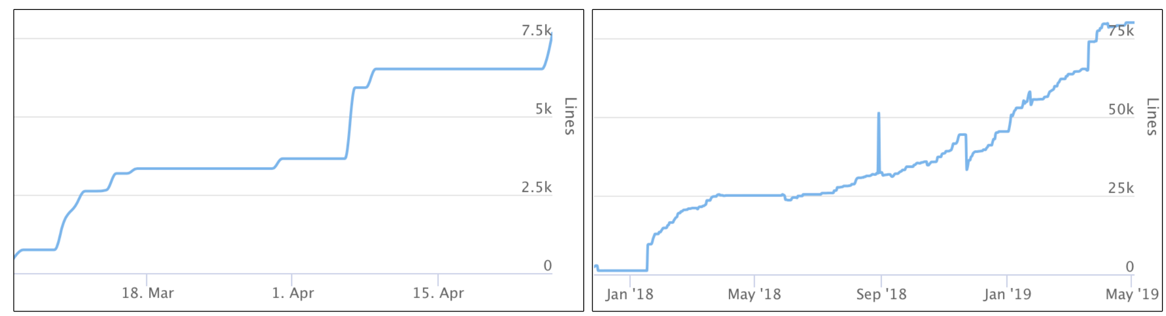
Implementácia v kapitole 7 a 8 je popísaná len veľmi obecne. Kód je v prípade **webovej aplikácie** značne komplikovaný. Riešenie **asynchrónnosti** vyžaduje značne veľkú podporu zo strany kódu. GraphQL server poskytuje údaje pre veľmi **veľký počet služieb**, ktorých zoznam je popísaný v prílohe D. Z hľadiska implementácie nie je systém automatizácie veľmi rozsiahly. Z veľkej časti implementuje **scenáre** popísané v sekcii 5. Tabuľky 9.1 a 9.2 popisujú **rozsah** implementácie a analyzujú jej segmentáciu na jednotlivé **programovacie jazyky**. Pri generovaní výsledkov bola použitá aplikácia *CLOC*[7]. Graf 9.9 zobrazuje závislosť počtu riadkov na čase. Odráža **priebeh vývoja** oboch modulov. Na generovanie grafu bola použitá knižnica *GitStats*[8].

Jazyk	Súborov	Prázdne	Komentáre	Zdrojový kód
PHP	593	6441	12813	20270
TypeScript	178	1334	947	5926
SASS	45	260	8	2213
Diff	2	11	30	611
YAML	2	25	1	469
JavaScript	2	45	22	234
JSON	3	0	0	164
Python	1	31	34	89
Celkovo	827	8147	13855	29976

Tabuľka 9.1: Analýza zdrojového kódu webovej aplikácie

Jazyk	Súborov	Prázne	Komentáre	Zdrojový kód
PHP	79	1062	1742	2641
Bourne Shell	2	9	0	52
JSON	1	0	0	32
Celkovo	82	1071	1072	2725

Tabuľka 9.2: Analýza zdrojového kódu systému automatizácie



Obr. 9.9: Závislosť počtu riadkov kódu na čase.
Vľavo systém automatickej správy serverov, vpravo webová aplikácia.

[7] Aplikácia CLOC https://github.com/AlDanial/cloc
[8] Knižnica GitStats https://sourceforge.net/projects/gitstats/

Kapitola 10

Záver

Cieľom práce bolo vytvorenie webovej aplikácie, ktorá poskytuje možnosť správy užívateľských dát a služieb prostredníctvom automatického systému správy serverov, ktorý je schopný spracovávať úlohy nezávisle, prepojením na rozhrania virtualizačných technológií.

Práca popisuje technológie, pomocou ktorých je možné vyvinúť moderný webový systém. Klasické systémy napísané v skriptovacích jazykoch zvyknú načítavať stránky spolu s dátami. Narozdiel od nich implementácia webovej aplikácie v tomto projekte využíva koncept **asynchrónneho načítavania**. Výsledkom je veľmi rýchle zobrazenie základnej štruktúry stránky, ktorá si ďalšie informácie získa cez oneskorené volania. Skúmané sú možnosti pridania moderných JavaScript knižníc a princípov tohto jazyka do bežnej PHP aplikácie. Takéto riešenie zabezpečí moderný kód, rýchle odozvy, kompatibilitu a možnosť odhalenia chýb a problémov s dátovými konverziami už pri kompilácii.

Komunikácia so serverom poskytujúcim užívateľské dáta a operácie nad službami prebieha cez protokol technológie **GraphQL API**. Na zabezpečenie tejto komunikácie práca poskytuje vlastnú implementáciu SDK. Knižnica umožňuje definovanie entít a schém, ktoré priamo mapujú schémy protokolu. Nad entitami je možné vytvorenie požiadavky, ktorú je server schopný spracovať. Výsledok, ktorý zasiela GraphQL server je spracovaný do pôvodnej kompozície entít, z ktorých bola vytvorená žiadosť. Tento mechanizmus zabezpečuje plné využitie objektového modelu. Takéto riešenie je menej náchylné na programátorské chyby a znižuje čas implementácie a údržby aplikácie. Práca v tomto smere prináša značný prínos pre projekty napísané v jazyku PHP, ktoré využívajú GraphQL server.

Scenáre popisujú znalosti, pomocou ktorých je možné definovať procesy, ktoré umožnia vytvoriť nové virtuálne servery vo viacerých virtualizačných technológiách. Skúsenejší človek, ktorý si prečíta prácu, by mal byť schopný bez väčšej náročnosti implementovať vlastné riešenie automatizácie týchto operácií. Príklad takejto implementácie je súčasťou projektu a poskytuje plne automatizovaný systém sprístupnený cez rozhranie REST API. Umožňuje vytvorenie virtuálnych inštancií v systémoch OpenStack, VMware alebo Proxmox.

Testovanie výsledkov práce je dôležitou súčasťou práce. Posledná časť práce zobrazuje časovú náročnosť tvorby severov. Skúma využiteľnosť dočasnej pamäte Redis a asynchrónnych volaní pri rýchlosti odoziev webovej aplikácie. Popisované sú možnosti integrácie webovej aplikácie s validáciou kódu. Opísané je použitie monitorovacieho systému Grafana na diagnostiku a hlásenie výpadkov servera. Využitý nástroj Cachet umožňuje informovanie zákazníkov v prípade odstávok. Tieto nástroje uzatvárajú celý ekosystém projektu.

Aktuálny **responzívny** design webovej aplikácie je postačujúci, avšak je potrebné ho vylepšiť a niektoré časti špecificky navrhnúť pre mobilné telefóny. Na dosiahnutie tohto cieľa bude potrebná abstrakcia všetkých tabuliek na úrovni implementácie. Takto generované

tabuľky budú inak zobrazované na menších obrazovkách. Myslieť je potrebné na všetky detaily takéhoto mobilného návrhu. Bude sa jednať o rozšírenie sekcie 7.2.2, ktorá sa zaoberá aktuálnym riešením problémov responzívnosti.

Implementácia webovej aplikácie bude doplnená o možnosť **monitorovania zakúpených služieb** prostredníctvom systému *Icinga*[1]. Umožnené by malo byť zobrazenie stavu služieb bežiacich na diagnostikovanom serveri. Komunikácia so systémom Icinga bude prebiehať buď na strane GraphQL API, ktoré bude poskytovať len výsledný stav alebo priamo cez webový server. V tomto prípade bude potrebné vytvoriť SDK.

Zo spätnej väzby získanej z testovania popísaného v sekcii 9.2.6 vyplýva nutnosť vytvorenia modulu **užívateľských rolí**. Aktuálne je implementovaná len jednoduchá jedna rola bežného užívateľa systému, ktorá slúži na prístup do systému všetkým oprávneným osobám. V module oprávnených osôb je možné nastaviť osoby, ktoré môžu využívať technickú podporu spoločnosti, nie je však možné týmto osobám definovať rôzne oprávnenia.

Do budúcna bude potrebné napojiť všetky operácie nad servermi na systém správy serverov. Plánované je rozšírenie automatizácie tak, aby podporovala aj iné funkcie ako len vytvorenie nového servera. Malo by byť možné editovanie konfigurácie serverov, manipulácia s nimi a ich mazanie. Funkcie by nemali smerovať len k virtualizačným technológiám ale mali by pokrývať aj správu iných komponentov datacentra. V prípade nárastu požiadaviek bude treba navrhnúť možnosti implementácie *load balancingu* automatizovaných systémov.

Vývoj technológií využitých pri webovom vývoji napreduje veľmi veľkou rýchlosťou. Aktuálne webová aplikácia nepoužíva najnovšiu verziu Nette frameworku. V novej verzii bola pridaná podpora viacerých moderných techník, ktoré by bolo dobré do aplikácie zahrnúť. Jadro responzívnosti využívajúce knižnicu Nette na podporu volaní v jazyku JavaScript je zastaralé. Na opravu niektorých problémov nebolo možné použiť optimálne riešenie, ktoré spĺňa moderné kritériá. Potrebné bude nahradenie momentálne používaného jadra novou knižnicou *Naja*[2], ktorá zabezpečuje riešenia väčšiny problémov súčasnej knižnice.

Systém webového servera aplikácie využíva technológiu **dočasného úložiska**. Dôležité je vytvorenie prepojenia dočasnej pamäte na systém automatickej tvorby serverov. V prípade vytvorenia alebo zmeny konfigurácie servera je potrebné uložené informácie správne zmazať a tak vynútiť načítanie aktuálnych dát.

[1]Projekt monitoringu Icinga https://icinga.com

[2]Projekt Naja https://naja.js.org

Literatúra

[1] Bierman, G.; Abadi, M.; Torgersen, M.: Understanding typescript. In *European Conference on Object-Oriented Programming*, Springer, 2014, s. 257–281.

[2] Daněk, P.: *Velký test PHP frameworků: Zend, Nette, PHP a RoR*. 2008, [Online; navštíveno 14.11.2018].
URL www.root.cz/clanky/velky-test-php-frameworku-zend-nette-php-a-ror/

[3] Foundation, N.: *Rychlý a pohodlný vývoj webových aplikací v PHP*. 2008-2018, [Online; navštíveno 19.04.2019].
URL https://nette.org/cs

[4] Gavalda, M.: *Server-side I/O Performance: Node vs. PHP vs. Java vs. Go*. 2017, [Online; navštíveno 14.11.2018].
URL
www.toptal.com/back-end/server-side-io-performance-node-php-java-go

[5] Gibb, R.: Redis Explained in 5 Minutes or Less. *MaxCDN Support*, 2018, [Online; navštíveno 24.11.2018].
URL https://www.maxcdn.com/one/visual-glossary/virtual-private-server/

[6] Goldman, R.: *Learning Proxmox VE*. Packt Publishing Ltd, 2016, ISBN 978-1783981786.

[7] Grudl, D.: Začínáme s Nette Framework. *Zdroják. cz*, 2009, [Online; navštíveno 14.11.2018].
URL https://www.zdrojak.cz/serialy/zaciname-s-nette-framework/

[8] Gulati, A.; Holler, A.; Ji, M.; aj.: Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, ročník 1, č. 1, 2012: s. 45–64.

[9] Hartig, O.; Pérez, J.: An initial analysis of Facebook's GraphQL language. Technická správa, Linköping University, Universidad de Chile, 2017.

[10] Huhtakangas, A.; aj.: Productization of web services. 2018.

[11] Keller, E.; Szefer, J.; Rexford, J.; aj.: NoHype: virtualized cloud infrastructure without the virtualization. In *ACM SIGARCH Computer Architecture News*, 3, ACM, 2010, s. 350–361.

[12] Lerdorf, R.; Tatroe, K.; MacIntyre, P.: *Programming Php*. Ö'Reilly Media, Inc.", 2006, ISBN 9780596006815.

[13] Malhotra, L.; Agarwal, D.; Jaiswal, A.: Virtualization in cloud computing. *J. Inform. Tech. Softw. Eng*, ročník 4, č. 2, 2014.

[14] Mazinanian, D.; Tsantalis, N.: An empirical study on the use of CSS preprocessors. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2016, s. 168–178.

[15] O'Neil, E. J.: Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ACM, 2008, s. 1351–1356.

[16] OpenStack: *OpenStackClient*. Https://docs.openstack.org/python-openstackclient/pike/index.html.

[17] Poirier-Ginter, M.: *Using Node.js Express to Quickly Build a GraphQL Server*. 2019, [Online; navštíveno 09.05.2019].
URL https://snipcart.com/blog/graphql-nodejs-express-tutorial

[18] Razina, E.; Janzen, D. S.: Effects of dependency injection on maintainability. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, 2007, str. 7.

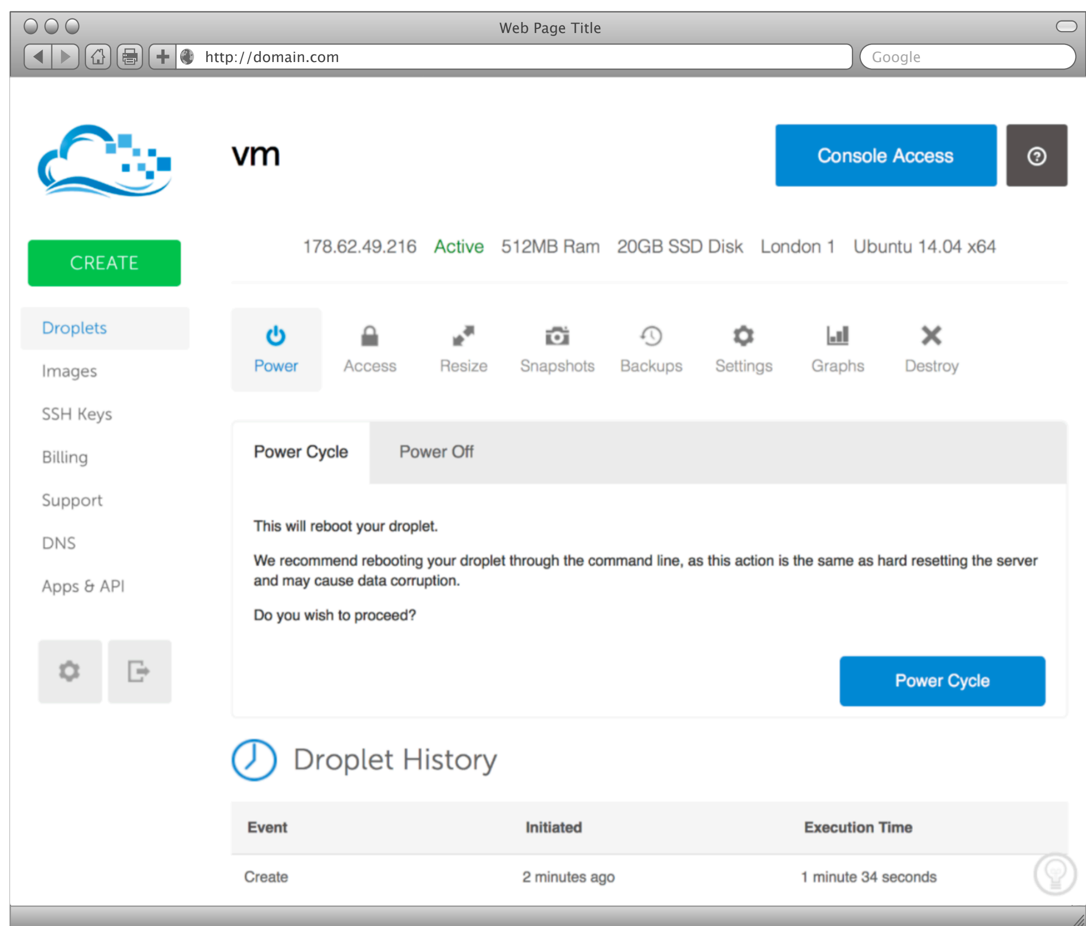
[19] Sahasrabudhe, S. S.; Sonawani, S. S.: Comparing openstack and vmware. In *Advances in Electronics, Computers and Communications (ICAECC), 2014 International Conference on*, IEEE, 2014, s. 1–4.

[20] Sefraoui, O.; Aissaoui, M.; Eleuldj, M.: OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, ročník 55, č. 3, 2012: s. 38–42.

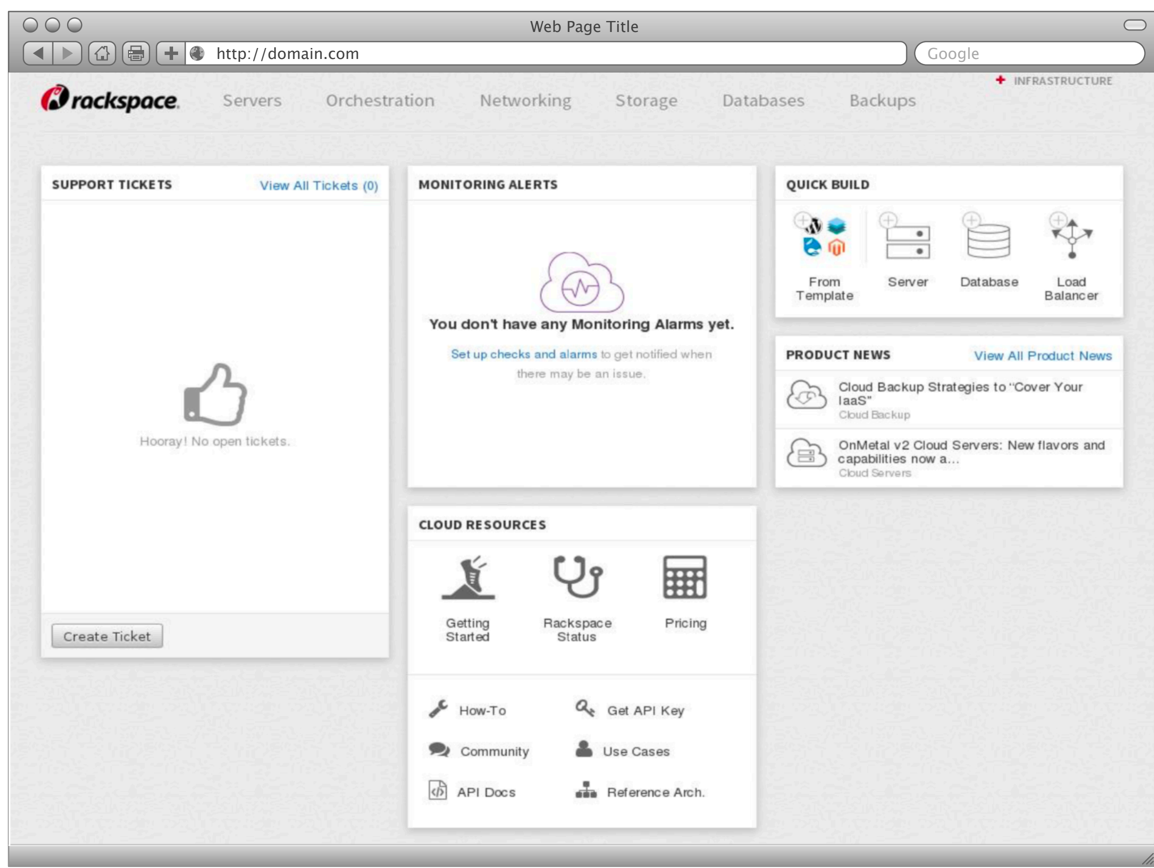
[21] Shon, P.: What is a Virtual Private Server? *Cedera.com*, 2014, [Online; navštíveno 24.11.2018].
URL https://www.credera.com/blog/technology-insights/java/redis-explained-5-minutes-less/

Príloha A

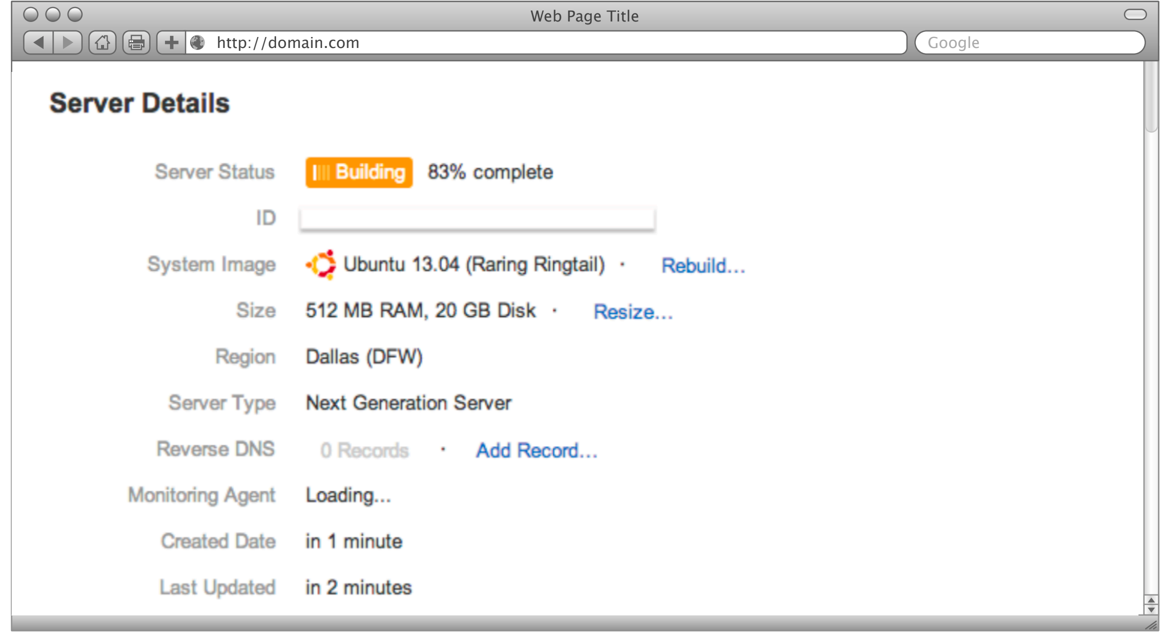
Konkurenčné systémy



Obr. A.1: Detail serveru na portály Digital Ocean



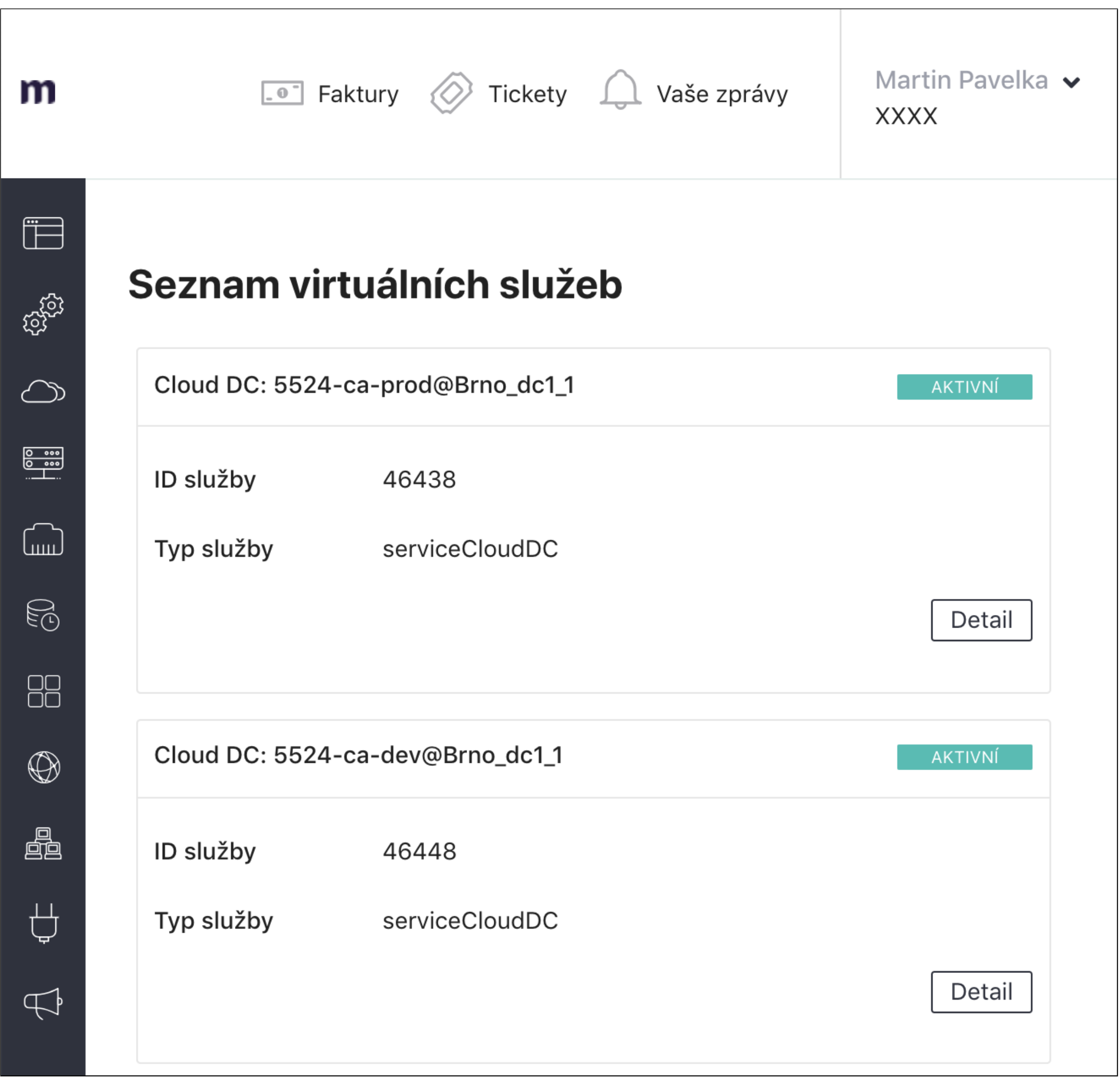
Obr. A.2: Zobrazenie dashboardu v systéme Rackspace



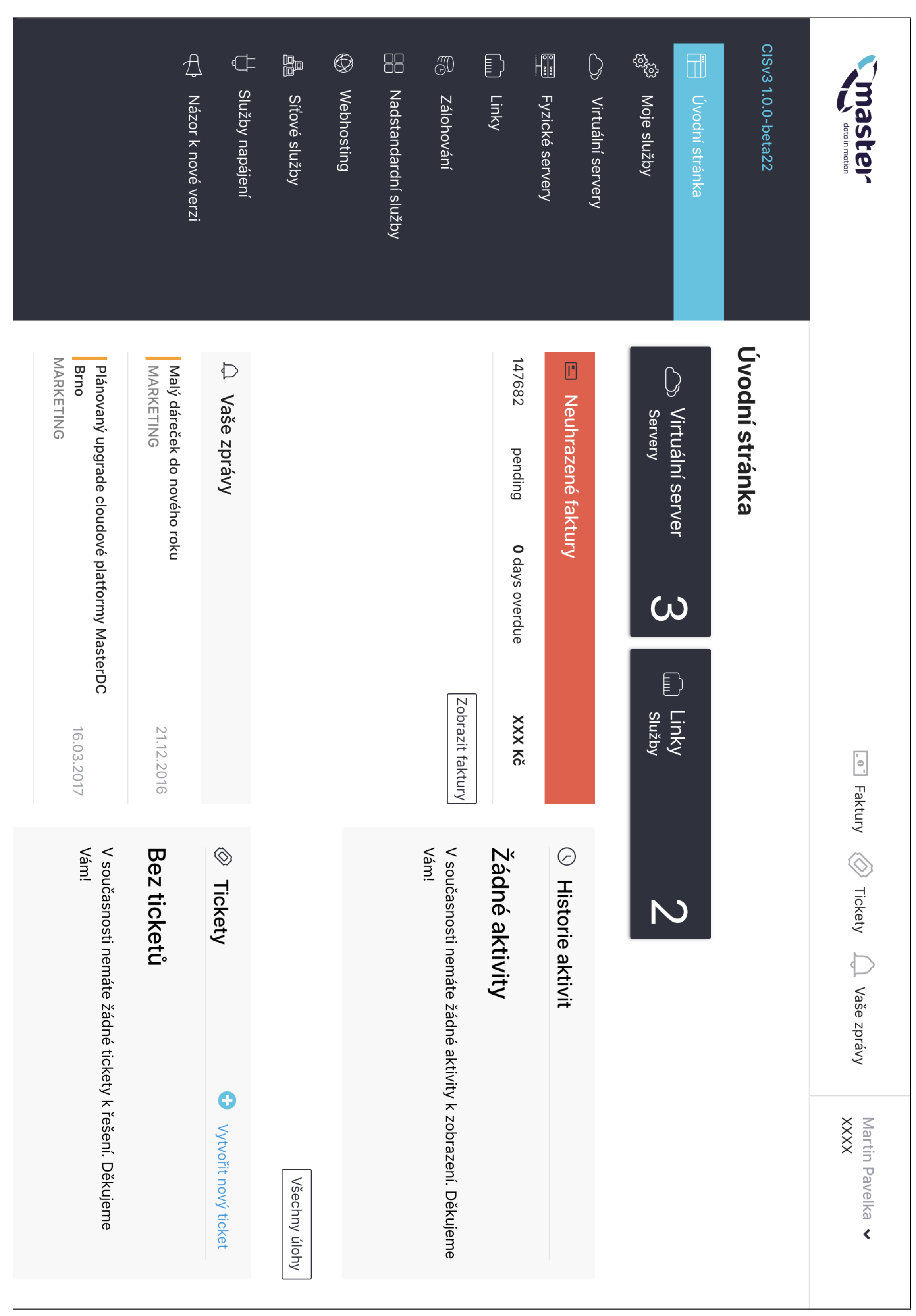
Obr. A.3: Detail serveru systému Rackspace

Príloha B

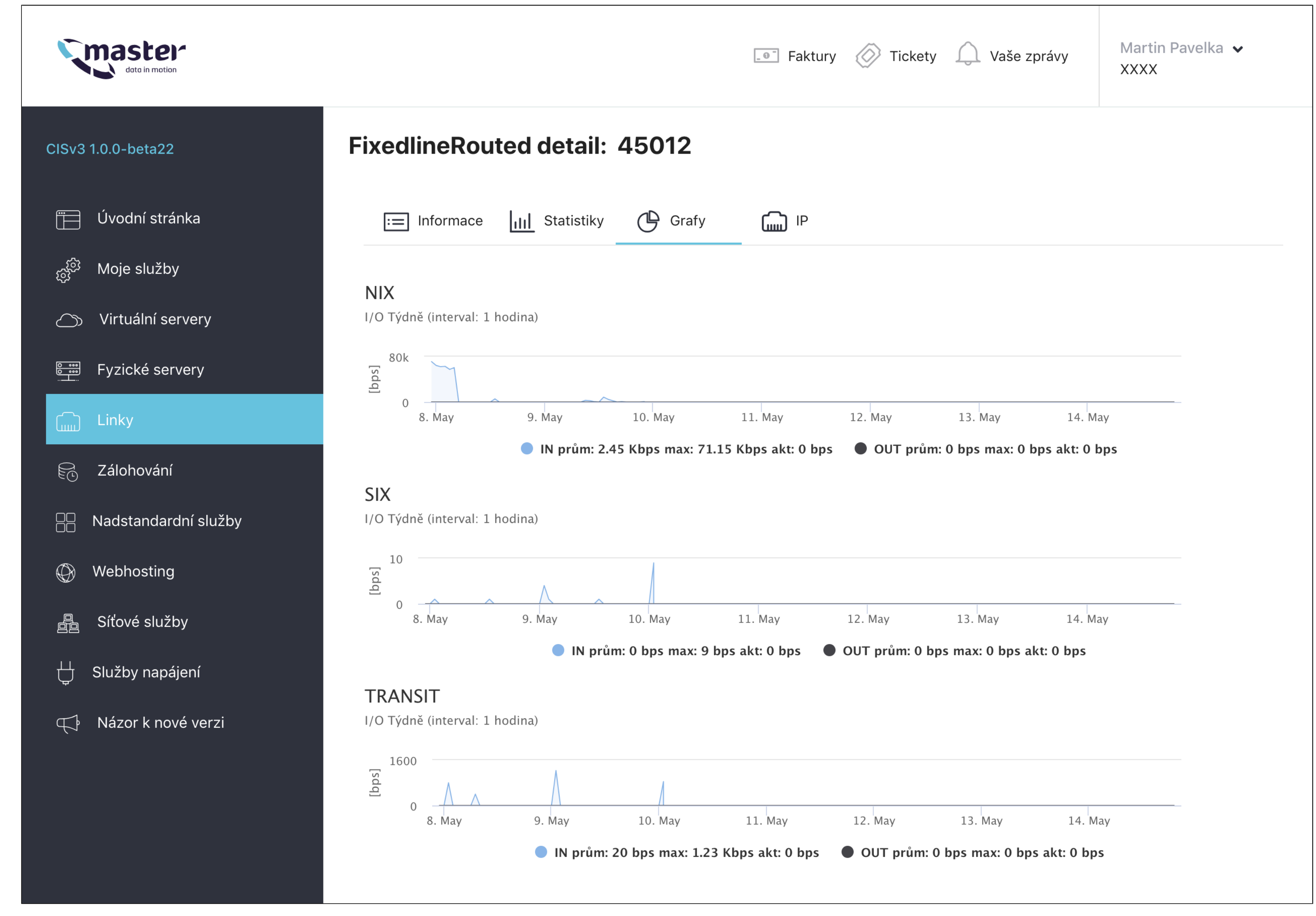
Ukážky webovej aplikácie



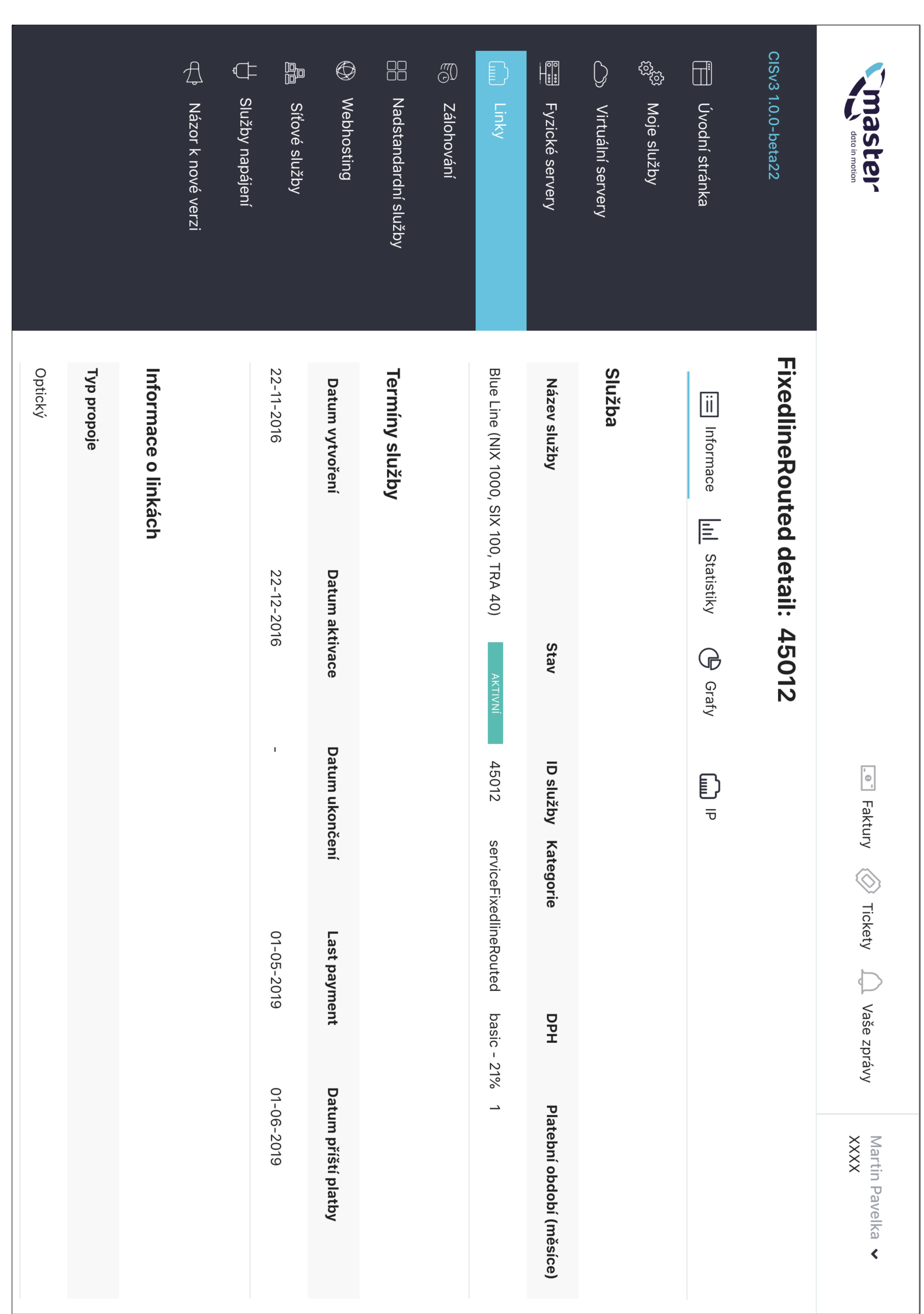
Obr. B.1: Responzívne zobrazenie stránky pri službách kategórie



Obr. B.2: Dashboard s informačnými panelmi

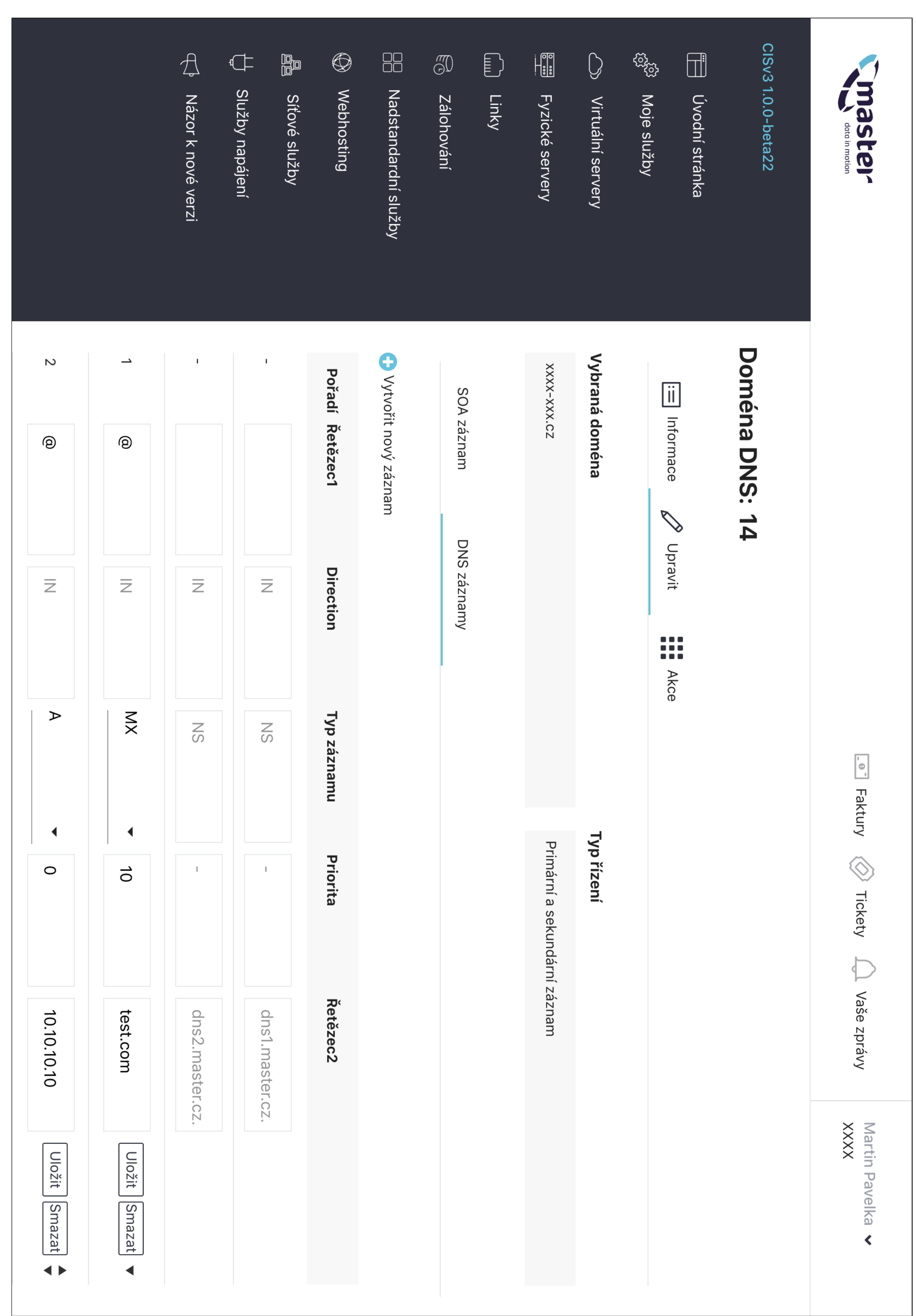


Obr. B.3: Grafy zobrazené u služby routovanej linky

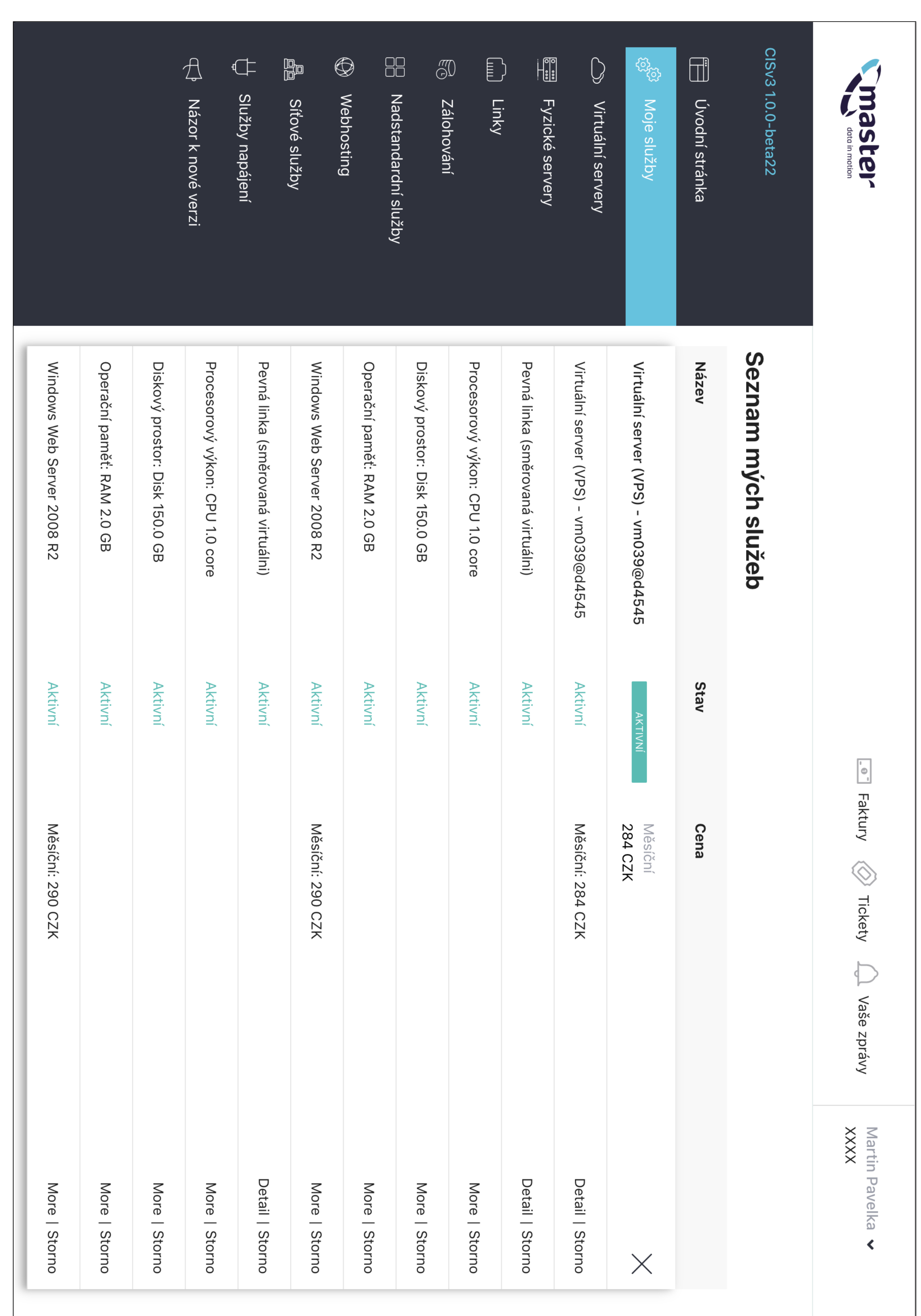


Obr. B.4: Konkrétny detail služby routovanej linky

Obr. B.5: Zobrazenie zoznamu faktúr a informačnej notifikácie



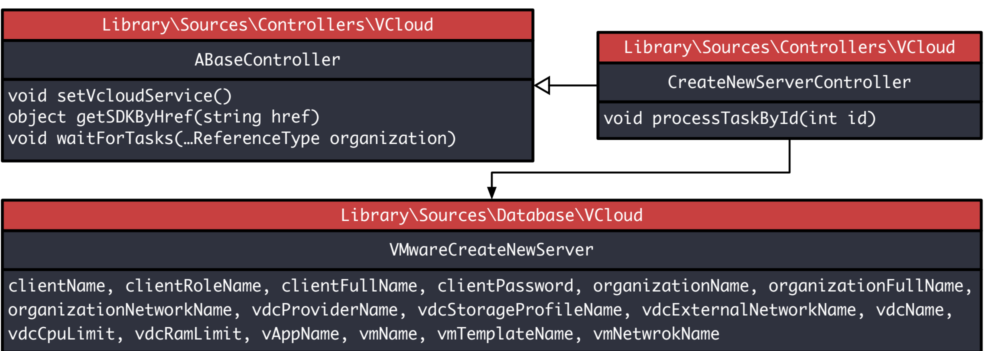
Obr. B.6: Príklad editácie služby - úprava DNS záznamov domény



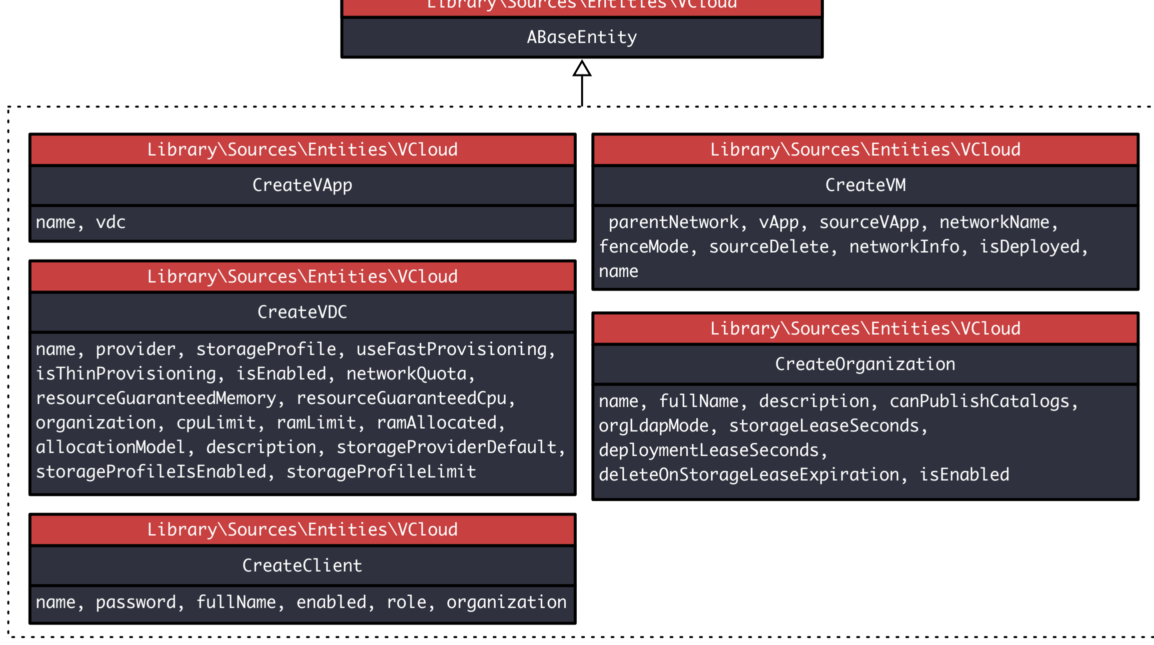
Obr. B.7: Strom zobrazujúci všetky služby zákazníka

Príloha C

OOP vizualizácia automatizácie



Obr. C.1: Objektový model kontrolérov systému VMware



Obr. C.2: Objektový model entít systému VMware

Library\Sources\Models\VCloud

ABaseModel

const string serviceCacheKey
VMware_VCloud_SDK_Service vCloudService
VMware_VCloud_SDK_Admin vCloudAdmin
VMware_VCloud_SDK_Extension vCloudExtension
object getSDKByHref(string href)
…ReferenceType getReferenceByHref(string href, ?string name, ?string tag, ?string type)
array getResultsByQuery(string type, ?string fields, ?string filter)
array getReferencesByQuery(string type, ?string fields, ?string filter)

Library\Sources\Models\VCloud

Organization

…AdminOrgType getOrganizationRecordByName(string name)
…ReferenceType getOrganizationReferenceByName(string name)
…AdminOrgType createOrganizationFromEntity(CreateOrganization cloudOrganization)
…ReferenceType getNetworkRefByName(…ReferenceType organization, string name)

Library\Sources\Models\VCloud

VApp

…QueryResultAdminVAppRecordType getVAppRecordByName(string name)
…ReferenceType getVAppReferenceByName(string name)
…VAppType createVAppFromEntity(CreateVApp cloudVApp)
void setOwnerForVApp(…ReferenceType vApp, …ReferenceType client)

Library\Sources\Models\VCloud

Client

…ProviderVdcType getProviderRecordByName(string name)
…ReferenceType getProviderReferenceByName(string name)
…ReferenceType getStorageProfileReferenceByName(string name, …ReferenceType provider)

Library\Sources\Models\VCloud

VDC

public getVDCRecordByName(string name) : …QueryResultAdminVdcRecordType
public getVDCReferenceByName(string name): …ReferenceType
public createVDCFromEntity(CreateVDC cloudVDC) : …AdminVdcType
public setNetworkForVDC(SetVDCNetwork cloudVDCNetwork) : …OrgVdcNetworkType

Library\Sources\Models\VCloud

Client

…QueryResultVAppTemplateRecordType getTemplateRecordByName(string name)
…ReferenceType getTemplateReferenceByName(string name)
…ReferenceType getVAppByTemplate(…ReferenceType template)

Library\Sources\Models\VCloud

ExternalNetwork

…Extension_VMwareExternalNetworkType getExternalNetworkRecordByName(string name)
…ReferenceType getExternalNetworkReferenceByName(string name)

Library\Sources\Models\VCloud

Role

…RoleType getRoleRecordByName(string name)
…ReferenceType getRoleReferenceByName(string name)

Library\Sources\Models\VCloud

Client

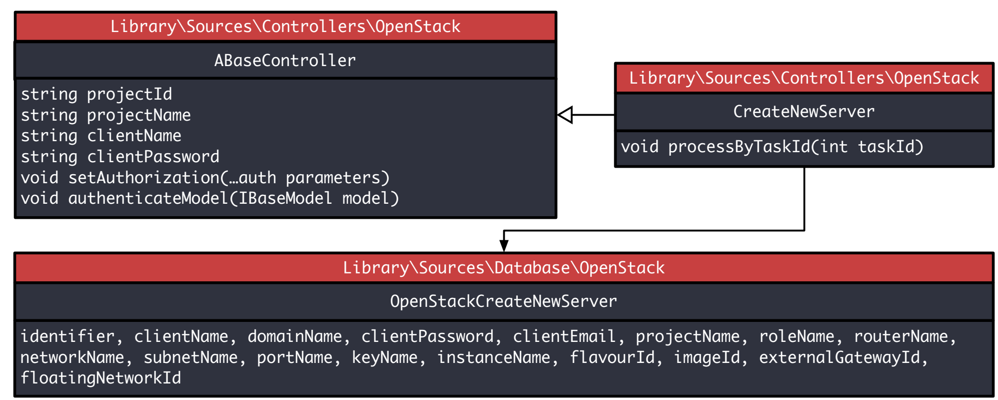
…TaskType createVmFromEntity(CreateVM cloudVm)

Library\Sources\Models\VCloud

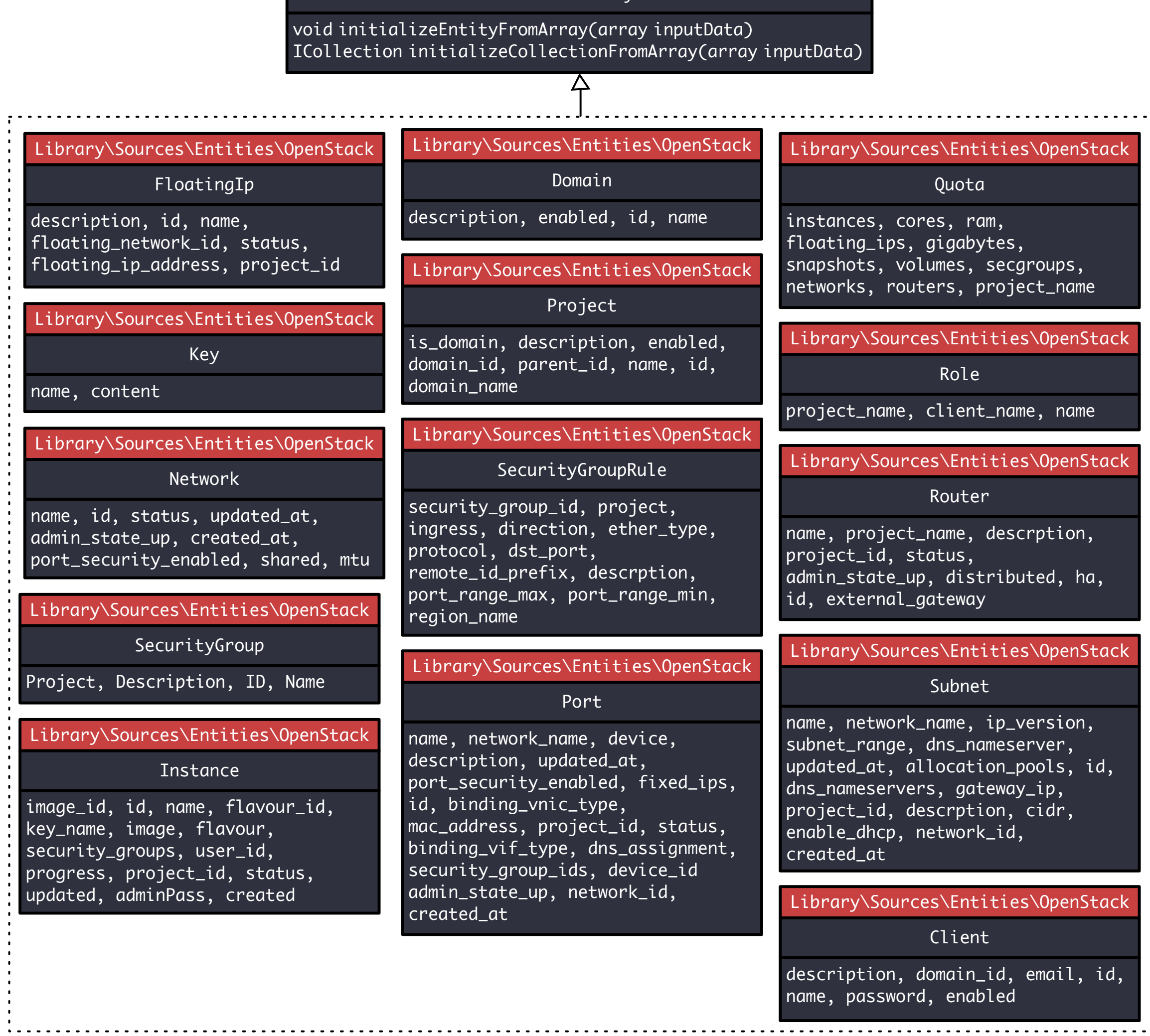
Client

…QueryResultAdminUserRecordType getClientRecordByName(string name)
…ReferenceType getClientReferenceByName(string name)
…UserType createClientFromEntity(CreateClient cloudClient)

Obr. C.3: Objektový model modelov systému VMware

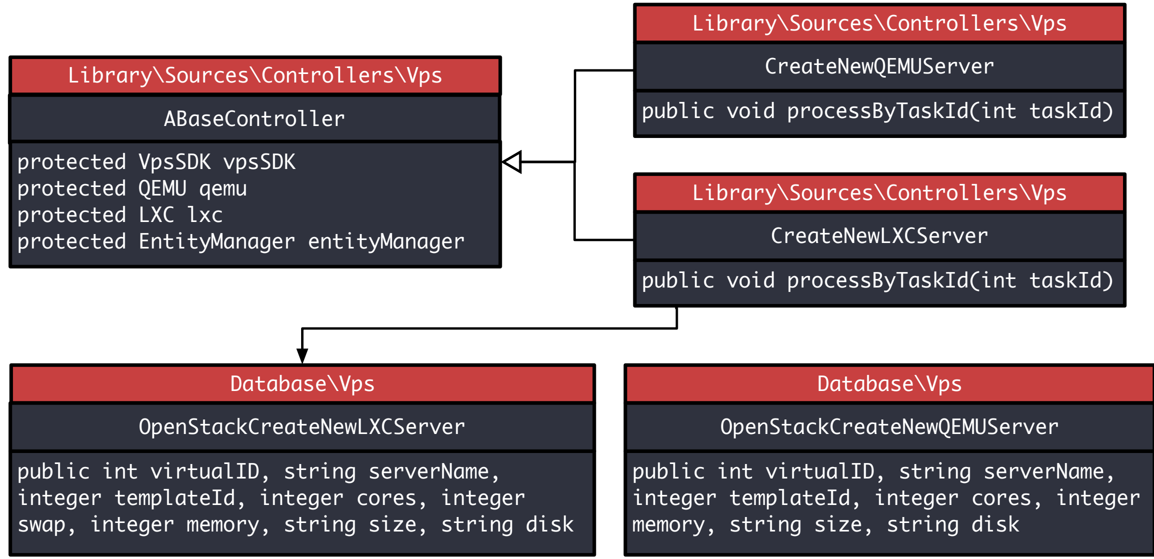


Obr. C.4: Objektový model kontrolérov OpenStack systému

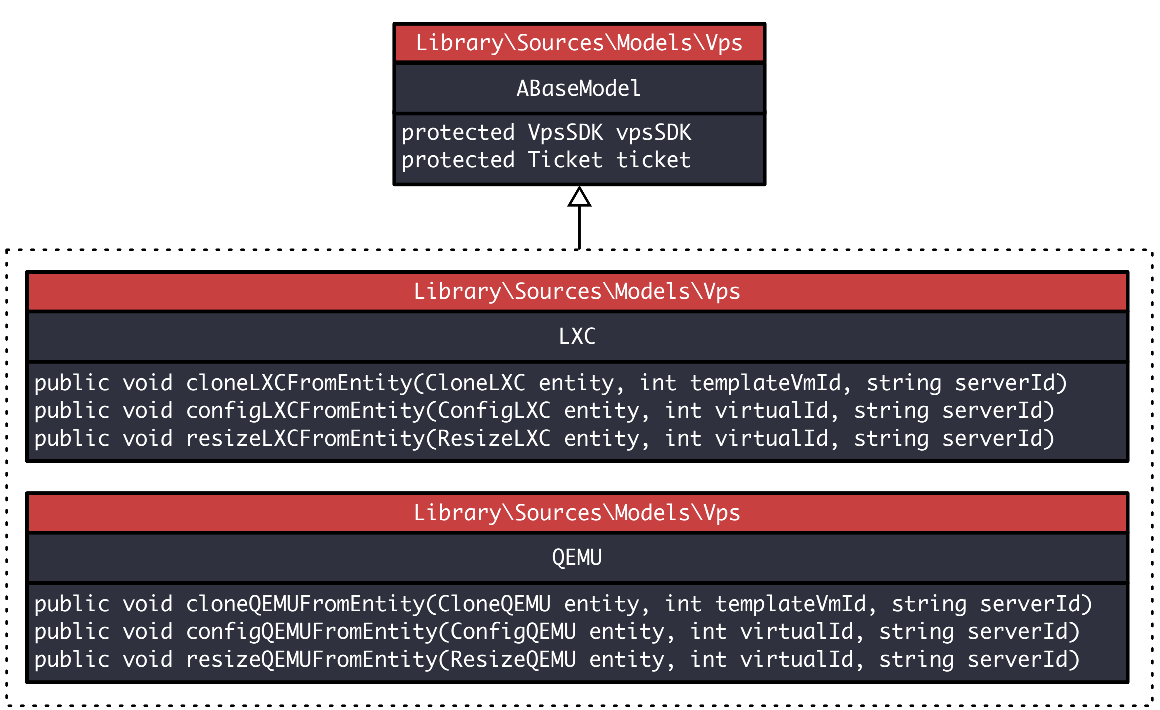


Obr. C.5: Objektový model entít systému OpenStack

Obr. C.6: Objektový model modelov systému OpenStack



Obr. C.7: Objektový model kontrolérov systému Proxmox



Obr. C.8: Objektový model modelov systému Proxmox

Obr. C.9: Objektový model entít systému Proxmox

Príloha D

Zoznam služieb systému

Tabuľka D.1: Prehľad služieb implementovaných v systéme

Služba	Popis
serviceAutonomicSystem	Služba autonómneho systému
serviceBackup	Služba externého zálohovania
serviceBreaker	Služba ističov
serviceCircuit	Služba L2 okruhu
serviceCloudComponent	Komponenta cloudového serveru
serviceCloudDC	Server OpenStack cloudovej virtualizácie
serviceCloudHardware	Fyzický server Mikrotik alebo Dell
serviceCloudVLAN	Vyhradená sieťová VLAN pre zákazníka
serviceControlPanel	Prenájom software pre ovládanie ISP panelu
serviceDataDispatcher	Monitoring a merač datových prenosov
serviceDDoSProtection	Anti DDoS ochrana Radware
serviceDextra	Dodatkové služby k dedikovanému balíku
serviceDNS	Služba DNS záznamov domén
serviceDomain	Registrácia domény u poskytovateľa
serviceDualBreaker	Služba duálneho fyzického ističa
serviceFextra	Dodatočné služby ku konektivite alebo linke
serviceFextraHousing	Vyvedenie linky na strechu
serviceFixedline	Služba obyčajnej linky
serviceFixedlineAverage	Služba obyčajnej linky - účtovaný priemer
serviceFixedlineBGPData	Linka protokolu BGP
serviceFixedlineData	Služba datovej linky

serviceFixedlineFract	Služba delenej linky
serviceFixedlineRouted	Služba smerovanej linky
serviceFixedlineRoutedData	Služba smerovanej datovej linky
serviceFixedlineRoutedFract	Služba smerovanej delenej linky
serviceFixedlineRoutedVirtual	Služba virtuálnej smerovanej linky
serviceFixedlineTops	Služba obyčanej linky - účtovaný strop
serviceHardware	Virtuálny server technológie VMware
serviceHWextra	Nešpecifikované hardwarové služby
serviceIPs	Zoznam IP adries pri službe
serviceMailServer	Poskytovanie mail servera
serviceMasterDisk	Úložisko vo virtuálnom cloudovom úložisku
serviceMobileApplicationsLicence	Licencia na mobilnú aplikáciu
serviceOversight	Služba monitoringu Icinga
servicePackage	Balík obsahujúci viacero služieb
servicePeering	Služba MPLS okruhu
servicePerIPAccounting	Štatistiky jednotlivých IP adries
servicePower	Extra odber energie nad limit
servicePowerMeter	Služba merania spotreby energie
servicePowerSupply	Prenájom napájania v racku
serviceSoftwareLicence	Licencia na nešpecifikovaný software
serviceSpace	Priestor v racku pre housing
serviceSsl	Správa SSL certifikátov
serviceSupport	Administrátorká podpora pre služby
serviceVextra	Doplňkové služby k virtálnym serverom
serviceVirtual	Webhosting
serviceVirtualComponent	Komponenta virtuálneho VPS servera
serviceVirtualDedicated	Virtuálny privátny server - VPS
serviceVMComMonthly	Zálohovanie VMware - mesačná platba
serviceVMComPerpetual	Zálohovanie VMware - neobmedzené
serviceOther	Nešpecifikovaná služba
serviceConnection	Napríklad optické prepojenie
serviceDedicated	Balík služieb fyzických serverov

Príloha E

Príklady volaní API

```
curl -X GET 'apiAddress/vmware/v-mware-create-new-server?clientId=2433' \
  -H 'Content-Type: application/json'
```

Výpis E.1: Získanie požiadaviek klienta 2433 na nový server VMware

```
curl -X POST 'apiAddress/vmware/v-mware-create-new-server' \
  -H 'Content-Type: application/json' \
  -d '{
    "clientName" : "dlxko", "clientRoleName" : "Org Administrator",
    "clientFullName": "dlxko", "clientPassword": "dlxko",
    "organizationName": "dlxko", "organizationNetworkName" : "internet",
    "organizationFullName" : "dlxko", "vAppName" : "dlxko",
    "vdcExternalNetworkName" : "VLAN2004 - cloud Brno DC4",
    "vdcProviderName" : "Brno DC4 Entry", "vdcStorageProfileName" : "Entry",
    "vdcName" : "dlxko", "vdcCpuLimit" : 2048, "vdcRamLimit" : 1024,
    "vmName" : "dlxko", "vmTemplateName" : "Ubuntu16LTS",
    "vmNetworkName" : "internet", "clientId" : 2434, "progress" : 0}'
```

Výpis E.2: Vytvorenie požiadavky na VMware server

```
curl -X GET 'apiAddress/openstack/open-stack-create-new-server?clientId=2433' \
  -H 'Content-Type: application/json'
```

Výpis E.3: Získanie požiadaviek klienta 2433 na nový server OpenStack

```
curl -X POST 'apiAddress/openstack/open-stack-create-new-server' \
  -H 'Content-Type: application/json' \
  -d '{
    "clientName" : "dlxko", "clientEmail" : "pavelka@master.cz",
    "instanceName" : "dlxko", "keyName" : "dlxko", "progress" : 0,
    "networkName" : "dlxko", "portName" : "dlxko", "clientId" : 2433,
    "externalGatewayId" : "cf84295e-fc1a-42a8-8025-c987c6f2d8b8",
    "floatingNetworkId" : "cf84295e-fc1a-42a8-8025-c987c6f2d8b8",
    "clientPassword": "dlxko", "domainName": "dlxko",
    "flavourId" : "cf7861a9-2055-44e1-9e96-e12ec1f1cd27",
    "imageId" : "690f7807-45ce-4b63-b148-b9c60f7bc2a7",
    "projectName" : "dlxko", "roleName" : "_member_",
    "routerName" : "dlxko", "subnetName" : "dlxko"}'
```

Výpis E.4: Vytvorenie požiadavky na OpenStack server

```
curl -X GET \
  'apiAddress/vps/vps-create-new-lxc-server?clientId=2433' \
  -H 'Content-Type: application/json'
```

Výpis E.5: Získanie požiadaviek klienta 2433 na nový server LXC

```
curl -X POST \
  apiAddress/vps/vps-create-new-lxc-server \
  -H 'Content-Type: application/json' \
  -d '{
    "cores" : 2, "disk" : "rootfs",
    "memory": 2048, "serverName": "d4422",
    "size" : "15G", "swap" : 1024,
    "templateId" : 101, "virtualId" : 123,
    "clientId" : 2433, "progress" : 0
  }'
```

Výpis E.6: Vytvorenie požiadavky na LXC server

```
curl -X GET \
  'apiAddress/vps/vps-create-new-qemu-server?clientId=2433' \
  -H 'Content-Type: application/json'
```

Výpis E.7: Získanie požiadaviek klienta 2433 na nový server QEMU

```
curl -X POST \
  apiAddress/vps/vps-create-new-qemu-server \
  -H 'Content-Type: application/json' \
  -d '{
    "cores" : 2, "disk" : "rootfs",
    "memory": 2048, "serverName": "d4422",
    "size" : "15G", "templateId" : 101,
    "virtualId" : 123, "clientId" : 2433,
    "progress" : 0
  }'
```

Výpis E.8: Vytvorenie požiadavky na QEMU server

Príloha F

Expirácia Redis záznamov

Tabuľka F.1: Expirácia položiek v dočasnej pamäti.
asap - 60 sekúnd, **workday** - 4 hodiny

Typ volania	Doba platnosti
Prihlasovacie údaje užívateľa	**asap**
Nastavenia profilu, spoločnosti a support hesla	**asap**
Stav portov fyzických serverov	**asap**
Stiahnutie doménového súboru	**asap**
Zoznam a detaily tiketov	**asap**
Denné, DDoS, IP a napájacie štatistiky	**hour**
Štatistiky poskytnutej podpory	**hour**
Zoznam portov služby linky u grafov	**deň**
Zoznam smerov služby linky u grafov	**deň**
Správa kreditných kariet	**workday**
Zoznam služieb a zobrazenie stromu	**workday**
Zobrazenie detailu služby	**workday**
Zoznam emailových adries klienta	**workday**
Všetky faktúry, detail faktúry, nezaplatené položky	**23:59:59**
Správy a novinky	**deň**
Správa oprávnených osôb	**deň**
Mesačné štatistiky štatistiky	**deň**
Technické informácie u služieb	**month**
Zoznam lokalizácii systému	**month**

Príloha G

Inštalácia a konfigurácia systému

G.1 Závislosti

- PHP 7.3.* (viď. 3.1)
- Redis 3.2.* (viď. 3.7)
- Yarn 1.* (viď. 3.5)
- Composer 1.7.* (viď. 3.5)
- SCSS 3.5.* (viď. 3.3)
- TypeScript 3.3.* (viď. 3.4)
- Webpack 4.29.* (viď. 3.6)
- Babel 6.23.* (viď. 3.10)
- Httpd 2.4.*
- Varnish 6.*
- Grafana 6.1.* (viď. 9.4.3)
- Prometheus 2.5.* (viď. 9.4.3)
- Node exporter 0.17.* (viď. 9.4.3)
- Varnish exporter 1.4.* (viď. 9.4.3)
- Stylint 10.* (viď. 3.3)

G.2 Inštalácia

Postup inštalácie aplikácie popisuje súbor *.gitlab-ci.yml*
Postup inštalácie systému automatizácie a API

```
composer install
php ./library/bin/Console.php orm:schema-tool:create
```

Príloha H

Obsah pamäťového média

- Zdrojové súbory k webovej aplikácii
  `application`
- Produkčná verzia aplikácie s knižnicami
  `applicationDistribution.7z`
- Zdrojové súbory k systému automatickej tvorby serverov a API
  `automaticSystemDistribution.7z`
- Produkčná verzia systému automatickej tvorby serverov a API
  `automaticSystem`
- Latex verzia diplomovej práce
  `thesis`
- PDF verzia diplomovej práce
  `thesis.pdf`
- Konfiguračný súbor dashboardu Grafana
  `grafana.json`
- Konfiguračný súbor Varnish
  `varnish.conf`
- Konfiguračný súbor Apache
  `httpd.conf`