České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Ondřej Janata

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Implementace systému pro správu instalací webových služeb především v prostředí cloudu

Pokyny pro vypracování:

Prostudujte současný stav a standardy software pro správu cloudu. Navrhněte řešení pro automatickou správu (vytvoření, monitoring, upgrade, zrušení) instancí webové služby v prostředí cloudu též s možností snadného spuštění instance na dedikovaném serveru mimo cloud. Toto řešení implementujte.

Seznam odborné literatury:

[1] FURHT Borko, Escalante Armando, Handbook of cloud computing. Springer Science + Business Media, LLC, 2010. 634 stran. ISBN 978-1-4419-6524-0.
[2] ANTHONY T. Velte, Toby J.Velte, R. Elsenpeter. Cloud computing praktický průvodce. Computer press, a.s., 2011. 344 stran. ISBN 978-80-251-3333-0.

Vedoucí: Ing. Pavel Troller CSc.

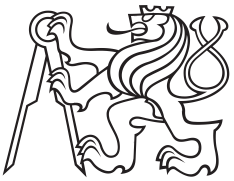
Platnost zadání do konce letního semestru 2017/2018

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 4.10.2016

Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická

Katedra počítačů

Implementace systému pro správu instancí webových služeb především v prostředí cloudu

Bc. Ondřej Janata

Květen 2017

Poděkování / Prohlášení

Chtěl bych poděkovat vedoucímu práce Ing. Pavlu Trollerovi CSc. za jeho čas věnovaný konzultacím a korektuře. Dále bych chtěl poděkovat startupu Factorify za možnost zpracovat toto téma, zvláště pak Bc. Jakubovi Petrovi.

V neposlední řadě patří velké díky rodině, přítelkyni a kamarádům , kteří mi byli v době psaní této práce oporou.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací [1].

V Praze dne 23. 5. 2017

.....

Abstrakt / Abstract

Prostudujte současný stav a standardy software pro správu cloudu. Navrhněte řešení pro automatickou správu (vytvoření, monitoring, upgrade, zrušení) instancí webové služby v prostředí cloudu též s možností snadého spuštění instance na dedikovaném serveru mimo cloud. Toto řešení implementujte.

Klíčová slova: správa cloudu; Docker; cloud platforma; Spring Boot; modulární služby; on-premise nasazení;

Study current state and trends in cloud management software. Design solution for automatic management (creation, monitoring, upgrade and cancellation) of the web service instance in the cloud. This solution can easily run on the dedicated server outside cloud. Implement this solution.

Keywords: cloud management; Docker; cloud platform; Spring Boot; modular services; on-premise deployment;

Title translation: Implementing system solution for managing instances of web services primarily in a cloud environment

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

Tato práce vznikla na základě podnětu a za podpory startupu *Factorify*, který vyvýjí stejnojmenný systém pro plánování a řízení, zejména, dávkové výroby. S rostoucím počtem klientů začalo přibývat prostředí, které bylo nutné spravovat. Prostředím se rozumí Jails[1] v operačním systému FreeBSD. Některým klientům bylo potřeba s každým novým nasazením dodávat seznam změn. Vzhledem k plánované expanzi byl tento přístup dlouhodobě neudržitelný, a proto vznikl interní projekt *Factorify Platform*. Ten má za úkol zjednodušit a co nejvíce zautomatizovat proces vytváření, správy a monitoringu klientských instalací jak v cloudu, tak on-premise. Dále ve stručnosti čtenáře seznámíme se systémem *Factorify*.

1.1 Factorify

Systém je architekturou klient-server. Backend je napsán v programovacím jazyce Java a využívá frameworku Spring Boot. Frontend je napsán v Javasciptu s využitím frameworku Angular. Komunikace mezi frontendem a backendem probíhá přes RESTful API. Datový formát je většinou JSON. Plánování výroby běží jako samostatná služba, která s *Factorify* komunikuje opět přes RESTful API.

Systém *Factorify* je napsán modulárně, aby bylo možné dodávat jednotlivým klientům specifickou funkcionalitu. Modularizace probíhá na úrovni build procesu, který je vykonáván nástrojem Gradle[2]. Proces nasazení je realizován jako build task v systému continuous integration (aktuálně TeamCity). Zkompilovaný JAR je nasazen na server přes *ssh*.

1.2 Softwarový projekt

Před samotnou diplomovou prací jsem v rámci předmětu *Softwarový nebo výzkumný projekt* zkoumal varianty cloudu. V rámci rešerše jsem prozkoumal jak veřejné cloudy[3], tak i systémy pro vytvoření vlastního cloudu, neboli privátního, například pomocí technologie Openstack[4], Openshift[5] nebo Kubernetes[6].

Během vytváření rešerše jsem zjistil, že trend kontejnerizace webových aplikací je velký a každý z trojice veřejných cloudů – Google, Amazon a Microsoft – již nabízí

[1] Rozšíření konceptu chroot. Lze brát jako virtualizaci na úrovni OS. `https://www.freebsd.org/doc/handbook/jails.html`
[2] https://gradle.org/
[3] Konkrétně Google Cloud Platform, AWS a Microsoft Azure
[4] Platforma pro vytvoření vlastní cloudové infrastruktury. Více: https://www.openstack.org/
[5] Platforma pro vytvoření vlastní cloudové infrastruktury, která dále řeší orchestraci, monitoring a volitelně celý proces nasazení.
[6] Cloudová platforma pro orchestraci, škálování a monitoring aplikací založených na Docker kontejnerech.

službu pro nasazení aplikace v podobě kontejnerů. Jedním z nejběžnějším formátů kontejneru se nepochybně stal Docker.

Jako jednu z největších výhod kontejneru osobně považuji možnost nasazení do jakékoliv služby, která podporuje danou technologii. Již žádná instalace potřebného SW, veškeré běhové prostředí je definováno uvnitř kontejneru. Další zásadní výhodou je možnost škálování pomocí spouštění nových instancí. Samozřejmě v prostředí cloudu nejsme limitováni fyzickým výkonem serveru a vyrovnávání zátěže může probíhat automaticky. Toto je zvlášť důležité pro služby, které jsou vytěžované dynamicky, což není náš případ.

1.3 Návaznost na *Factorify Platform*

Vytvořením platformy je základním stavebním kamenem pro vybudování systému konfigurovatelného samotnými uživateli. Myšlenka je taková, že konfiguraci jednotlivých serverů si budou v budoucnosti klienti z velké části vytvářet sami. Sloužit k tomu bude konfigurátor.

Konfigurátor bude jednoduchá webová aplikace, která provede uživatele procesem výroby v jejich firmě, pomůže jim s nastavením a dovolí naimportovat data specifická pro jejich výrobu. Výstupem konfigurátoru bude požadavek na RESTful API *Factorify Platform*, ve které se založí nový server a nakonfigurují se služby včetně potřebných modulů. Na přání klienta bude možné spustit i automatické vytvoření prostředí v cloudu. Zde bude klient moci pokračovat se zkoušením nastavené aplikace.

1.4 Řízení projektu

Vytvoření platformy pro *Factorify* jsem získal na starost osobně. Jelikož šlo o rozsáhlejší projekt, dohodli jsme se s vedením startupu, že práci rozdělíme mezi více lidí. Pro řízení projektu jsme zvolili agilní přístup.

Většinu frontendu programoval kolega Petr Skalička. Na této části jsem se podílel na návrhu obrazovek, poskytoval konzultace a zajišťoval kontrolu kódu. Mechanismus kontroly byl v podobě pull requestů na Bitbucketu[1] (viz sekce 5.1).

Já jsem měl na starosti rešerši řešení, návrh architektury platformy, naprogramování backendu *Factorify Platform* – Factorify Platform Manager (dále jen *FPM*) a dalších důležitých částí systému (viz sekce 4.8).

1.5 Architektura orientovaná na služby

V angličtině `Service Oriented Architecture` neboli SOA, což budeme používat dále v textu. SOA využívá několik, nejčastěji webových, služeb, které spolu vzájemnou spoluprací poskytují službu další. Služba se zde stává stavebním kamenem. Služba může, ale nemusí pro část své funkcionality využívat služeb dalších. Nejčastějším komunikačním protokolem je HTTP a formátem zpráv je buď XML nebo JSON.

[1] https://bitbucket.org/

Výhodou SOA je jejich nezávislost na použité technologii. Jediné, co musí služba umět, je vystavit webové API. V praxi to znamená, že uvnitř firmy můžeme udržovat několik systémů, které mohou být napsány v různých programovacích jazycích. Jednotlivé služby se mnohem lépe udržují, testují a také se lépe stanovuje odpovědnost. Dalším neméně důležitým kladem je znovupoužitelnost. Často se stává, že určitá funkcionalita je duplikována v několika službách. S využitím SOA a chytrého návrhu můžeme tuto společnou funkcionalitu vyjmout do samostatné služby.

Nevýhodou SOA je samozřejmě prvotní investice spojená s architekturu. Pokud firma funguje na monolitické aplikaci, pak musí dojít k rozpadu jednotlivých částí na služby. Mezi službami pak musí být vymezena vzájemná interakce. Dále musíme mít mechanismus jak spolu jednotlivé služby propojit. Variant je několik. Od jednoduché možnosti přiřadit IP adresu a port každé službě a udržovat globální index až po použití služby pro automatickou detekci služeb na místní síti, neboli `service discovery`.

Použití service discovery je důležitým prvkem pokud začneme škálovat jednotlivé služby. Poté do architektury musíme přidat zařízení pro vyvažování zátěže, neboli `load balancer`. Tento prvek musí znát adresy svých služeb, aby věděl mezi jaké body má rozkládat zátěž. Pokud škálování potřebujeme dělat dynamicky, statický globální index fungovat nebude. Nemáme jinou možnost, než zapojit do systému service discovery.

SOA budeme chtít využít i ve *Factorify Platform*, jelikož častým přáním zákazníků je dodat specifický systém potřebný pro jejich business. Prozatím se jednalo o konfigurátory produktů ale lze předpokládat, že do budoucna se množství požadavků bude zvyšovat.

Kapitola 2
Analýza

Tato kapitola se věnuje soupisu nashromážděných požadavků a definicí jednotlivých rolí.

2.1 Role

Zde definujeme role, které budou s *FPM* provádět interakci. Jedná se o následující 4 role:

2.1.1 FP agent

Dále budeme v textu označovat jako *FP Agent*. *FP Agent* komunikuje s *FPM* zabezpečeným kanálem formou zasílání zpráv. Využívá pro to technologii *Apache ActiveMQ*[1]. *FP Agent* je povinnou součástí všech klientských nasazení. Automaticky zasílá do platformy zprávu `heartbeat`, která spolu s informací, že je klient online, obsahuje také monitorovací data.

2.1.2 FP platforma

Dále budeme v textu označovat jako *FPM*. *FPM* rolí se rozumí samotný systém, který v pravidelných smyčkách kontroluje stavy určitých úloh. Například se jedná o kontrolu stavu spuštěných build úloh.

2.1.3 Uživatel systému Factorify

Jedná se o uživatele, který má uživatelský účet v systému *Factorify* a právo komunikovat se systémem *FPM*. Interakce ze strany této role je omezená na 2 úlohy:

- Vyžádat změny (changelog) mezi nasazenou a aktuální verzí aplikace.
- Naplánovat aktualizaci systému na konkrétní datum a čas.

2.1.4 Správce systému

Správcem sytému je některý ze zaměstnanců *Factorify*, který se stará o administraci *Factorify Platform*. Uživatel se ověřuje přihlášením pomocí uživatelského jména a hesla.

[1] `http://activemq.apache.org/`

2.2 Nefunkční požadavky

Nefunkční požadavky poskytují soupis důležitých aspektů systému, které by měly být ve valné většině splněny. Požadavky jsou:

- *Factorify Platform* bude založena na open-source technologiích.
- Jednotlivé služby (aplikace) musí být modulární.
- *Factorify Platform* musí umožňovat jednoduché vydávání nových verzí jednotlivých modulů.
- *FPM* bude napsán v Kotlinu[1]
- Použité technologie musí být udržované[2] a jejich vývoj aktivní[3]. Rozhodnutí zda technologii použít, či nepoužít záleží na individuálním posouzení. Snažíme se ale držet uvedených kritérií jak je to jen možné.
- Infrastruktura – datové centrum
  - Fyzické umístění musí být v regionu Střední Evropy a páteřní sít do Prahy musí mít propustnost alespoň 40 Gb/s.
  - Cena za měsíční provoz nejméně výkonné instance musí být do 1000 Kč bez DPH za měsíc.
  - Výkon instance musí být možné v čase změnit.
  - Připojení do internetu musí mít garantované pásmo minimálně 100 Mb/s.
- Zabezpečení
  - Klient bude zabezpečen podepsaným certifikátem.
  - Uživatelská hesla budou v databázi uložena šifrovaně.
  - Přístup k repositářům pro klienty bude zabezpečen pomocí SSL.
- *Factorify Platform* by měla podporovat nasazení služeb založených na různých technologiích. Snaha je se nevázat pouze na platformu JVM.

2.3 Funkční požadavky

Jak je možné si povšimnout, nepodporujeme mazání entit. Je to záměrné rozhodnutí. V některých případech používáme k označení neaktivní entity příznak `active`. Požadavky jsou vypsány v bodech a to pro snazší čitelnost čtenáře.

- Dashboard – monitoring
- Klient
  - Výpis (včetně seznamu aktivních serverů)
  - Detail – vytvoření, zobrazení, uložení
- Uživatel
  - Výpis
  - Detail – vytvoření, zobrazení, uložení (změna hesla)
  - Přihlášení
  - Odhlášení

[1] Staticky typovaný programovací jazyk pro JVM. Více: https://kotlinlang.org/
[2] Reakce na kritické bugy do týdne
[3] Commit v repositáři za poslední 3 měsíce

- Docker
  - Výpis
  - Detail – vytvoření, zobrazení, uložení
  - Výpis verzí modulu
  - Spuštění vytvoření nové verze
- Server
  - Výpis
  - Detail – vytvoření, zobrazení, uložení
  - Stažení logů s určitou maskou
  - Zobrazení monitorovacích dat za poslední hodinu
  - Spuštění serveru v cloudu
  - V případě výskytu alertu zašle email
- Stránkovaný výpis alertů
- Naplánovat aktualizaci systému na konkrétní datum a čas

Kapitola 3
Rešerše

Tato kapitola se zabývá rozborem problémů, které jsme uvedli v předchozích kapitolách a poskytuje možné návrhy řešení. Vybrané řešení pak naleznete spolu s detailním zdůvodněním v následující kapitole.

3.1 Možnosti modularizace

Modularizaci uvažujeme na úrovni jednotlivých služeb. Pro každou službu chceme volitelně určit množinu dostupných modulů, které lze posléze aktivovat jednotlivým serverům. Z požadavků víme (viz sekce 2.2), že služby bychom nechtěli vázat na konkrétní technologii ale zároveň musíme podporovat Javu, ve kterém je napsán systém *Factorify*. Dále uvádíme jednotlivé možnosti řešení.

3.1.1 JAR balíčky

Pokud se omezíme pouze na prostředí JVM, modularizaci můžeme jednoduše udělat tím, že na classpath [2] přidáme kód. To můžeme udělat například zakompilováním specifického kódu pro každého klienta anebo lépe pomocí distribuce funkcionality v jednotlivým JAR balíčcích, u kterých musíme zajistit, že budou umístěné na classpath. Vytváření balíčku bychom pak dělali na základě bussines logiky.

Komplikace řešení spočívá v tom, že budeme muset začít řešit správu balíčků. Jednoduchým příkladem může být například potřeba nainstalovat balíček `prodej`, který ale pro korektní fungování potřebuje balíček `sklad`. Pro správný chod by tedy musela vzniknout komponenta pro správu komponent, která by tuto správu závislostí řešila. Pokud bychom chtěli ještě umožnit určovat verze aktivních komponent, pak by komponenta pro správu komponent musela umět vyřešit i toto, což není zcela triviální problém.

Pokud použijeme framework, který provádí nastavení skenováním konfiguračních souborů při startu aplikace, jako např. Spring Boot, tak po přidání či aktualizaci balíčku musíme aplikaci restartovat.

3.1.2 Capsule

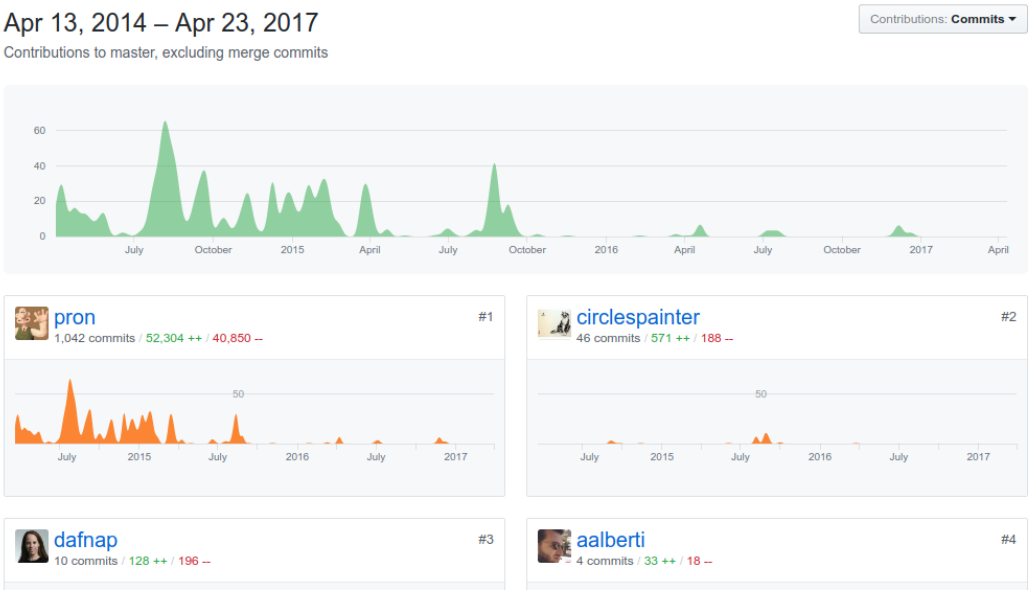
Pokud se omezíme pouze na technologii Java, tak zajímavým řešením modularizace je projekt Capsule[1]. Výstižný popis nám nabízí sám autor Capsule na svých stránkách: „One way of thinking about a Capsule is as a fat JAR on steroids. Just as a build tool manages your build, Capsule manages the launching of your application.“ [3]. Ve stručnosti to znamená, že Capsule za nás dokáže vyřešit:

- „Fat JAR“ – archiv, který v sobě má veškeré závislosti a metadata potřebné k běhu.

[1] `http://www.capsule.io/`

- Automatické stažení závislostí.
- Automatický update závislostí.
- Umožňuje definovat platformě závislou konfiguraci pomocí specifického zaváděcího skriptu.
- Představuje jednotku nasazení aplikace.
- Umí provozovat architekturu orientovanou na služby – nasazení WAR archivů do Jetty [4].
- Umožňuje izolovat aplikaci uvnitř LXC kontejneru [4].
- Nabízí rozšiřitelnost pomocí modulů (tzv. Capsletů).

V první chvíli byl tento nástroj velkým kandidátem na vítěze. Bohužel by nás ale připravil o možnost nasazovat služby fungující mimo JVM. Dalším citelným problémem byla aktivita vývoje projektu, kde jsme zjistili, že za poslední 3 měsíce nebyl proveden žádný commit a že většina kódu pochází od jednoho vývojáře.



Obrázek 3.1. Příspěvky kódu do projektu Capsule [5]

Modularizace systému je jedním ze stavebních kamenů *Factorify Platform*. Znatelným rizikem do budoucna je skutečnost, že projekt udržuje pouze jeden vývojář.

3.1.3 Specifický build instance

Specifický build představuje jeden z nejběžnějších přístupů k řešení problému modularizace. Většina vyspělejších nástrojů pro build zdrojového kódu jako je např. Ant[1], Gradle[2] či Maven[3] umožňují uzpůsobit výstup dle vstupní parametrizace. Pro následující odstavce uvažujme, že kód není rozdělen například do Maven artefaktů, ale je umístěn v jednom repositáři.

Výhodou tohoto řešení je, že klient dostane výslednou aplikaci uzpůsobenou jeho potřebám. Nemá žádnou starost ani volbu si určitá rozšíření zapínat či vypínat. Nalezneme i řadu další výhod, které by na první pohled nemusely být patrné. Konkrétně:

[1] `http://ant.apache.org/`
[2] `https://gradle.org/`
[3] `https://maven.apache.org/`

- Odhalení chyb vzniklých během kompilace.
- Vyšší pravděpodobnost odchycení běhových chyb (testujeme a vyvýjíme aplikace v konkrétní konfiguraci).
- Možnost provádět optimalizace jelikož známe všechny komponenty systému.
- Sdílené knihovny jsou na disku pouze jednou (v případě modulů může být knihovna k dispozici několikrát, v nejhorším případě i v podobě nekompatibilních verzí).
- Obfuskace celého řešení, které je hůře prolomitelné než v případě obfuskovaných modulů.
- Snazší provádění code review – recenzent má lepší představu o změnách (vidí je všechny).

Žádná varianta nepřináší pouze výhody. Shrňme si dále tedy podstatné problémy tohoto řešení, kterých není rozhodně málo:

- Je nutná infrastruktura pro build a distribuci řešení.
- Údžba parametrizace build skriptu pro jednotlivé klienty.
- Při změně jedné části je nutný build celého řešení.
- Může svádět k vytváření nemodulárního kódu a zbytečné duplikaci.
- Přináší vyšší nároky na HW při práci s velkou kódovou základnou.
- Stanovení zodpovědnosti za konkrétní modul.

Pokud provedeme shrnutí, tak se dá říci, že tato varianta je použitelná za předpokladu, že si dokážeme opodstatnit vznik nutné infrastruktury pro build jednotlivých klientů.

3.1.4 Docker

Docker je dnes velmi populární a používaná technologie pro kontejnerizaci aplikace. Co to vlastně kontejner je? Ve zkratce by se dalo říci, že se jedná o virtualizaci na úrovni OS [6]. Kontejner uvnitř sebe zařídí izolaci prostředí (programy, služby, souborový systém) a dovolí nám také omezit systémové prostředky. Všechny kontejnery využívají pro běh společné jádro systému.

Základem technologie Docker je obraz, který se vytváří pomocí tzv. `Dockerfile` [8]. Každý řádek znamená jeden příkaz, který definuje vlastní vrstvu. Pomocí souboru tedy nadefinujeme potřebné prostředí pro běh aplikace. Také určíme zdrojové kódy, které se mají nahrát do distribuce. Dále se nadefinuje, které porty docker obraz vystavuje a jaký proces se má spustit při startu kontejneru. Obrazy oblíbených aplikací, převážně těch webových, jsou k dispozici v Docker registrech. Odtud je možné je stáhnout jednoduchým příkazem:

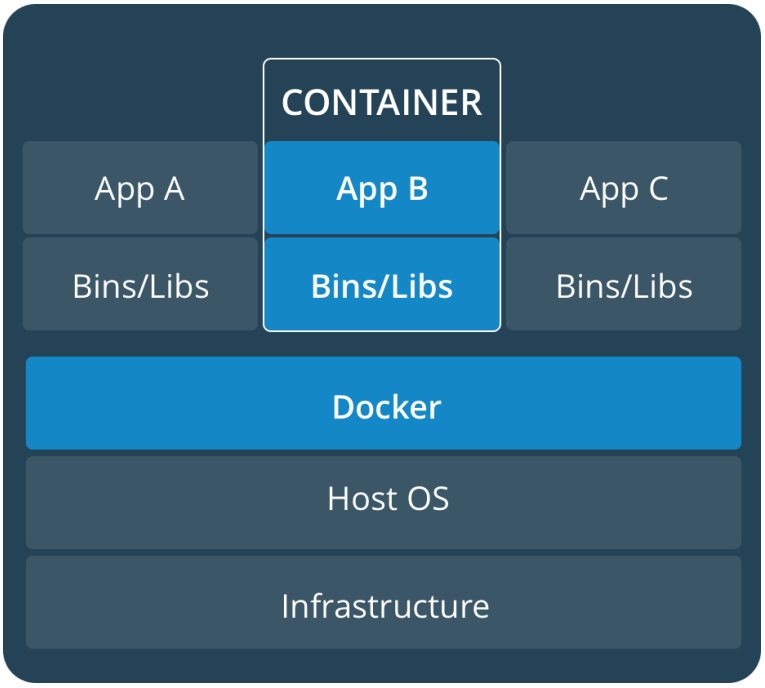
```
docker pull <název obrazu>
```

a následně spustit:

```
docker run -d --name <název služby> <název obrazu>
```

Níže poskytujeme ukázku `Dockerfile` pro jednu z aplikací *Factorify Platform*.

- `FROM` – určuje výchozí kontejner, který se použije jako základ.
- `LABEL` – přidá metadata, primárně slouží k organizaci obrazů.



Obrázek 3.2. Diagram Dockeru [7]

- `RUN` – spustí příkaz. Změny, které jsou jím způsobeny jsou poté uloženy uvnitř obrazu. Slouží pro doinstalování potřebných programů, knihoven, ke konfiguraci služeb, či k nastavení souborového systému a jeho práv.
- `COPY` – zkopíruje soubory z lokálního souborového systému do obrazu.
- `WORKDIR` – nastaví pracovní adresář uvnitř obrazu pro příkazy `COPY`, `ADD`, `RUN`, `CMD` a `ENTRYPOINT`.
- `EXPOSE` – určí, které porty bude daný kontejner vystavovat. Při spuštění kontejneru je poté nutné ještě explicitně určit mapování portů, jinak nebude možné na port zvenčí přistoupit.

```
FROM phusion/baseimage:0.9.21
LABEL maintainer 'Ondrej Janata (janata@factorify.cz)'

RUN add-apt-repository -y ppa:certbot/certbot
RUN apt-get update
RUN apt-get install -y wget openjdk-8-jre certbot

COPY docker/default-keystore.p12 /usr/local/fpm/default/keystore.p12

RUN mkdir -p /etc/my_init.d
COPY docker/init/* /etc/my_init.d/
RUN chmod -R +x /etc/my_init.d/

WORKDIR /usr/local/fpm/service/fp-gateway
COPY build/libs/fp-gateway-1.0.0-SNAPSHOT.jar fp-gateway.jar
RUN sh -c 'touch fp-gateway.jar'

COPY docker/cron_tab/* /etc/cron.d/
RUN chmod +x /etc/cron.d/*
```

```
COPY docker/cron/* ./
RUN chmod +x *.sh

COPY docker/service/gateway /etc/service/gateway
RUN chmod -R +x /etc/service/*

EXPOSE 443 80
```

Doporučení Dockeru je, aby uvnitř kontejneru běžela jen jedna aplikace/služba. Výše uvedený obraz to nesplňuje a to z důvodu, aby bylo možné využít systémové služby jako je CRON. Konkrétně `phusion/baseimage` [9] je oblíbeným obrazem, který poskytuje velice minimální systém s několika základními systémovými službami – init, cron, syslog.

Kontejnery spolu mohou komunikovat po lokální síti, do které jsou přiděleny. Pokud není žádná specifikována, pak jsou umístěny ve výchozí. Kontejner je na síti dostupný pod svým jménem, které je následně přeloženo na IP adresu. Funguje zde jednoduché interní DNS.

Naším požadavkem je abychom mohli provozovat klienty i on-premise na Linux a Windows systémech. Dlouhou dobu byl Docker k dispozici pouze pro Linux a ostatní OS jako MacOS a Windows poskytovaly Docker skrz virtuální stroj, na kterém byl nainstalován Linux.

Aktuálně (květen 2017) je možné Docker na Windows provozovat již nativně [10]. Řešení je na stránkách Dockeru pojmenováno *Docker for Windows*. Požadavkem je zapnutá podpora virtualizační technologie Hyper-V a nainstalovaný systém Windows 10 ve verzi 64bit Pro, Enterprise nebo Education.

3.2 Orchestrace

Wikipedia definuje orchestraci následovně: „Orchestrace popisuje automatickou koordinaci a řízení komplexních počítačových systémů, middleware a služeb. Orchestrace zabezpečuje koordinaci procesů a výměnu informací pomocí webových služeb.“ [11].

3.2.1 Skripty

Základní variantou může být použití skriptů. Můžeme předpokládat, že jsme ve světě Unixu, takže se budeme bavit o *shell* nebo *bash* skriptech. Pomocí nich můžeme spustit množinu služeb, které potřebujeme pro naši aplikaci. Jednotlivé služby se budou muset umět vypořádat se stavem, že některá jiná služba není k dispozici. To není velký problém. Je třeba s tím počítat při vývoji aplikace. Jednoduché řešení spočívá např. v přidání čekací smyčky.

Dalším problémem, který je třeba vyřešit, je opětovné spuštění služby po jejím pádu. Opět můžeme řešit sami anebo použít již léty prověřený mechanismus služeb OS. Pokud jednotlivé aplikace spustíme jako služby OS, pak máme zaručeno, že po neočekávaném pádu budou automaticky znovu spuštěny.

Nevýhodou je, že na všechna prostředí se musí jednotlivé služby nastavit. Nejčastěji to znamená napsat skript, který obsahuje kód pro spuštění, vypnutí a restart služby. Dále je pak třeba službu povolit. To se např. v OS FreeBSD zařídí přidáním následujícího kódu do souboru `rc.conf`, kde *service* je název služby. Mechanismus je silně závislý na OS.

```
<service>_enable="TRUE"
```

3.2.2 Puppet

Populární nástroj pro deklarativní konfiguraci prostředí. Umožňuje nám pomocí konfiguračního souboru určit uživatele systému, software, konfiguraci či běžící služby. Jedná se o architekturu *klient-server*. Na každý node, který má být součástí infrastruktury konfigurované přes Puppet, musí mít nainstalovaného agenta. Agent periodicky zjišťuje, zda server nemá k dispozici novější konfiguraci. Pokud ano, tak agent pomocí potřebných příkazů převede prostředí do požadovaného stavu. Komunikace probíhá zabezpečeně pomocí SSL certifikátu.

Puppet je uvolněn jako open-source. Nabízena je také verze *Enterprise*, která nabízí mnoho vylepšení. Jmenujme jen některé z nich: reporting Puppet serveru, API pro orchestraci, uživatelské role či podpora 24/7.

Dále uvádíme ukázku konfigurace, která nám na daném prostředí zajistí běžící web server Apache2 jako službu a existujícího uživatele *backup*, který má jako shell nastavený `/bash/bash`.

```
# Vytvoř uživatele www
user { 'backup':
   ensure     => present,
   uid        => '1000',
   gid        => '1000',
   shell      => '/bin/bash',
   home       => '/home/backup'
}

# Nainstaluj webový server Apache2
package { 'apache2':
   ensure => installed,
}

# Zajisti běžící Apache2 jako službu
service { 'apache2':
   ensure => running,
}
```

3.2.3 Docker Compose

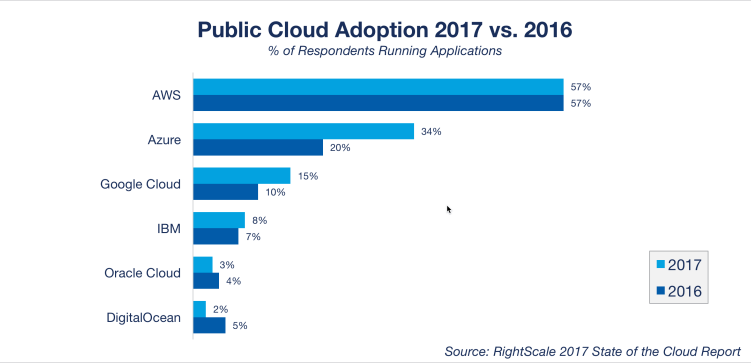
Samotný Docker je skvělý nástroj pro běh jedné služby uvnitř kontejneru. Co když ale potřebujeme nastartovat aplikaci, která má architekturu orientovanou na služby? Přesně pro tyto potřeby vznikl nástroj Docker Compose, který nám umožňuje pomocí jednoduchého konfiguračního souboru `docker-compose.yml` popsat proces spuštění potřebných kontejnerů aplikace, neboli služeb.

Docker Compose automaticky přidělí všechny služby do nové sítě. Dále dovoluje definovat pojmenovaný svazek (anglicky volume), který slouží pro sdílení perzistentních dat mezi kontejnery. Mezi důležitá nastavení pak také patří konfigurace vystavených portů, vlastních oddílů kontejneru či chování při neočekávaném ukončení služby, kde nejčastěji využijeme možnost automatického startu.

3.3 Nasazení do cloudu

V aktuální sekci se podíváme na možnosti nasazení aplikace do cloudu. V rešerši jsou zahrnuta pouze řešení, kde jako vývojáři dostáváme minimálně infrastrukturu jako službu (`IaaS`). Výběr dále omezíme na veřejný cloud, který má k dispozici datové centrum v regionu střední Evropy.

Důležitým ukazatelem kvality nabízených služeb je i počet produkčních nasazení. Pro představu čtenáře přikládáme výsledek ankety využití jednotlivých cloudů provedenou společností RightScale mezi 1000 IT profesionály. 40% respondentů představují profesionálové velkých korporací s více jak 1000 zaměstnanci.



Obrázek 3.3. Graf využití public cloudu [12]

Pro firmu existuje řada důvodů, proč upřednostnit cloud před nákupem a správou vlastního HW. Shrňme ty nejdůležitější [13]:

- Zajištěná vysoká úroveň dostupnosti služby. SLA dosahuje běžně 99.9%.
- Provoz v cloudu nevyžaduje investice do nákupu a modernizace HW.
- Cloud umožňuje dynamicky upravovat výkon tak, abychom zajistili dostupnost služby pro jakýkoliv počet uživatelů.

AWS[1], Google Cloud Platform[2] i Azure[3] má k dispozici řešení pro nasazení aplikace v kontejnerech. Základem je vytvoření clusteru z několika nodů (standardní VPS), do kterého se následně nasazují kontejnery. Výhoda řešení spočívá ve snadném škálování pomocí přidávání či ubírání nodů a kontejnerů. Vyvažování zátěže mezi více instancí kontejnerů poté zařizuje sama platforma. My bychom ale potřebovali v cloudu provozovat několik stejných kontejnerů, které by ale byly specifické určitému klientovi. Tyto kontejnery by byly vzájemně izolovány. Během průzkumu jsme nenalezli jednoduchý mechanismus, jak toto nakonfigurovat. Z tohoto důvodu jsme se nakonec rozhodli nasazovat klienta vždy na jeden VPS. To nám poskytne i mnoho výhod:

- Výkonu instance nasavujeme dle požadavků klienta.
- Možnost škálovat výkon instance.
- Stejné nasazení je jak pro cloud, tak i pro on-premise.
- Překvapivě dosáhneme nižší ceny.

[1] https://aws.amazon.com/
[2] https://cloud.google.com
[3] https://azure.microsoft.com/cs-cz/

- Fyzicky izolujeme prostředí klientů.

Existují samozřejmě řešení jako je OpenStack, které by nám dokázaly virtuální infrastrukturu vybudovat na vlastním HW. Jako startup ale nemáme kapacitu ani dostatečné znalosti na to danou službu provozovat. Naší filozofií je dodávat kvalitní software a soustředit naše kapacity do vývoje a vzdělávání.

V příloze je k nahlédnutí detailně rozepsaný odhad nákladů na provoz klienta v jednotlivých cloudech. V nákladech je započtena výpočetní síla pro hosting aplikace a výpočetní síla pro provoz databázového serveru. Nejlépe vychází varianta od OVH, kde nasazení jednoho klienta by měsíčně přišlo na necelých 1000 Kč vč. DPH.

3.3.1 Microsoft Azure

Jak název napovídá, jedná so o celosvětový cloud provozovaný softwarovým gigantem Microsoft, který aktuálně ve velkém expanduje a otevírá nová datová centra nejen v Evropě. Vzrůstající trend je patrný i z předchozího grafu. Azure nabízí skvělé geografické pokrytí a širokou škálu služeb. Nejbližší datové centrum se nachází ve Frankfurtu. Bohužel je to jeden z nejdražších poskytovatelů cloudu. Pro naši potřebu nás bude zajímat nabídka virtuálních serverů. Zde je výběr rozsáhlý. Varianty jsou členěny dle předpokládaného využití do několika kategorií:

- Obecné použití
- Výpočtově optimalizované
- Paměťově optimalizované

Pro potřebu kalkulace jsme vybrali pro klientské VPS instanci z kategorie „Obecné použití“ – *A2*. Pro sdílený databázový server poté instanci z kategorie „Paměťově optimalizované“ – *D13 v2*. Detailní popis viz tabulka 3.1.

Instance	A2	D13 v2
vCPU	2	8
RAM	3,5	56
HDD	60	400
Cena/měsíc [USD]	55,8	441

Tabulka 3.1. Specifikace instancí Azure

3.3.2 AWS

AWS je prozatím nejoblíbenějším veřejným cloudem, což je dle našeho odhadu způsobeno zejména faktem, že z velké trojice (Azure, Google Cloud Platform) je tu nejdéle. Nejbližší datové centrum je k dispozici také ve Frankfurtu. Z velké trojice AWS nabízí nejpřijatelnější ceny pro hodinovou kalkulaci. Pokud si můžeme instance předplatit na rok či déle dopředu, dokážeme výslednou sumu srazit i na polovinu.

Výhodou AWS je již zmiňovaná cena, široké celosvětové pokrytí a také technologie zakládané na otevřeném zdrojovém kódu. To nám umožňuje menší závislost na celé AWS platformě v případě, že bychom chtěli migrovat jinam. Dokumentace k jednotlivým službám je na velmi vysoké úrovni a krom textu je k dispozici i mnoho instruktážních videí. Stinnou stránkou je dle nás nepříliš zdařilé uživatelské rozhraní.

Dále je třeba počítat s poplatkem za přenesená data. Poplatek není velký[1], ale v případě aplikace intenzivní na přenos po síti může dodatečný poplatek přesáhnout i samotnou cenu VPS.

Pro potřebu kalkulace jsme vybrali pro klientské VPS instanci *t2.medium*. Pro sdílený databázový server poté instanci *c3.4xlarge*. Detailní popis viz tabulka 3.2.

Instance	t2.medium	c3.4xlarge
vCPU	2	16
RAM	4	30
HDD	50	2x160
Cena/měsíc [USD]	42	459

Tabulka 3.2. Specifikace instancí AWS

3.3.3 Google Cloud Platform

Google svůj cloud spustil nejpozději a to koncem roku 2011. Jak je zvykem této společnosti, tak kvalita je na prvním místě. Nejbližší datové centrum je v Belgii, v blízké budoucnosti vznikne také další ve Frankfurtu. Výhodou Google je přehledně zpracované administrační rozhraní a rozsáhlá, kvalitně zpracovaná dokumentace.

Také Google fakturuje přenesené GB dat[2]. Částka opět není astronomická, ale je o mnoho vyšší než v případě AWS.

Pro potřebu kalkulace jsme vybrali pro klientské VPS zákazníkem konfigurovatelnou instanci *Custom 2-4*. Pro sdílený databázový server poté instanci *n1-standard-16*. Detailní popis viz tabulka 3.3.

Instance	Custom 2-4	n1-standard-16
vCPU	2	16
RAM	4	60
HDD	50	400
Cena/měsíc [USD]	58	495

Tabulka 3.3. Specifikace instancí Google Cloud Platform

3.3.4 VSHosting s.r.o.

VS Hosting, je oproti ostatním variantám, pouze malá firma s jedním datovým centrum umístěným v Praze. S firmou jsme byli jednat po doporučení získaného od jednoho z investorů, se kterým jsme jako startup jednali. VS Hosting se zabývá zejména poskytováním infrastruktury a správou serverů klientů.

Ve svém portfoliu mají také cloud servery. Tato služba běží v pozadí na řešení OpenStack. Při porovnání cen s OVH je jejich nabídka více než předražená. My jsme s VS Hostingem měli osobní jednání, kde jsme poptávali infrastrukturu a případné možné platformní řešení založené na Dockeru. Výsledná cenová kalkulace pro cloud o kapacitě předpokládaných 30 klientů by stála necelých 34 tis. Při přepočtu na klienta to není vůbec špatná cena. Bohužel je třeba uvést, že řešení by bylo postavené na

[1] 0.12 USD/GB
[2] 0.17 USD/GB

dedikovaných strojích, takže bychom zpočátku platili provoz několika serverů, které by běžely naprázdno.

Od škálovatelné varianty založené na virtuálních strojích jsme byli odrazeni z výkonnostních důvodu. Očekávali jsme, že nebude problém se domluvit na vystavení API pro rozběh dodatečných VPS. VS Hosting ale tuto možnost nenabízí a ani ji neplánuje. Veškerý proces je alespoň částečně manuálně zpracován některým z administrátorů.

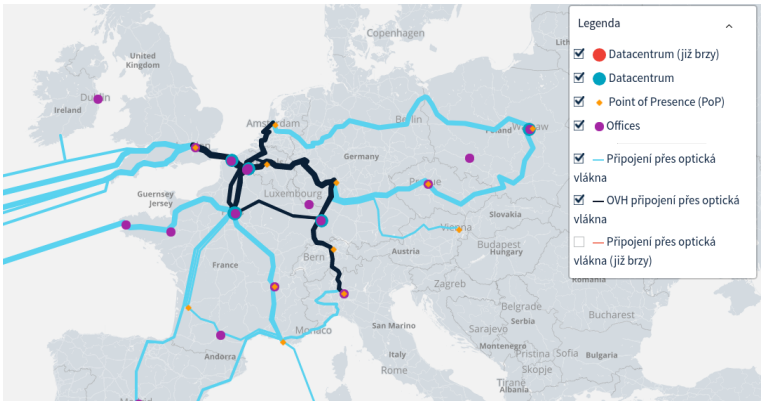
3.3.5 OVH

Francouzská firma, která se zabývá webhostingem, provozem a správou dedikovaných, virtuálních a nově i cloud serverů založených na technologii OpenStack.

Mezi hlavní výhody patří rozsáhlá optická síť vybudovaná po Evropě. OVH patří se 700 000 zákazníky ke špičce v oblasti poskytování webhostingu. Oproti konkurenci se OVH nesnaží nabízet bohatou paletu cloudových služeb jako AWS, Google Cloud Platform či Azure. Díky tomu mohou být experty v poskytování infrastruktury a virtuálních serverů a tyto služby nabízet za bezkonkurenční ceny. Například instanci, kterou dále uvádíme pro použití na klientské VPS stojí 540 Kč bez DPH, tedy nějakých 22 USD. Když si to porovnáme s konkurencí, jsme zhruba na polovině nejlevnější nabídky.

Jako jediný, z námi uvedených, velký poskytovatel nezpoplatňuje OVH přenesená data. Navíc garantovaná konektivita jednotlivých VPS do internetu je minimálně 100 Mbit/s, v případě výkonnějších instancí pak celých 250 Mbit/s.

OVH má aktuálně nejbližší datové centrum ve Varšavě, což je pro naše klienty z Moravy skvělá poloha. V případě potřeby komunikace je možnost se obrátit na kanceláře firmy přímo v Praze.



Obrázek 3.4. Síť OVH [14]

Pro potřebu kalkulace jsme vybrali pro klientské VPS zákazníkem konfigurovatelnou instanci *B2-7*. Pro sdílený databázový server poté instanci *EG-60*. Detailní popis viz tabulka 3.4.

Instance	B2-7	EG-60
vCPU	2	16
RAM	7	60
HDD	50	1600
Cena/měsíc [USD]	22	175

Tabulka 3.4. Specifikace instancí OVH

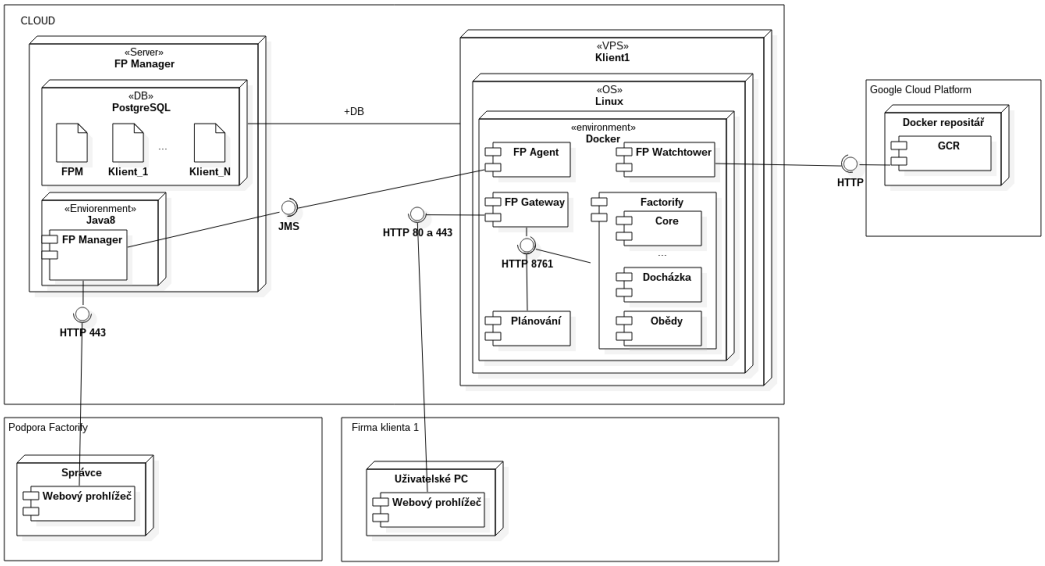
Kapitola 4
Návrh

Tato kapitola se zabývá návrhem. Jsou zde popsány jednotlivé technologie, které byly v jednotlivých částech projektu použity. Následuje popis architektury systému. Zde se zaměříme na komunikaci mezi jednotlivými subsystémy, popíšeme si způsob perzistence dat a podíváme se na uspořádání služeb v Dockeru.

Rozebereme si také jednotlivé subsystémy platformy, které jsou v základu tři – gateway, agent a watchtower. Návrh zakončíme přehledem podpůrných služeb, na které je platforma integrována.

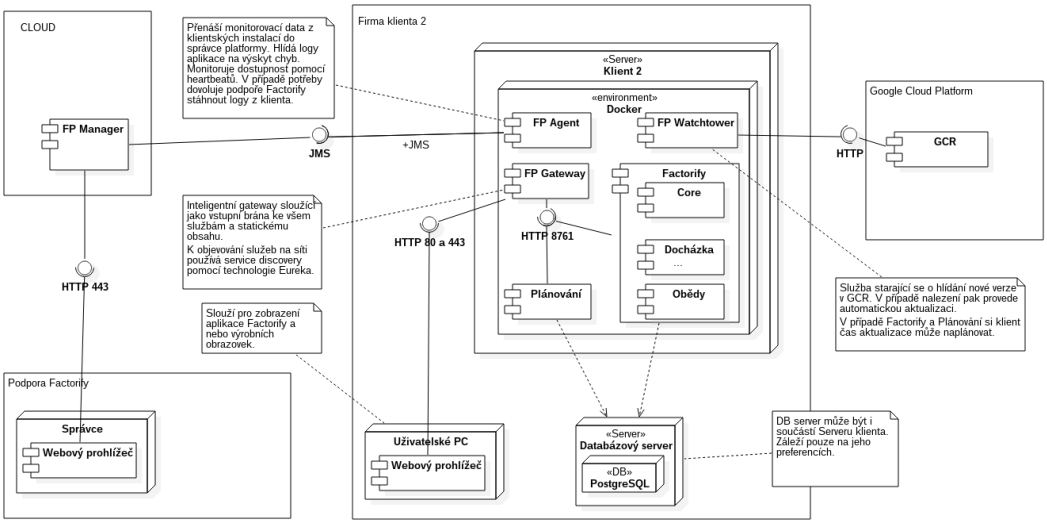
Ve stručnosti popíšeme způsob nasazení *Factorify Platform* pro provoz se systémem *Factorify*. Využití platformy může být jiné, v závislosti na potřebách konkrétní společnosti. *Factorify* zde ale může sloužit jako fungující a referenční model.

Na obrázku 4.1 je zobrazena architektura pro klienta, který je nasazený v cloudu. Základem je vlastní, v cloudu umístěný, VPS s nainstalovaným Dockerem. Veškerá persistentní data se ukládají na databázový server. Je to z důvodu, aby jednotlivé VPS byly bezstavové a v případě potřeby je bylo možné odstranit nebo přesunout [15]. Databázový server a VPS jsou umístěny v privátní síti s vysokou propustností.



Obrázek 4.1. Architektura FP – klient v cloudu

Na obrázku 4.2 vidíme architekturu pro klienta, který si provozuje systém *Factorify* na vlastním HW. Rozdíl oproti cloudové variantě je v tom, že správa běhového prostředí je v režii klientské firmy. To se také týká databáze, která není poskytována v cloudu.



Obrázek 4.2. Architektura FP – klient on-premise

4.1 Technologie

Jelikož startup *Factorify* má většinu svých produktů založených na programovacím jazyku Java a frameworku SpringBoot, tak z pochopitelných důvodů bude většina modulů *Factorify Platform* založena právě na Javě anebo na Kotlinu.

4.1.1 PostgreSQL

Vyspělá a vývojářsky oblíbená SQL databáze, která je uvolněna pod open-source licencí. Technologie je velmi dobře udržovaná, má širokou vývojářskou komunitu složenou z databázových expertů z celého světa [16] a mnoho sponzorů jak ze strany komunity, tak i od komerčních firem.

Oproti zdarma dostupné variantě MySQL[1] nabízí PostgreSQL pokročilé funkce jako například *window functions*[2] či *CTE*[3]. Nutno říci, že CTE je již součástí MySQL ve verzi 8, ta však v době psaní této práce není určena pro produkční použití [17].

Hlavním důvodem pro výběr PostgreSQL byl fakt, že tuto databázi používáme interně ve *Factorify* na všechny projekty. Výhoda použití stejné technologie je zřejmá. Je možné se inspirovat kódem z již funkčního kódu. V případě údržby dalšími lidmi není nutnost se učit novou technologii.

4.1.2 EBean ORM

Jelikož používáme v projektu relační databázi, tak jsme museli buď psát SQL dotazy přímo, nebo si zvolit některý framework pro objektově-relační mapování. Nechtěli jsme použít z našeho pohledu příliš složitý Hibernate. Zvolili jsme tedy technologii, která nepracuje s relací a její architektura nepotřebuje komponentu pro správu entit (Entity manager). Ebean pracuje pouze se správou databázových transakcí.

[1] https://www.mysql.com/

[2] Aktuální řádek má přístup k jiným řádkům během vykonávání dotazu. Například můžeme využít k započtení hodnot předchozích řádků k aktuálnímu. Jinými slovy realizujeme kumulativní součet.

[3] Umožňuje pomocí WITH specifikovat pohledy, které dále použijeme v příkazu SELECT. Pokročilé enginy nabízejí i použití CTE pro rekurzivní dotazy.

```
1   package me.factorify.platform.manager.module.dm
2
3   import ...
7
8   @Entity
9   @Table(name = "modules")
10  class Module : Model {
11      @Id()
12      override var id: Long? = null
13
14      var artifactString: String = ""
15      var name: String = ""
16      var buildTaskId: String = ""
17      var repositoryUrl: String = ""
18      var description: String? = null
19
20      @ManyToOne()
21      @JoinColumn(name = "docker_id", referencedColumnName = "id")
22      lateinit var docker: Docker
23
24      @OneToMany(mappedBy = "module")
25      var versions: MutableList<ModuleVersion> = ArrayList()
26  }
```

Obrázek 4.3. Anotace entity v Ebeans ORM

EBean je poměrně vyspělá technologie, která přináší mnoho pokročilých funkcí a optimalizací na automaticky vytvářených SQL dotazech. Pro přehlednost uvádíme jen výčet [18]:

- Podporuje JDBC zpracování v dávce.
- Podporuje transakce.
- Snaží se chytře optimalizovat počet dotazů. Kde je to možné, použije se JOIN pro dotažení dalších informací. Kde to možné není, spustí se další dotaz, který vybere řádky jen pro příslušné záznamy.
- Pokud je třeba, umožňuje přímý přístup k JDBC.
- Podporuje bezstavový update entity. Pro modifikaci není nutné nejprve entitu načíst z DB.
- Lze nastavit úroveň izolace transakce.

4.1.3 Kotlin

Kotlin je staticky typovaný jazyk pro JVM. Je velice podobný Javě. Důležitou vlastností je, že v Kotlinu můžeme využívat standardní Java knihovny. Kotlin se snaží zjednodušit a zefektivnit kód potřebný ke splnění problému. Výhody oproti Javě, které jsme ocenili, jsou zejména následující:

- properties známé například v C#,
- systém kontroly null datových typů,
- vlastní kolekce s rozšířeným API,
- nepovinný středník,
- datové třídy – automaticky generované funkce equals(), hashCode(), toString() a copy() a
- možnost určit výchozí hodnotu parametru funkce.

4.1.4 Gradle

Jako nástroj pro build aplikace jsme zvolili Gradle, který používáme ve *Factorify* pro všechny projekty běžící v JVM. Dříve jsme používali Maven, ale kvůli jeho rozsáhlým

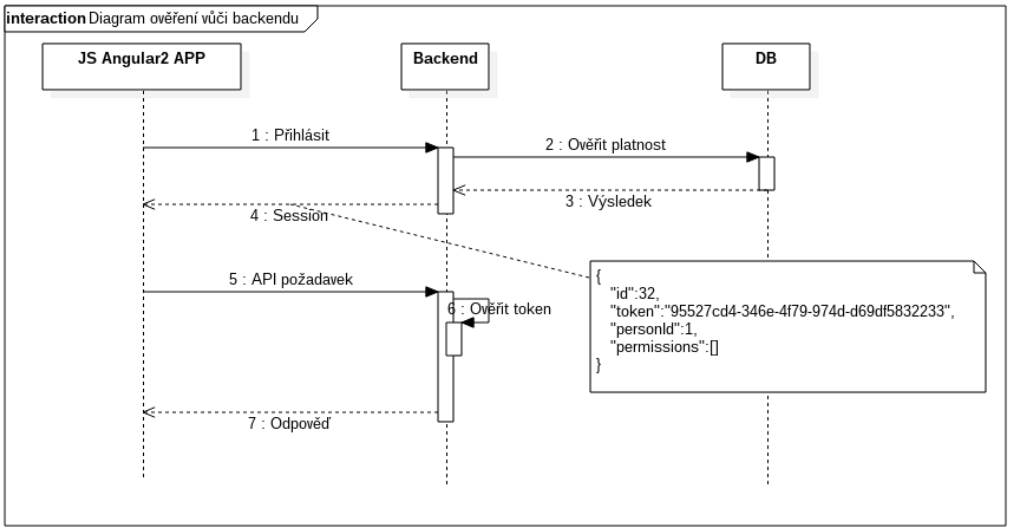
a „upovídaným“ konfiguracím jsme přešli právě na modernější Gradle, který se konfiguruje pomocí jazyka Groovy. Groovy je pro programátora mnohem intuitivnější a dosažení požadovaného výsledku je rychlejší.

4.1.5 Angular2

Pro webovou aplikaci k řízení *FPM*, která je integrovaná na RESTful API, jsme zvolili JS framework Angular[1] ve verzi 2. Jedná se o technologii pro tvorbu „Single page application“, která funguje tak, že aplikace se načte pouze jednou a další přechody jsou již realizovány překreslením dosavadní obrazovky.

Prvotní načtení aplikace je delší z důvodu stažení potřebných knihoven a statického obsahu. To jsme ale ochotni tolerovat. Při práci lze očekávat bohatou interakci s aplikací. Ve výsledku bude čas nutný pro první načtení mnohem kratší v porovnání s tím, kolik bychom strávili času čekáním na překreslení celé stránky.

Autentizace na backend začíná požadavkem obsahujícím uživatelské jméno a heslo na endpoint `/login`. V případě, že jsme vložili platné údaje, tak nám je vrácena session, která obsahuje autentizační klíč, kterým se budeme ověřovat při zasílání dalších požadavků.

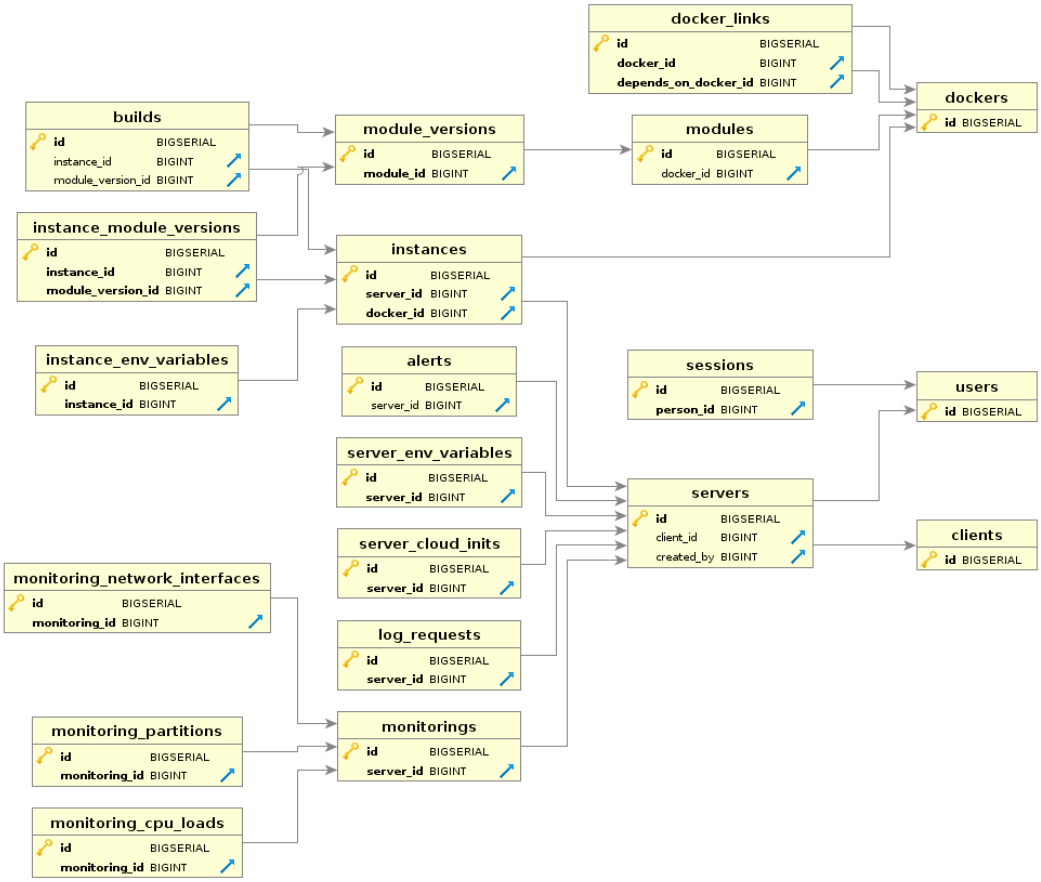


Obrázek 4.4. Sekvenční diagram ověření vůči backendu

4.2 Perzistentní úložiště

Kontejner neposkytuje v základu perzistentní uložení dat. Po vypnutí kontejneru jsou všechna doposud zapsaná data smazána. Pokud potřebujeme data uchovat, pak musíme využít perzistentní úložiště, jakým je například databáze nebo připojený svazek. Připojený svazek (volume) není nic jiného, než obyčejný mount z hostitelského počítače dovnitř kontejneru.

[1] https://angular.io/



Obrázek 4.5. ER diagram *FPM*

4.2.1 Databázové schema

K ukládání strukturovaných dat používáme PostgreSQL. Na obrázku 4.5 je zobrazen ER diagram celé aplikace *FPM*. Jedná se o pohled bez sloupečků. Verzi s nimi naleznete v příloze A.

4.2.2 Data

Aby uložená data měla nějakou strukturu, vymezili jsme standardní adresáře (viz obrázek 4.6) pro ukládání logů, vlastních dat jednotlivých služeb či umístění pro statické soubory, které jsou k dispozici přes webový server. My je používáme pro vlastní JS aplikace.

- `exports` – Tento adresář slouží k importu a exportu dat. Typicky například pro integraci s účetními systémy jako je Pohoda[1]. Do adresářů je poskytnut přístup přes sFTP. Klienti poté sami či pomocí skriptů nahrávají (import) a stahují (export) konkrétní soubory. V období standardu API je tento přístup zastaralý ale na druhou stranu funkční a v mnoha systémech se stále jedná o jediný způsob pro import/export dat.
- `log` – Důležitý adresář, kam by měly všechny služby ukládat své logy. Umístění je zde velmi důležité. *FP Agent* (viz podsekce 4.8.2) monitoruje všechny soubory s maskou `*.log` a v případě, že se v nich objeví záznam s úrovní `ERROR`, tak odesílá zprávu *FPM*.

[1] https://www.stormware.cz/pohoda/

```
1  ├── exports
2  │   └── pohoda
3  │       ├── import
4  │       └── import_result
5  ├── log
6  │   ├── certbot.log
7  │   ├── factorify.log
8  │   ├── fp-agent.log
9  │   ├── gateway.log
10 │   ├── logrotate.log
11 │   └── planning.log
12 ├── service
13 │   └── factorify
14 │       ├── documentation
15 │       ├── pohoda
16 │       ├── storage
17 │       └── unsecure_storage
18 ├── ssl
19 │   └── keystore.p12
20 └── static
21     └── configurator
22         ├── configurator.azub.eu
23         ├── konfigurator.azub.cz
24         └── .rewrite
```

Obrázek 4.6. Struktura perzistentních dat

Další využití spočívá v možnosti stáhnou si logy právě z adresáře `log/` do *FPM*. Všechny logy vyhovující masce jsou umístěny do `ZIP` archivu, přeneseny přes JMS do *FPM* a zde uloženy na disk.

Služba *gateway* dále kromě routingu zajišťuje rotaci logů umístěných v této složce. Rotace se provádí standardním Linuxovým nástrojem logrotate[1]. Více v podsekci 4.8.1.

- `service` – Adresář pro ukládání perzistentních dat jednotlivých služeb. Každá služba si zde v případě potřeby vytvoří adresář se svým jménem a do něj ukládá data. Na obrázku 4.6 můžeme vidět ukázku služby *factorify*.
- `ssl` – Adresář určený pro SSL certifikáty.
- `static` – Adresář pro statický obsah. Gateway (viz podsekce 4.8.1) před směrováním požadavku na základě registrovaných služeb zkouší, zda URI nepředstavuje soubor v tomto adresáři. Pokud ano, vrátí ho. V podsložce je možné vytvořit speciální soubor `.rewrite`, který umožňuje nadefinovat routing na základě domény. Využijeme to, když klient žádá vlastní doménu pro konfigurátor. Detail konfigurace je v uvedené podsekci.

4.3 Komunikace mezi službami

Jednotlivé služby je možné adresovat na vnitřní síti díky DNS jejich jmény. Poněkud komplikovanější situace nastává, pokud potřebujeme poskytnout službu mimo síť. Běžně to nastane, když máme frontendovou JS aplikaci, která komunikuje s několika službami. Pokud by byla každá služba přístupná pod jinou doménou, tak nastane problém se *Same-origin policy*.

[1] http://www.linuxcommand.org/man_pages/logrotate8.html

My se tomuto vyhneme typickým přístupem. Přidáme do systému bránu (gateway), která nám bude rozdělovat požadavky na konkrétní služby.

Standardně je komunikace zajištěna pomocí RESTful API, které jako formát zpráv využívá JSON. V případě, že je komunikace provozována pouze mezi Java aplikacemi, využíváme i JMS. Konkrétně pak zasílání zpráv pomocí technologie ActiveMQ [1].

4.4 Orchestrace

V sekci 3.2 jsme popsali jednotlivé možnosti orchestrace služeb včetně jejich výhod a nevýhod. Jako řešení pro *Factorify Platform* jsme nakonec zvolili Docker. Důvodů pro toto rozhodnutí bylo několik:

- Docker je produkční standard, který je nabízen například v AWS, Azure či Goole Cloud Platform.
- Většina populárního open-source software je k dispozici také v Docker kontejneru.
- Umožňuje snadnou správu instancí služeb.
- Poskytuje izolaci služby uvnitř kontejneru s možností omezit HW zdroje, které smí využít.
- Prostředí a konfigurace je určena v obrazu. Klient se nemusí o nic starat. V případě potřeby nastavení je to možné provést předáním systémových proměnných dovnitř kontejneru anebo připojením konfiguračního souboru.
- Distribuce pomocí obrazů, které je možné verzovat. V případě problému je možné se snadno vrátit k předchozí verzi.
- Lze provozovat nativně i na systému Windows. Jedná se o novinku. Podpora ze strany Microsoftu je zárukou i do budoucna.

Docker sám o sobě orchestraci neumí. K tomu existuje oficiální nástroj Doker Compose, který pomocí konfiguračního souboru umožňuje snadno popsat služby, které je potřebné spustit. Dovolí nám specifikovat síťová rozhraní, připojené perzistentní jednotky, předané systémové proměnné či limitaci HW zdrojů. Pro seznam všech možných nastavení odkážeme čtenáře do oficiální dokumentace [19].

Je potřeba zmínit, že jsme schopni vyjádřit závislosti jednotlivých služeb na sobě a Docker Compose pak při startu služby spustí nejprve všechny služby, které jsou definovány jako závislosti. Co nedokáže zajistit je, že v době spuštění služby budou všechny závislosti připraveny plně k provozu. S tímto je třeba počítat v samotné aplikaci a přizpůsobit chování tak, že při nezdaru o připojení nedojde k jejímu pádu.

Všechny systémy platformy s touto skutečností počítají a v případě nedostupnosti některé služby se snaží svůj požadavek po definovaném intervalu opakovat. Skutečnost, že je služba nedostupná, je zaznamenána zpravidla do logu aplikace na úrovni *WARN*, popřípadě *ERROR*.

4.5 Modularizace

Jedná se o jeden z požadavků, který se od platformy očekává. Existují aplikace/služby, které mohou být všem klientům nabídnuty ve stejné variantě. To je pohodlné především

[1] http://activemq.apache.org/

pro vývojáře. Není nutné do systému zavádět tolik abstrakce a systém tím z pravidla bývá čitelnější, přehlednější, snadněji se udržuje a především mnohem snáze se testuje. Poté jsou aplikace, které je třeba modularizovat. To jsou zejména složité systémy, které nabízejí mnoho funkcionality, ale klient chce využít jen ty části, které dávají smysl pro jeho oblast podnikání. Nezanedbatelná je také stránka finanční. Zpravidla se nabídne základní verze za menší částku a zákazník si dále dokupuje jednotlivé moduly.

Nesmíme zapomenout ani na zakázkový vývoj. Specifický kód musíme někam umístit a mít jednoduchý mechanismus, jak ho zahrnout do distribuce pro konkrétního klienta. Vzhledem k tomu, že budeme podporovat moduly, použijeme je pro specifický kód.

Aktuálně máme ve *Factorify* dvě aplikace, které jsou modulární. Jedná se o samotný systém *Factorify* a dále přidruženou službu *plánování*. Obě jsou napsané v Javě a používají framework Spring Boot. Abychom mohli aplikaci modularizovat, je třeba si stanovit jednotku modularizace, tedy co modul představuje. My jsme se rozhodli, že modul bude Maven artefakt.

Jednotlivé moduly pak budou umístěny v privátním Maven repositáři založeném na programu Artifactorfy, detailněji popsaném v podsekci 4.9.2.

Výsledná aplikace s požadovanými moduly bude sestavována na našem build serveru založeném na Jenkins[1] (viz 4.9.1). Build úloha je parametrizována dvěma parametry:

- `FP_FSID` a
- `FP_SERVER_DEPENDENCIES`.

První unikátně identifikuje server v *FPM*. Druhý parametr poté obsahuje jednotlivé moduly oddělené pomocí znaku „;“. V případě *Factorify* by tento řetězec mohl vypadat například následovně:

```
me.factorify:lunch:3;me.factorify:factorify:stock:1
```

Vidíme, že k základu *Factorify* budou přidány dva moduly. První rozšíří systém o funkcionalitu obědů, druhý pak o funkcionalitu skladu. Pokud si řetězec jednoho modulu rozdělíme oddělovačem „:“, tak nám vzniknou následující tři Maven části:

- `me.factorify` představující skupinu,
- `lunch` určující název artefaktu a
- `3` definující verzi.

4.6 Instance VPS v cloudu

Jak bylo dříve uvedeno, platforma podporuje dva typy serverů. Prvním je on-premise, zpravidla umístěný přímo u klienta ve firmě a spravovaný vlastním IT technikem. Druhým je pak virtuální server umístěný v cloudu.

V této sekci se zaměříme na druhý typ. Výhodou virtualizace je zde zejména možnost snadno zvyšovat a snižovat výkonu dle aktuálních potřeb zákazníka. Server umístěný v cloudu je zajímavý zejména pro klienty, kteří:

- si chtějí systém vyzkoušet,

[1] https://jenkins.io/

- nemají finance na provoz vlastního HW,
- nechtějí se starat o HW či
- nemají zájem řešit fyzické zabezpečení.

4.6.1 Vytváření instance

Založení nového serveru musí být zcela automatizované. Poskytovatel služby tedy musí nabízet API, které to umožní. Vytváření bude řídit *FPM*. Platformu jsme navrhovali tak, aby byla co nejvíce obecná a tak jsme zabudovali podporu pro snadné přidání dalších poskytovatelů VPS. První implementací bude OVH (viz podsekce 3.3.5). Tuto společnost jsme dříve vybrali v rešerši.

Před samotným požadavkem na vytvoření instance musí být daný server nakonfigurovaný v *FPM*. Zejména musí být nastavené služby (Dockery), které mají být na serveru spuštěny.

Některé parametry budou do konfigurace serveru přidány automaticky během procesu vytváření. K tomu slouží uživatelsky konfigurovatelné bloky kódu (tzv. hooky), které se spustí před (pre hook) a po samotném dotazu na vytvoření serveru (post hook).

Výstupem hooků je, zda doběhly úspěšně. Pokud některý hook skončí chybou, tak se další v řadě nespouštějí a celý proces zakládání serveru skončí chybou. Ta je nahlášena obsluze[1], která musí dále rozhodnout, jak daný problém vyřešit.

Popsali jsme si již architekturu vytváření a nyní se podíváme na konkrétní řešení pro *Factorify*. V systému jsou implementovány tyto hooky:

1. Pre hooky
   - Vytvoří se databáze a vygenerují se autentizační údaje, které se zapíší do konfigurace serveru v *FPM*.
   - Provede se build všech modulárních instancí, které budou na serveru spuštěny.
2. Zašle se požadavek na vytvoření serveru.
3. Post hooky
   - Pokud se jedná o doménu, které spravujeme DNS, tak provedeme konfiguraci.
   - Zašle se notifikace klientovi, že byl nainstalován jeho server.

4.7 FPM

Java aplikace založená na frameworku SringBoot, která je centrální částí *Factorify Platform*. Pro jednotlivé agenty představuje server, kam zasílají zprávy a nahrávají data. Správcům platformy pak poskytuje nástroj pro konfiguraci serverů, jejich monitoring a v případě problému alerting.

Výpadek *FPM* neovlivní chod stávajících serverů. V čase nedostupnosti však nebude fungovat monitoring a samozřejmě musíme počítat s nefunkčností jakékoliv služby, kterou poskytuje platforma. Zprávy z jednotlivých agentů, které se nepodařilo odeslat během výpadku, tak se neztratí[2], ale budou odeslány jakmile se obnoví spojení.

[1] V podobě alertu.
[2] Pokud nedojde k restartu agenta. Fronta zpráv se uchovává pouze v operační paměti.

Aby po delším výpadku nedošlo k přehlcení *FPM* zprávami z agentů, tak je počet odeslaných zpráv z fronty omezen na 50. Ne všechny zprávy mají také nastaveno, aby se při selhání doručení zařadily do fronty pro opakované odeslání. Z důvodu zamezení uvíznutí některých zpráv, které není možné opakovaně doručit, je stanoven maximální počet opakování na tři pokusy.

4.7.1 Identifikace serveru

Pro jednoznačné označení serveru používáme identifikátor `FSID` – Factorify server ID. Databázové id nepoužíváme záměrně. Jsou jisté případy, kdy potřebujeme označit veřejně dostupné zdroje pro určitého klienta, ale nechceme, aby je bylo možné snadno odhalit pomocí hrubé síly.

Jedním z případů jsou modulární Docker obrazy. K názvu obrazu se připojuje právě `FSID`. Server díky znalosti svého identifikátoru může stáhnout daný obraz. Veřejně ale není nabízen ostatním uživatelům systému *Factorify Platform*.

Další využití je identifikace serveru při komunikaci přes JMS. Využívá se k tomu dvojice hodnot `FSID` a `secret`.

4.7.2 Build

Build fronta je udržována v databázi. Nový požadavek je do fronty vložen ve stavu *SCHEDULED*. Volitelně je možné i specifikovat čas, od kterého je požadavek aktivní.

Zpracování build požadavků zajišťuje v pravidelných intervalech 20 s spouštěné vlákno, které z databáze vyčte požadavky, které by měly být zpracovány a spustí pro ně build úlohu na daném build serveru, což je konkrétně v našem případě Jenkins.

Protože build proces trvá i několik minut, je potřeba pravidelně zjišťovat, zda již nedošlo k dokončení. To obstarává další vlákno, které v intervalu 30 s zjišťuje stav aktuálně probíhajících build procesů. Pokud dojde ke změně stavu, je to zapsáno do databáze.

4.7.3 Monitoring

Monitorovací data jsou získávána ze zpráv zasílaných agentem v přibližném intervalu 30 s. Monitoruje se téměř vše:

- cpu (každé jádro),
- využití pamětí,
- odezva systému *Factorify*,
- obsazení disku a
- přenosy síťových rozhraní.

Vše je ukládáno do relační databáze. Protože se jedná o hodně dat, tak záznamy starší týdne jsou v pravidelných intervalech odstraňovány. Hlavním smyslem monitoringu je mít přehled o tom, co se děje na serveru. V budoucnosti možná dojde k implementaci specifických alertů.

4.7.4 Alerting

Alerty jsou ukládány do databáze. Při vytváření alertu je možné určit:

- server,

- typ,
- textový popis,
- závažnost – ERROR, WARN, INFO a
- čas vzniku.

Alerty jsou k dispozici jako jeden z mála endpointů ve stránkované podobě. Jde o jednoduchý výpis řazený od nejnovějších po nejstarší. Dále je alerty možné nalézt na endpointu pro konkrétní server. Zde jde pouze o výpis posledních 20 položek.

4.7.5 REST API

Tato podsekce popisuje aplikační rozhraní REST vystavené vnějšímu světu. V přehledu není zahrnutý popis požadavků a odpovědí, protože bychom tím zaplnili několik desítek stran. Zájemce se může podívat přímo do kódu na jednotlivé transportní objekty, které jsou vraceny z příslušných RESTových kontrolérů.

Jednotlivé endpointy jsou v *FPM* definovány metodami s anotací `@RequestMapping`, které jsou umístěné ve třídách anotovaných `@RestController`. Výchozí nastavení je takové, že pro přístup na endpoint musí být uživatel ověřen pomocí hodnoty klíče `securityToken` v `Cookie`. V případě, že není nutná autentizace, je metoda anotována `@NotAuthenticated`.

Z důvodu většího rozsahu API jsme umístili kompletní seznam endpointů RESTful API do přílohy B.

4.8 Subsystémy platformy na straně serveru

Na každém serveru, který bude napojen na *Factorify Platform* bude Docker. Ten se na prostředí automaticky nainstaluje s dodaným instalačním skriptem. K plnému rozsahu funkčnosti platformy je třeba zajistit, aby byly v Dockeru spuštěny služby *FP Gateway*, *FP Agent* a *FP Watchtower*.

4.8.1 FP Gateway

Brána bude sloužit pro obsluhu dvou typů příchozích požadavků. Prvním budou požadavky na statický obsah. Ten bude čten z adresáře `/fp-data/static`. Druhým typem budou požadavky zpravidla na RESTful API jednotlivých služeb, které budou spuštěny v Dockeru. Tyto služby identifikujeme z URL pomocí prefixu. Například `http://my.domain/api/`, `http://my.domain/api4/` atd.

Soubor .rewrite slouží k volitelnému směrování domény do nastavené složky. Soubor `.rewrite` je hledán v podadresářích umístěných v `/fp-data/static`. Dále uvádíme ukázku zápisu:

```
konfigurator.klient.cz=cs
```

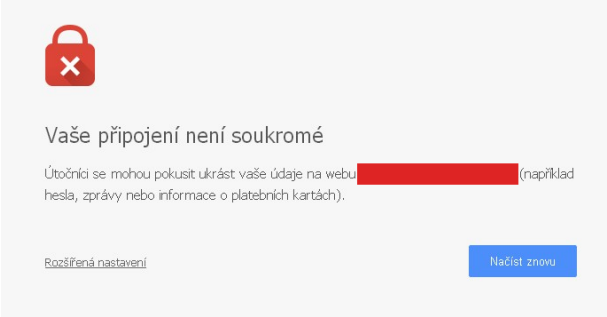
Tato konfigurace zajistí směrování požadavků z domény `konfigurator.klient.cz` do složky `cs`, která se nachází ve stejné složce jako soubor `.rewrite`.

Rotace logů je prováděná již zmíněným nástrojem `logrotate` a to pomocí `CRONU` v pravidelných hodinových intervalech. Samotná rotace je nastavená po dnech nebo v případě, že soubor překročí velikostí `50 MB`. V konfiguraci použitá volba

`copytruncate` zajistí, že rotovaný soubor je nejprve zkopírován a poté nastaven jako prázdný. Důležité je to z důvodu, aby se nám nezměnily popisovače souboru logu a my nemuseli restartovat službu, aby logování fungovalo nadále.

Generování SSL certifikátu je realizováno přes neziskovou certifikační autoritu *Let's Encrypt*[1]. Pokud chceme náš web zabezpečit a používat SSL, tak potřebujeme certifikát. Ten může být podepsaný nebo nepodepsaný.

Výhoda nepodepsaného certifikátu spočívá v tom, že si ho můžeme vystavit sami, nevýhoda pak v tom, že při vstupu na stránku chráněnou takovým certifikátem nás bude varovat hláška podobná té na obrázku 4.7. Toto chování si můžeme dovolit na interní webové prezentaci, nikoliv však v komerční aplikaci.



Obrázek 4.7. Nezabezpečené připojení – hláška v Chrome

Je zřejmé, že budeme muset použít podepsanou variantu certifikátu. Aby se nám nezobrazovala varovná hláška v prohlížeči, je nutné, aby autorita, která podepíše náš certifikát, byla zahrnuta v kořenových certifikátech. Ty jsou součástí OS, prohlížeče či i některých SDK, jako je například Java. Většina certifikačních autorit požaduje za vydání certifikátu vysoké částky. Existuje ale alternativa v podobě certifikační autority *Let's Encrypt*.

Let's Encrypt nabízí plně automatizované vydávání certifikátů. Pro zjednodušení celé operace byla uvolněna aplikace *certbot*[2] naprogramovaná v Pythonu. Vlastně se jedná o certifikačního agenta. Celý proces ověření probíhá zjednodušeně tak, že agent zadá požadavek k vytvoření certifikátu na certifikační autoritu (dále CA) Let's Encrypt. CA nazpátek zašle ověřovací kód (tzv. challenge). Tento kód se uloží do souboru a musí se zpřístupnit na určené URI. Poté agent znovu notifikuje CA, že může danou doménu ověřit. Pokud CA úspěšně ověří doménu, tak je nazpět zaslán certifikát.

Certifikáty jsou vystavovány vždy na 90 dní. Je tedy dobré nastavit CRON[3] pravidelně spouštějící certbota, který zkontroluje platnost certifikátu a pokud se bude blížit konec platnosti, tak automaticky zajistí prodloužení.

4.8.2 FP Agent

FP Agent je důležitou součástí platformy. Jedná se o Spring Boot aplikaci, která zajišťuje komunikace s centrálním bodem *Factorify Platform*, tedy *FPM*. Komunikace probíhá v podobě vyměňování zpráv pomocí technologie JMS.

[1] https://letsencrypt.org/
[2] https://certbot.eff.org/
[3] Proces spouštějící úlohy v pravidelně nastavený čas.

Důvod, proč jsme nezvolili komunikaci pomocí RESTful API je, že bychom museli využít přístupu „PULL“. To znamená, že agent by si musel příkazy z *FPM* stahovat periodicky. Aby reakční doba na požadavek byla přijatelná, musel by interval být nastaven okolo 5 s. To je nezanedbatelná zátěž na server. Metoda „PUSH“ by samozřejmě fungovala v případě, že klienti by byli dosažitelní na určité doméně/IP adrese. To jsme schopni garantovat pro prostředí cloudu. V podmínkách on-premise nasazení však nikoliv.

Hlavním úkolem *FP Agent* je zasílat heartbeaty do *FPM*. Tyto zprávy navíc obsahují informace, které se používající k monitoringu vytížení. Jde především o monitoring využití paměti RAM, vytížení procesoru, PING monitorované aplikace [1].

Další úloha, kterou agent zastává, je monitoring logu. Jak již bylo zmíněno, všechny logy služeb platformy by měly být umístěny v adresáři `/fp-data/log`. Agent monitoruje všechny soubory končící příponou `.log`. Pokud je řádek detekován jako chyba, tak je odeslána zpráva do *FPM*, která je převedena na alert. Ten je vidět v platformě a také v pravidelných intervalech zasílán na email. Pro snadný přístup k tomu, co se děje uvnitř služeb, je implementována funkce pro stažení logů. Požadavek obsahuje masku ve formátu regulárního výrazu, kterým jsou vyfiltrovány požadované soubory. Vyhovující soubory jsou poté zkomprimovány do formátu ZIP a zaslány do *FPM*.

Návrh bere v potaz možný výpadek připojení k *FPM*. V takovém případě jsou zprávy uloženy do fronty a odeslány v momentě, kdy je spojení obnoveno.

4.8.3 FP Watchtower

Tato služba má jednoduchou, ale zároveň kritickou funkci. Zajišťuje hlídání nové verze docker obrazu v repositáři. Jakmile je nová verze nalezena, tak dojde k jejímu stažení. Následně je provedeno vypnutí běžícího kontejneru se starou verzí a spuštění kontejneru založeném na novějším obrazu. Nový kontejner je spuštěn se všemi parametry, se kterými běžel ten předchozí.

4.9 Podpůrné služby

V této sekci shrneme služby, které jsou nezbytné pro fungování *Factorify Platform*. Na většinu z nich se systém integruje pomocí RESTful API.

4.9.1 Jenkins

Open-source build server, který je díky nulovým pořizovacím nákladům velmi oblíbený. Pokud se podíváme na statistiku použití (viz obrázek 4.8), tak s vysokou mírou pravděpodobnosti můžeme říci, že Jenkins je nejpoužívanějším build serverem.

Ve *Factorify* používáme paralelně i TeamCity. To je dle subjektivního názoru více odladěné a uživatelsky přívětivější. Důvodem částečného přechodu na Jenkins je limit počtu build konfigurací ve verzi zdarma. Cenová politika[21] byla bohužel nastavena tak, že se připlácí přibližně 8 tis. Kč za dalších 10 build konfigurací či necelých 55 tis. Kč pro zakoupení nejlevnější neomezené varianty.

[1] Nyní monitorujeme ednpoint factorify – http://factorify:8080/languages. To je možné změnit v souboru `application-docker.properties` vlastností `fpa.healthcheck-url`

Product	Install base (# of companies we found using this product)	Market Share (%)
Microsoft Visual Studio	68,049	28%
Jenkins	26,619	11%
Microsoft TFS	16,682	6%
Apple xCode	12,392	< 5%
JIRA Software	12,234	< 5%
Progress Software	10,717	< 5%
Telerik	10,074	< 5%
Sybase PowerBuilder	7,683	< 5%
IBM Rational Rose	6,490	< 5%
IntelliJ	6,188	< 5%
TeamCity	5,946	< 5%
Adobe Flex	5,173	< 5%

Obrázek 4.8. Statistika využití nástrojů pro vývoj software (zdroj iDatalabs [20])

Komunikace s Jenkins probíhá přes RESTful API. Využíváme k tomu již vytvořeného klienta pro Javu[1]. Autentizace je zajištěna pomocí uživatelského jména a hesla. Obojí je nastaveno pomocí konfiguračního souboru `application.properties` pomocí vlastnosti `fpm.jenkins.user`, resp. `fpm.jenkins.password`.

4.9.2 Artifactory

Snadno nastavitelný repositář pro nepřeberné množství typů artefaktů. My budeme využívat pouze rozšíření pro Maven, které je k použití zdarma. Doplňky, které jsou k dispozici v placeném rozšíření, nyní nepotřebujeme. Přístup pro čtení a zápis je chráněn uživatelským jménem a heslem.

V tomto repositáři budou umístěné jednotlivé moduly jako Maven artefakty.

4.9.3 Ticketovací nástroj Factorify

Factorify má v sobě integrovaný jednoduchý úkolovací systém, který umožňuje vytvářet tickety, přiřazovat je k projektu, komentovat a vykazovat na ně strávený čas. Z důvodu, abychom interní funkcionalitu systému *Factorify* stále zdokonalovali, rozhodli jsme se používat úkolovací modul a plynule přejít z aktuálně používaného Redmine[2].

Smyslem integrace na ticketovací nástroj je automatické generování poznámek k vydané verzi modulu. V momentě, kdy bude klient chtít provést aktualizaci, bude mu možné nabídnout seznam všech změn mezi verzí, kterou má aktuálně nasazenou a nejnovější verzí, která je k dispozici v repositáři.

4.9.4 Google Container Registry

Architektura postavená na Dockeru vyžaduje registry, kde budou uloženy všechny potřebné obrazy. Registry musí být privátní a dostupné pouze těm, kteří mají platné oprávnění. Vzhledem k bezpečnostním a výkonnostním požadavkům jsme zvolili řešení od Google.

Do registrů jsou vytvořeny dva uživatelské účty. Jeden s oprávněním pro zápis, který je určen pro vývojáře a build server. Druhý s oprávněním pouze pro čtení a ten je

[1] https://github.com/jenkinsci/java-client-api
[2] http://www.redmine.org/

určen pouze pro klienty. Tento klíč obdrží klient spolu s instalačním skriptem, aby byl schopen pomocí vygenerovaného `docker-compose.yml` stáhnout potřebné docker obrazy a spustit určené služby.

Kapitola 5
Implementace

V předchozí kapitole jsme si prošli všechny důležité části systému z pohledu návrhu. V této kapitole se zaměříme na implementační detaily důležitých a zajímavých částí systému. Pro názornost nebudou chybět ani ukázky kódu.

5.1 Komunikace v týmu

V úvodu dokumentu jsme zmínili, že na projektu jsme pracovali dva. Celý projekt jsem zastřešoval já. Kolega z Moravy, Petr Skalička, mi pomohl z velkou částí frontendu *FPM*. Některé části práce jsou lépe prezentovatelné čtenáři z pohledu UI. V takovém případě je u obrázku vždy uvedeno, že autorem je Petr Skalička.

Při vývoji jsme se snažili o agilní přístup. Na začátku dne jsme si písemně či hovorem sdělili stav práce a stanovili si další postup. Během dne jsem pak v případě problémů či otázek byl Petrovi Skaličkovi k dispozici.

Práci jsme měli rozdělenou na menší celky, které byly popsány v podobě ticketů v systému *Redmine*. Petr vždy po dokončení celku vytvořil pull request[1], který jsem pročetl a často doplnil o komentáře týkající se problémových míst nebo námětů na vylepšení. Jakmile bylo vše v pořádku, tak jsem připojil (provedl merge) změny do projektu (master větve).

5.2 Verzování kódu

K tomuto účelu využíváme systém Git[2]. Jedná se o roky prověřený standard, který našim potřebám synchronizace kódu mezi dvěma vývojáři zcela dostačuje. Do zprávy commitu se vždy snažíme přidávat číslo ticketu v Redmine. Již několikrát se nám tento přístup osvědčil, když jsme potřebovali zpětně zjistit význam některých částí kódu. Aktuálně je hlavní repositář umístěn v systému Bitbucket.

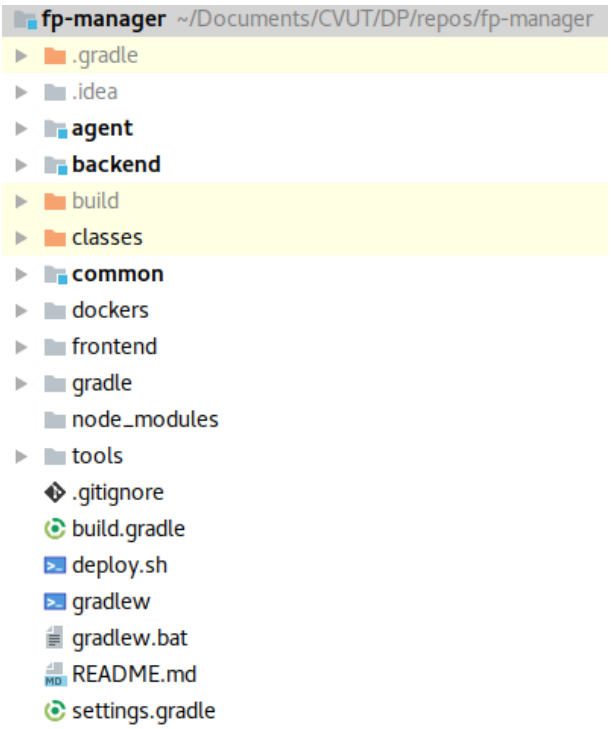
5.3 FPM

Tato sekce nás seznámí s implementačními detaily správce platformy. Kód pro frontend a backend je umístěn ve společném repositáři. Členění je pak docíleno pomocí adresářové struktury, jak můžeme vidět na obrázku 5.1. Mezi důležité složky patří:

- `agent` – obahuje kódy k *FP Agent*.
- `backend` – obsahuje kódy k *FPM*.

[1] Žádost o přidání jeho úprav do projektu.
[2] https://git-scm.com/

- `common` – nachází se v ní sdílený kód pro *FPM* a *FP Agent*. Zejména se jedná o transportní objekty, které jsou používané pro komunikaci pomocí JMS mezi *FP Agent* a *FPM*. Dále jsou zde umístěny společné Gradle závislosti v souboru `build.gradle`, který je importován oběma předchozími aplikacemi.
- `frontend` – obsahuje veškeré zdrojové kódy pro frontend.



Obrázek 5.1. Struktura projektu *FPM*

5.3.1 Jak spustit

Zde uvádíme postup pro zprovoznění aplikace pro vývoj. Před pokračováním se prosím ujistěte, že máte následující nezbytné nástroje, aplikace či knihovny:

- Nginx 1.10.3 či vyšší,
- Gradle 3.2.1 či vyšší,
- PostgreSQL 9.5 či vyšší,
- NodeJS 6.9.1 či vyšší,
- npm 3.10.8 ci vyšší a
- JDK 8 či vyšší.

Díky použití nástroje Gradle je build a spuštění Java aplikací otázkou jednoho příkazu:

```
gradle bootRun
```

Ten spustíme vždy ve složce projektu. Gradle automaticky stáhne potřebné závislosti z repositářů, provede zpracování entit pro Ebeans ORM a nakonec spustí samotnou aplikaci. Při prvním spuštění je potřebné připojení k internetu kvůli stažení závislostí. Opakovaný start již přístup do repositářů nevyžaduje, můžeme být klidně offline.

V případě *FPM* je před prvním spuštěním nutné vytvořit databázi a schéma `public`. O vytvoření tabulek se postará inicilizační skript `/resources/db/01\_dm.sql`, který musíme spustit manuálně.

Jestliže chceme spustit i frontend, musíme nastavit proxy. To z důvodu, abychom se nedostali do problému se `Same-origin policy`. My používáme jako proxy Nginx. Dále uvádíme ukázku konfigurace právě pro tento webový server.

```
server {
listen               4202;
  server_name          localhost;

  error_log    /var/log/nginx/fpm.error.log info;
  rewrite_log on;

  client_max_body_size 100M;

  location / {
    proxy_pass   http://localhost:4201/;
  }

  location /api/ {
    proxy_pass   http://localhost:8080/;
  }

  location /api2/ {
    proxy_pass http://private-aadb2-fpm1.apiary-mock.com/;
  }
}
```

Cestu pro `error_log` si každý musí upravit dle svých potřeb. K aplikaci poté přistupujeme na lokálním počítači přes url `http://localhost:4202`. Pokud jsme úspěšně spustili obě aplikace, tak by se měla zobrazit stránka s přihlašovacím formulářem.

Vývoj frontendu probíhá tak, že si spustíme lokální server ze složky `frontend` příkazem:

```
npm start
```

Ten spustí webový server na portu *4201*, kde vystavuje zkompilovaný projekt. Kompilaci zdrojového kódu a zabalení do formátu distribuce realizuje *angular-cli*[1], což je standardní nástroj pro vývoj aplikací v Angular 2. Jakmile dojde k modifikaci zdrojového souboru, tak se změny překompilují a automaticky dojde k opětovnému načtení webové stránky v prohlížeči.

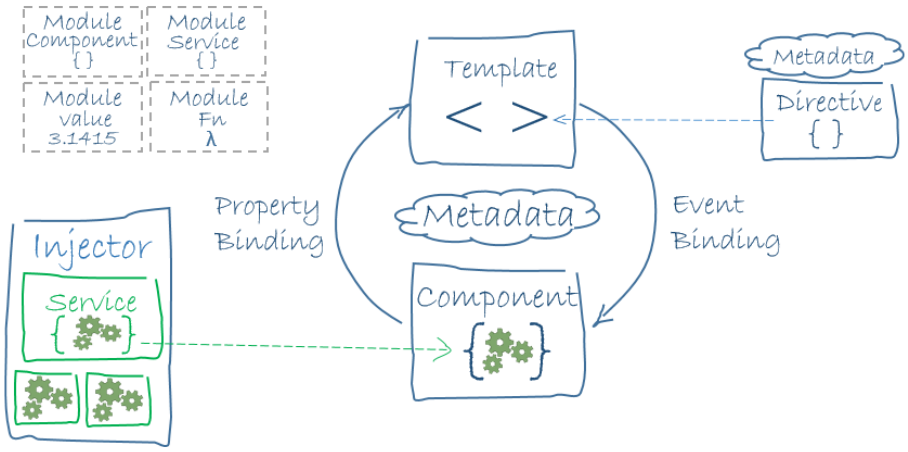
5.3.2 Frontend

I když většinu práce zde odvedl Petr Skalička, tato část systému neodmyslitelně patří k celé *Factorify Platform*. Z toho důvodu jsem se rozhodl ji zde alespoň v krátkosti popsat.

[1] https://cli.angular.io/

Základem je již zmíněný JS framework Angular 2. Veškerý kód je psán v TypeScriptu[1], který je při kompilaci převeden do JS. TypeScript přináší volitelné typování a mnoho konstruktů, které usnadňují práci. Za zmínku stojí například zavedení tříd, rozhraní a modulů.

Angular 2 za nás řeší členění kódu, *dependency injection*[2], což lze vidět na obrázku 5.2. Poskytuje šablonovací systém včetně automatické synchronizace proměnných mezi JS a HTML či směrování požadavků na komponenty. Dále nabízí mnoho vlastních služeb pro snazší práci s RESTful API, formuláři včetně jejich validace, ukládání dat do *localstorage*[3] a další.



Obrázek 5.2. Architektura Angular 2 [22]

Angular 2 zavádí dekorování tříd pomocí metadat. Jde o podobný koncept jako nabízí Java a anotace. Na obrázku 5.3 můžete vidět dekorátor `@Component`, který slouží k přidání kódu nutného pro fungování dané třídy jako komponenty. Dekorátory jsou definovány pomocí rozhraní, takže IDE našeptává možné parametry včetně datového typu.

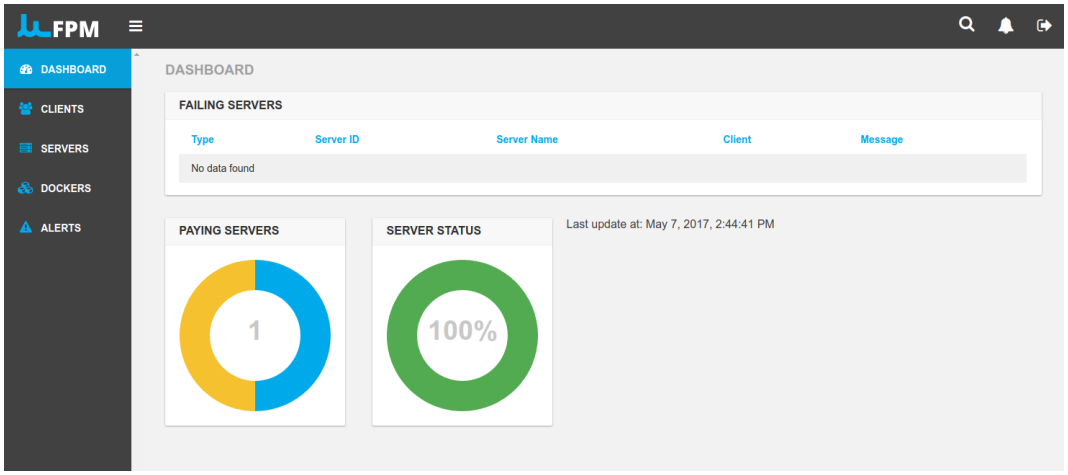
Komponenta spolu se službou (service) tvoří nejzákladnější stavební prvky aplikace. Komponenty jsou logické prvky kódu, které poskytují nějakou funkci a mají grafickou reprezentaci v podobě HTML šablony. Službou je pak třída, která poskytuje sadu funkcí.

```
10  @Component({
11      selector: 'layout',
12      templateUrl: './layout.component.html'
13  })
14  export class LayoutComponent {
15
```

Obrázek 5.3. Ukázka dekorátoru v Angular 2

Dále uvádíme ukázky z frontendové aplikace *FPM*. Znovu upozorňujeme, že autorem většiny kódu je Petr Skalička, což je pro jistotu uvedeno i u každého obrázku. Výchozím

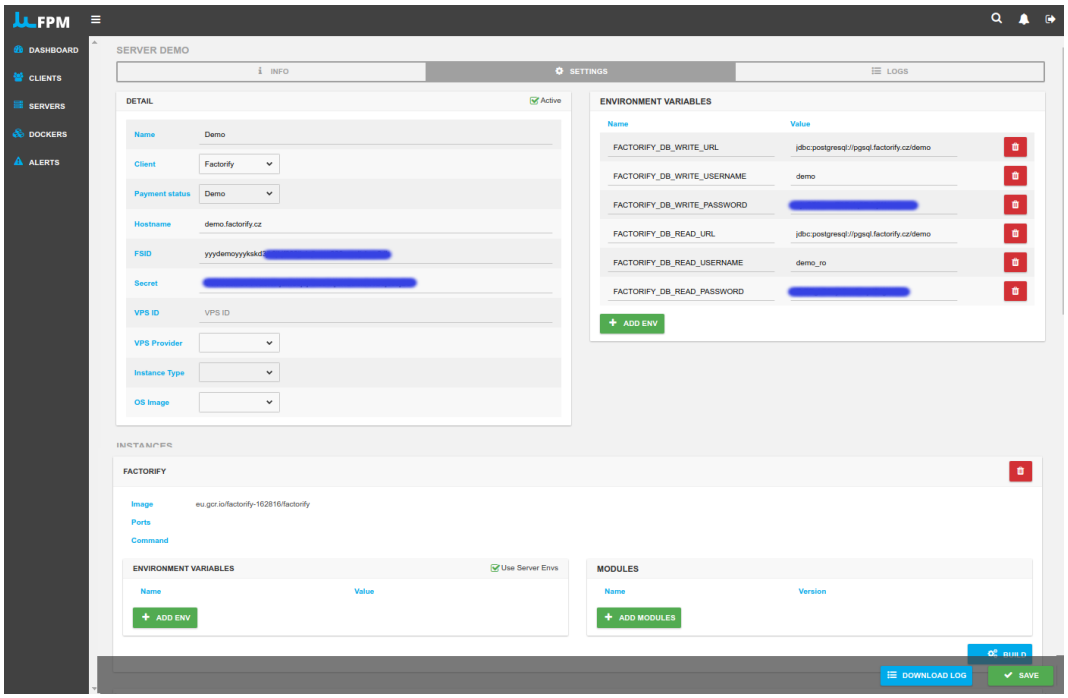
[1] https://www.typescriptlang.org/
[2] Poskytování závislostí do komponenty bez znalosti toho, jak jsou dané komponenty vytvořeny.
[3] Perzistentní úložiště dat prohlížeče vztažené k doméně



Obrázek 5.4. FPM – nástěnka [23]

jazykem aplikace je angličtina. Protože není aktuálně hotová lokalizace, tak jsou textace v angličtině.

Na obrázku 5.4 je grafická vizualizace dat z endpointu `/api/monitoring/stats` metodou `GET`. Ve výpisu chybových serverů jsou stroje, které jsou označeny jako aktivní a za posledních 15 min od nich nebyl obdržen heartbeat nebo kontrolní PING. Ten nebyl schopen navázat spojení s monitorovanou službou.



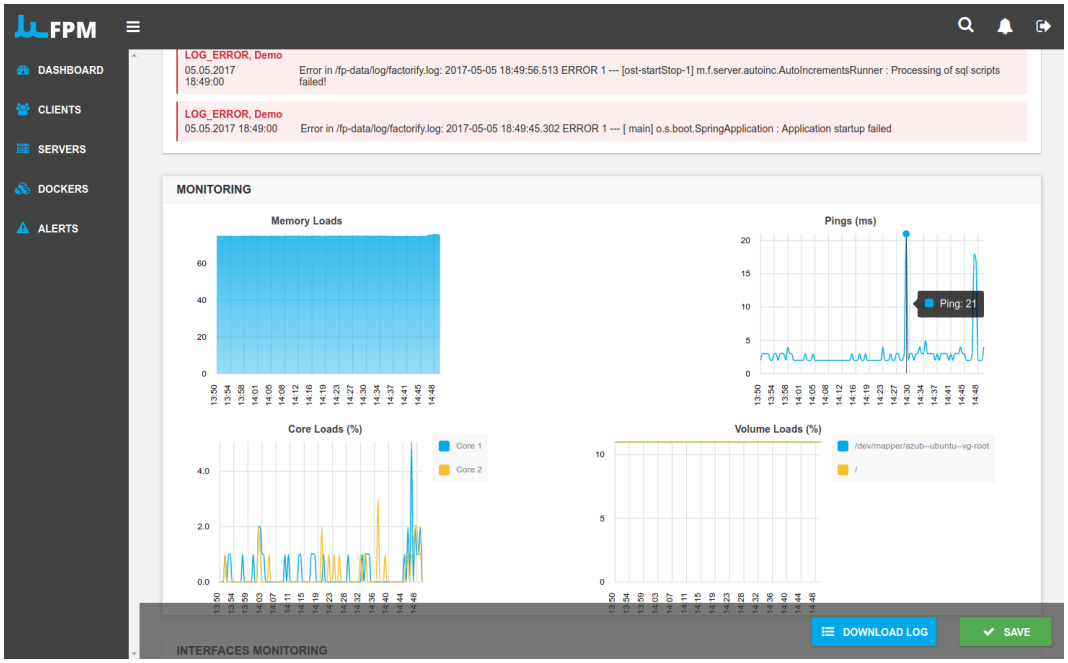
Obrázek 5.5. FPM – detail konfigurace serveru [23]

Obrázek 5.5 zobrazuje detail konfigurace serveru. Data jsou získána z endpointu `/api/server/{id}` metodou `GET`. Uložení detailu je možné provést zasláním dat na stejný endpoint s využitím metody `POST`.

Tento detail je komplexní. Data jsou na backendu uložena do několika databázových tabulek. Nejobecnější data jsou vidět v boxíku „Detail“. Spodní čtyři pole jsou

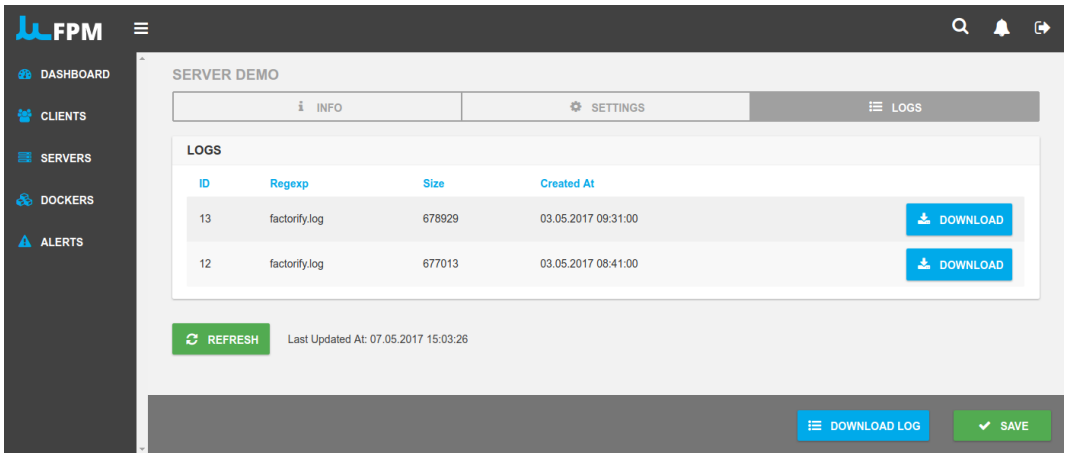
specifická pro použití v prostředí cloudu. Boxík „Environment variables“ umožňuje zadat proměnné prostředí, které budou k dispozici v instancích dockerů, jenž budou mít aktivní volbu „Use server envs“.

Dále je možné nastavit konkrétní instance dockerů, které na daném prostředí budou nasazeny. Můžeme zde určit proměnné prostředí specifické dané instanci. Pokud se jedná o modulární docker obraz, tak můžeme specifikovat i aktivní moduly včetně verze.



Obrázek 5.6. FPM – monitoring [23]

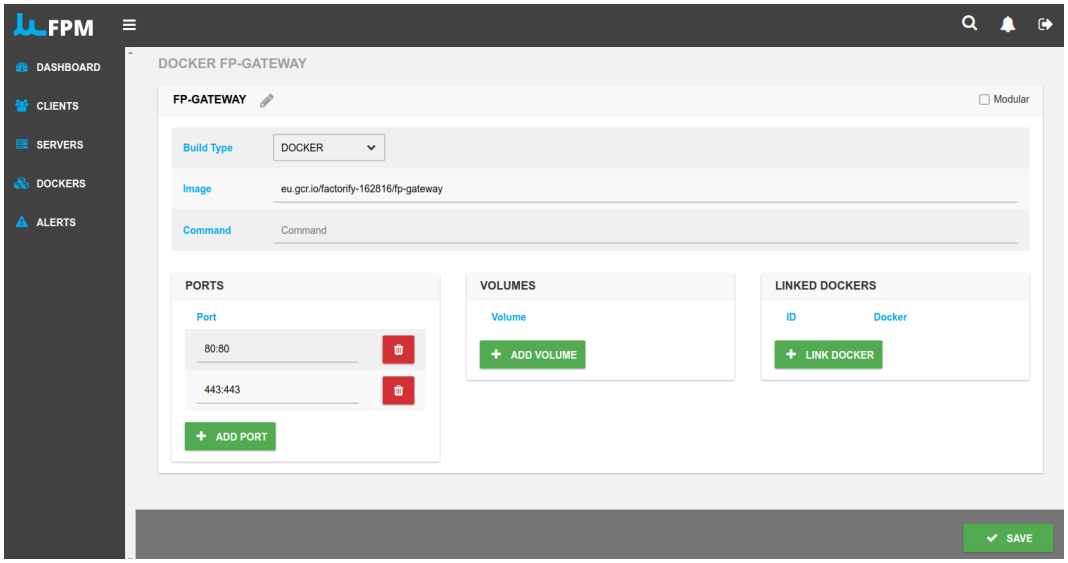
Obrázek 5.6 poskytuje přehled o tom, co se aktuálně odehrává v serveru z hlediska systémových prostředků. Data jsou získávána z endpointů: `/api/server/{id}/monitoring` a `/api/server/{id}/logs`.



Obrázek 5.7. FPM – logy [23]

Obrázek 5.7 ukazuje poslední záložku serveru, kde je vidět seznam logů stažených ze serveru. Log je možné si jednoduše vyžádat. Slouží k tomu tlačítko „Download log“

umístěné v patičce. Po kliknutí na tlačítko se otevře modální okno, kde je možné specifikovat masku logů, které chceme. Požadavek je asynchronní. Musíme dále zjišťovat, zda je již log k dispozici. Je to z důvodu, že soubory logů mohou být objemné a jejich zpracování zabere nezanedbatelný čas pohybující se i v řádu desítek sekund.



Obrázek 5.8. FPM – detail konfigurace Dockeru [23]

Obrázek 5.8 zobrazuje detail Dockeru. Data jsou stahována z `/api/docker/{id}` metodou `GET`. Pro uložení použijeme stejný endpoint v kombinaci s metodou `POST`. Zajímavý je zde přepínač „Modular“, který určuje, zda docker obraz bude obsahovat moduly, které si bude klient moci navolit.

5.3.3 Databáze

Konfigurace připojení je díky využití frameworku Spring Boot otázkou několika řádků v konfiguračním souboru. Následuje ukázka konfigurace ze souboru `application.properties`. Nastavení jsou použita pouze pro lokální vývoj. Konkrétně URL, heslo a uživatelské jméno je na produkčním serveru odlišné.

```
spring.datasource.platform = postgres
spring.datasource.url = jdbc:postgresql://127.0.0.1:5432/fpm
spring.datasource.username = fpm
spring.datasource.password = fpm
spring.datasource.tomcat.default-auto-commit = false
spring.datasource.tomcat.max-active = 2
```

5.3.4 Modularita

Většina částí systémů je vytvářena tak, aby si další člověk, který bude platformu používat, mohl dodat implementaci pro klíčové funkční celky. Abstrakce je realizována formou definovaných rozhraní. V rámci této práce je dodána vždy alespoň jedna implementace. Dále uvádíme seznam těchto rozhraní:

- `DockerBuilder`,
- `ModuleBuilder`,

- `VpsProvider` a
- `ReleaseNoteAdapter`.

5.4 Vytvoření instance v cloudu

Z důvodu, že jde o časově náročnější operaci, jsme zpracování opět rozdělili na uložení požadavku do DB. Následně proběhne ve smyčce obsloužení všech požadavků, které běží v samostatném vlákně. Postup vytváření instance VPS jsme uvedli v kapitole zabývající se návrhem.

Na obrázku 5.9 můžete vidět praktický průběh celého procesu zakládání instance VPS. V konkrétním případě dojde k vytvoření DB, založení VPS v OVH cloudu a následné registraci do DNS. Jedná se o reálný log z vytváření prostředí pro klienta *Factorify*. Některé údaje byly z logu odstraněny za účelem lepší čitelnosti.

```
23:15:46.839 - ServerDeploymentService   : ╔ Start processing new cloud init requests
23:15:46.841 - ServerDeploymentService   : Processing request #13 created at 2017-05-14T23:15:34
23:15:46.856 - ServerDeploymentService   : Running PRE VPS create hooks (1)
23:15:46.856 - ServerDeploymentService   : Running hook: PostgreSQL DB initializer for Factorify & planning
23:15:46.856 - CreateFyDatabase          : Creating connection to jdbc:postgresql://37.59.26.57/fpm with username fpm
23:15:46.862 - CreateFyDatabase          : Connected, running scripts ...
23:15:47.727 - CreateFyDatabase          : 3 queries processed
23:15:47.727 - CreateFyDatabase          : Connection to jdbc:postgresql://37.59.26.57/fpm closed
23:15:47.727 - CreateFyDatabase          : Creating connection to jdbc:postgresql://37.59.26.57:5432/xxxnitaraxxxgt1m
ul3s9nw99i85mcdtnjt9fbfxg4ldygb09o74 with username fpm
23:15:47.735 - CreateFyDatabase          : Connected, running scripts ...
23:15:47.768 - CreateFyDatabase          : 4 queries processed
23:15:47.768 - CreateFyDatabase          : Connection to jdbc:postgresql://37.59.26.57:5432/xxxnitaraxxxgt1mul3s9nw99
i85mcdtnjt9fbfxg4ldygb09o74 closed
23:15:47.768 - CreateFyDatabase          : Creating connection to jdbc:postgresql://37.59.26.57:5432/xxxnitaraxxxgt1m
ul3s9nw99i85mcdtnjt9fbfxg4ldygb09o74 with username xxxnitaraxxxgt1mul3s9nw99i85mcdtnjt9fbfxg4ldygb09o74
23:15:47.774 - CreateFyDatabase          : Connected, running scripts ...
23:15:47.781 - CreateFyDatabase          : 3 queries processed
23:15:47.781 - CreateFyDatabase          : Connection to jdbc:postgresql://37.59.26.57:5432/xxxnitaraxxxgt1mul3s9nw99
i85mcdtnjt9fbfxg4ldygb09o74 closed
23:15:47.864 - ServerDeploymentService   : Hook PostgreSQL DB initializer for Factorify & planning finished
23:15:47.864 - ServerDeploymentService   : Creating VPS using OVH
23:15:50.382 - ServerDeploymentService   : VPS #4d478dad-0c62-4996-b73f-26c5ed845ecc created
23:15:50.408 - ServerDeploymentService   : Updated server configuration saved
23:15:50.408 - ServerDeploymentService   : Running POST VPS create hooks (1)
23:15:50.408 - ServerDeploymentService   : Running hook: Register IP address to DNS if managed by OVH
23:15:50.409 - AddDnsRecordHook          : Registering nitara.factorify.cloud to the OVH DNS
23:15:51.321 - AddDnsRecordHook          : Missing IP addresses. Waiting ...
23:15:55.598 - ServerAgentEndpoint       : Received heartbeat from: yyydemoyyykskd3jk0jkjf903jjoiajiwmc320ncudjskeu3dk
23:15:55.666 - AddDnsRecordHook          : Adding to DNS successful. Refreshing DNS ...
23:15:55.828 - AddDnsRecordHook          : DNS refreshed
23:15:55.840 - ServerDeploymentService   : Hook Register IP address to DNS if managed by OVH finished
23:15:55.857 - ServerDeploymentService   : ╚ Finished processing new cloud init requests
```

Obrázek 5.9. Log zakládání instance VPS v cloudu

5.4.1 Build

Veškeré build úlohy jsou spuštěny na server mimo ten, na kterém běží *FPM*. Jak jsme již zmínili v předchozím textu, stav buildu je zjišťován periodicky. Kód implementace můžeme vidět na obrázku 5.10.

Nejprve se zjistí seznam aktivních build tasků z databáze. Poté jsou všechny ve smyčce zkontrolovány, zda nedošlo ke změně jejich stavu. Pokud ano, tak se nový stav zaznamená do DB. V případě, že build skončil úspěchem – hodnota `BuildStatus.SUCCESS` – tak jsou realizovány akce v závislosti na tom, zda se jednalo o build instance nebo modulu.

- `Instance` – Aktualizuje se datum posledního buildu. To je reprezentováno sloupečkem `build_at` v DB. Dále se vyvolá událost `InstanceBuiltEvent`, na kterou lze reagovat v případě potřeby kdekoli v aplikaci. My to používáme například, když detekujeme okamžik, kdy můžeme začít vytvářet VPS v prostředí cloudu.

- `Modul` – V případě úspěšného buildu je provedeno automatické vygenerování soupisu změn verze (release notes). Poskytovatele release notes si můžeme nastavit pomocí dodání implementace rozhraní `ReleaseNoteAdapter`. Nyní realizujeme třídou `FactorifyReleaseNoteAdapter`, kterou vytváříme v konfigurační třídě `FpmContext`.

```
@Scheduled(fixedDelay = 30000)
@Synchronized
fun checkForBuildStatusChanges() {
    log.info("╔ Started find build status change updates")
    val buildsInProcess = buildRepository.findBuildsInProcess()
    if (buildsInProcess.isEmpty()) {
        log.info("╚ Finished. No running builds ..")
        return
    }
    log.info("Build requests in queue: ${buildsInProcess.size}")

    buildsInProcess.forEach { build ->
        val buildStatus = if (!build.isModuleBuild()) {
            val dockerBuilder = dockerBuilderRegistry.getBuilder(build.type!!)
            dockerBuilder.buildStatus(build.buildId!!)
        } else {
            val buildTaskId = build.moduleVersion!!.module.buildTaskId
            moduleBuilder.fetchBuildStatus(buildTaskId, build.buildId!!)
        }

        if (buildStatus != build.status) {
            log.info("Updating status of build  {} from {} to {}", build.id, build.status, buildStatus)

            when(buildStatus) {
                BuildStatus.SUCCESS -> {
                    if (!build.isModuleBuild()) {
                        val instance = build.instance!!
                        if (buildStatus == BuildStatus.SUCCESS) {
                            val isFirstBuild = instance.buildAt == null
                            instance.buildAt = LocalDateTime.now()
                            instanceRepository.save(instance)

                            log.info("Publishing instance built event ...")
                            applicationEventPublisher.publishEvent(InstanceBuiltEvent(instance, isFirstBuild))
                        }
                    } else {
                        val moduleVersion = build.moduleVersion!!
                        val latestReleasedModuleVersion = moduleVersionRepository.findLatestModuleVersion(moduleVersion.module.id!!)
                        val closedAfter = if (latestReleasedModuleVersion != null) latestReleasedModuleVersion.createdAt!! else
                            LocalDateTime.of(1970, 1, 2, 0, 0)
                        try {
                            moduleVersion.releaseNotes = releaseNoteService.createReleaseNotes(moduleVersion, closedAfter)
                        } catch(e: Exception) {
                            log.error("Creating release notes for module #${moduleVersion.module.id} v${moduleVersion.version} " +
                                    "failed. Probably ticket server is down. Complete release notes manually or " +
                                    "fix the release note service and release new version.", e)
                        }
                        moduleVersionRepository.save(moduleVersion)
                    }
                    build.buildAt = LocalDateTime.now()
                }
                BuildStatus.FAILURE -> log.error("Build #${build.id} FAILED")
                else -> {/* Do nothing*/}
            }

            build.status = buildStatus
            buildRepository.save(build)
        }
    }
    log.info("╚ Finished find build status change updates")
}
```

Obrázek 5.10. Kód zjišťující stav build tasků

5.4.2 JMS

Pro zasílání zpráv používáme ActiveMQ[1]. Díky frameworku Spring Boot je konfigurace opět mnohem snazší. Do souboru `build.gradle` jsme přidali pro projekt *FP Agent* a *FPM* závislost na `org.springframework.boot:spring-boot-starter-activemq`. Tento artefakt nám přináší do projektu autokonfiguraci pro ActiveMQ. *FPM* pak navíc potřebuje závislost `org.apache.activemq:activemq-broker`, která obsahuje kód samotného JMS serveru.

Konfigurace je provedena pomocí konfiguračních tříd `JmsContext`. Jelikož jsme chtěli ověřovat uživatele, tak jsme implementovali `ServerSecretAuthenticationPlugin` a v něm ještě `ServerSecretAuthenticationBroker`. Implementaci můžete vidět na obrázku 5.11. Ověření serveru probíhá pomocí *FSID* a *secret* vůči DB.

V ActiveMQ potřebujeme k odeslání zprávy mít určený cíl, tzv. *Topic*, což je unikátní řetězec. Z těchto cílů je poté možno zprávy číst. Toho docílíme snadno pomocí anotace

[1] http://activemq.apache.org/

```
class ServerSecretAuthenticationBroker(
        broker: Broker,
        val serverService: ServerService,
        val masterClientId: String,
        val masterSecret: String
) : AbstractAuthenticationBroker(broker){

    @Throws(Exception::class)
    override fun addConnection(context: ConnectionContext, info: ConnectionInfo) {
        var securityContext: SecurityContext? = context.securityContext

        if (securityContext == null) {
            securityContext = authenticate(info.clientId, info.password, null)
            context.securityContext = securityContext
            securityContexts.add(securityContext)
        }

        try {
            super.addConnection(context, info)
        } catch (e: Exception) {
            securityContexts.remove(securityContext)
            context.securityContext = null
            throw e
        }
    }

    override fun authenticate(username: String?, password: String?, peerCertificates: Array<out X509Certificate>?): SecurityContext {
        val fsid = username ?: throw SecurityException("No FSID provided")
        val secretToken = password ?: throw SecurityException("No authentication secret provided for fsid: $fsid")

        if (fsid == masterClientId && secretToken == masterSecret) {
            return ServerSecretSecurityContext(fsid)
        }

        if (!serverService.authenticate(fsid, secretToken)) {
            throw SecurityException("No server with secret $secretToken found")
        }

        return ServerSecretSecurityContext(fsid)
    }
}
```

Obrázek 5.11. Implementace JMS Brokera s autentizací

`@JmsListener`. Parametrem metody je samotná zpráva. Je důležité, aby daná třída byla serializovatelná.

```
@JmsListener(destination = "TopicName")
fun topicNameListener(message: TopicNameMessage) {
// listener code here ...
}
```

Abychom mohli zasílat zprávy jednotlivým serverům, každý si přihlásí svůj topic. Ten je složen z logického názvu a FSID. Ukázku můžeme vidět na následujících řádcích. Pokud chce *FPM* kontaktovat některého *FP Agent*, tak přesně ví, na jaký topic má zprávu odeslat.

```
@JmsListener(destination = "${Topic.REQUEST_CLIENT_LOG}-${fpm.fsid}")
```

5.4.3 Monitoring

K vyčítání informací o využití zdrojů jsme se rozhodli použít multiplatformní knihovnu OSHI[1], která nás odstíní od použitého OS. To je pro nás důležité zejména u on-premise klientů, kteří nejčastěji budou využívat Windows server. Do projektu je knihovna přidána v podobě Gradle závislosti `com.github.dblock:oshi-core:3.3`.

Vyčítání informací se odehrává ve třídě `HealthService`, kterou nalezneme v balíčku `me.factorify.platform.agent.health`. Vše důležité se odehrává ve funkci `heartbeat()`, která je periodicky spouštěna každých 30 s. Kód běží v jiném vlákně, aby neblokoval zbytek funkcí agenta. Veškeré důležité operace jsou zapisovány do logu.

[1] https://github.com/oshi/oshi

5.4.4 Alerting pomocí emailu

Jelikož není vždy možné sledovat vzniklé problémy ve webovém UI *FPM*, tak jsme implementovali i zasílání alertů emailem. Agregované jsou vždy dosud nezaslané alerty[1]. Kontrola nově vzniklých chyb se děje každých 5 minut. V případě, že jsou nějaké chyby identifikovány, tak jsou odeslány na emailové adresy. Ty jsou zadané v konfiguračním souboru `appliacation.properties` pod klíčem `fpm.alert.emails`. Emailovou adresu, ze které se email odeslal, je možné nastavit pomocí `fpm.alert.from`.

Konfigurace poštovního serveru je opět zjednodušena pomocí Spring Boot. Gradle závislost `org.springframework.boot:spring-boot-starter-mail` přidá do projektu potřebé knihovny, autokonfiguraci a vytvoří beanu třídy `JavaMailSender`, která poskytuje jednoduché API pro odesílání emailu.

Dále je třeba zadat konfiguraci emailového serveru, který bude alerty posílat. Jedná se především o hodnoty těchto nastavení:

- `spring.mail.host`,
- `spring.mail.port`,
- `spring.mail.username` a
- `spring.mail.password`.

5.5 Klientské komponenty *Factorify Platform*

V této sekci ukážeme zajímavé části implementace komponent, které jsou nasazeny na klientských serverech. Popsány jsou části, které doposud nebyly rozebrány v předchozím textu.

5.5.1 FP Agent

K monitorování nových řádků logu využíváme třídu `Tailer` z velmi oblíbené knihovny `Commons IO`[2]. Implementaci můžete vidět na obrázku 5.12. Na řádku 31 je ukázán způsob inicializace Tailer. Třída `LogTailWatcher` slouží jako posluchač události nového načtení řádku. Zpracování řádku se odehrává ve funkci `handle`, která je na řádku 34. V případě, že je řádek vyhodnocen jako chybový, je vyvolána událost `NewErrorLogEntryEvent`. Ta je následně zpracována a odeslána do *FPM*.

5.5.2 FP Gateway

Důležitou komponentou celého systému je service discovery. My jsme k tomu využili již hotové open-source řešení od Netflix – Eureka[3]. Eureka představuje server s RESTful API, který je umístěn v síti na známé URL – typicky `http://localhost:8761`. Do projektu je přidán Gradle závislostí `spring-cloud-starter-eureka-server`.

Jednotlivé služby se do Eureky zaregistrují a poté pomocí zasílaných heartbeatů signalizují, že jsou v pořádku a mohou na ně být zasílány požadavky. Eureka umí registrovat několik služeb stejného názvu, funguje poté i jako load balancer. V platformě se momentálně takto registrují dvě služby. První z nich je *Factorify* a to pod názvem

[1] V databázi zjistíme pomocí sloupečku `notification_sent`.
[2] https://commons.apache.org/proper/commons-io/
[3] https://github.com/Netflix/eureka

```kotlin
class LogTailWatcher(
        private val logFile: File,
        private val eventPublisher: ApplicationEventPublisher
) : TailerListenerAdapter() {
    companion object {
        val log: Logger = LoggerFactory.getLogger(LogTailWatcher::class.java)
        val LOG_READ_INTERVAL_MILLIS = 1000L
        val errorLineRegex: Regex = Regex("^[0-9\\:\\.\\-\\ ]*ERROR")

        @JvmStatic
        fun isErrorLine(line: String?): Boolean {
            if (line == null) return false
            return errorLineRegex.containsMatchIn(line)
        }
    }

    private lateinit var tailor: Tailer

    fun run() {
        log.info("Starting tail watch on the log file: " + logFile.absolutePath)
        Tailer.create(logFile, this, LOG_READ_INTERVAL_MILLIS, true)
    }

    override fun handle(line: String?) {
        if (isErrorLine(line)) {
            eventPublisher.publishEvent(NewErrorLogEntryEvent(logFile, line!!))
        }
    }
}
```

Obrázek 5.12. Hlídání kritických chyb v logu

`api`. Druhou je poté plánování a to pod názvem `api4`. Název služby se nastaví v konfiguračním souboru `application.properties` klíčem `spring.application.name`.

Zuul plní funkci gateway. Jedná se o další open-source projekt z dílny Netflix a plně se integruje s Eurekou. Pro zaregistrované služby automaticky vytváří směrovací pravidla tak, že požadavek `http://my.domain/api/request` je přeposlán na URL zaregistrovanou službou `api`.

Kromě automatického směrování máme implementovaný filtr, který se stará o poskytování statického obsahu ze složky `/fp-data/static`. Implementaci naleznete ve třídě `StaticFilter`. Je třeba zdůraznit, že tento filtr má prioritu. Teprve pokud není nalezen statický obsah, tak je směrování předáno filtru Zuulu. Důležitou část implementace můžete vidět na obrázku 5.13.

Důležitou roli ve filtru vykonává metoda `doFilter` začínající na řádku 41. Je zde vidět, že nejprve se zkouší, zda doména nevyhovuje směrovacímu pravidlu z některého souboru `.rewrite`. Pokud nevyhovuje, tak veškeré URI krom „/“ jsou otestovány, zda nepředstavují statický obsah na disku. V případě, že ano, tak je soubor načten a vrácen jako odpověď. V opačném případě se předá kontrola dalšímu filtru, který je velice pravděpodobně filtr Zuulu.

```java
public class StaticFilter implements Filter {
    private static Logger log = LoggerFactory.getLogger(StaticFilter.class);

    private String staticPath;
    private static final String INDEX_FILE = "index.html";
    private static final String ROUTING_CONFIG_FILE = ".rewrite";
    private static final String CONFIG_SEPARATOR = "=";
    private final Map<String, String> domainRoutes = new HashMap<>();

    public StaticFilter(String staticPath) { this.staticPath = staticPath; }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        log.info("Registering StaticFilter. Assets served from: " + staticPath);
        collectDomainRoutes();
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
        if (servletRequest instanceof HttpServletRequest) {
            HttpServletRequest httpRequest = (HttpServletRequest) servletRequest;
            String requestUri = httpRequest.getRequestURI();
            String domainName = httpRequest.getServerName();

            // Handle specific static files routed based on domain
            if(domainRoutes.containsKey(domainName)) {
                Path absoluteAssetPath = Paths.get(domainRoutes.get(domainName), requestUri);
                if (serveAssetIfExists(absoluteAssetPath, domainName, requestUri, servletResponse)) {
                    return;
                }
            }

            // Index route to factorify
            if (requestUri.equals("/")) {
                filterChain.doFilter(servletRequest, servletResponse);
                return;
            }

            // Handle generic static files
            Path absoluteAssetPath = Paths.get(staticPath, requestUri);
            if(serveAssetIfExists(absoluteAssetPath, domainName, requestUri, servletResponse)) {
                return;
            }
        }

        filterChain.doFilter(servletRequest, servletResponse);
    }
```

Obrázek 5.13. Část filtru pro statický obsah

Kapitola 6
Zhodnocení

Myšlenka vytvoření platformy vznikla již před rokem a půl. Od té doby jsme museli vyřešit několik desítek problémů při návrhu a následném vývoji. Poté, co se nám nepodařilo nalézt vhodný existující software, jsme se rozhodli pro vytvoření systému vlastního, na míru našim potřebám. Po řádném prozkoumání možností jsme pro distribuci služeb použili Docker. Mezi hlavními důvody byla dobrá podpora a dokumentace, velká komunita, mnoho možností pro spuštění dockeru v prostředí cloudu a nově možnost nativního běhu v OS Windows.

Hlavní motivací byla automatizace nasazování, aktualizace klientů a generování release notes. Dále pak jednoduchá správa modulárních částí aplikace, klientů a serverů. V případě poruchy pak zasílání upozornění emailem a poskytnutí základního monitoring zdrojů. Základní požadavky se nám podařilo kompletně splnit a během dosavadního provozu *FPM* jsme ocenili zejména monitoring a alerting chyb.

K dokončení modularizace aplikace *Factorify* nás čeká rozpadnutí gradle modulů *Factorify* do samostatných maven artefaktů. Tímto přesuneme konfiguraci klientů z build skriptů do nastavení v *FPM*. Aktuální klienti, kteří jsou spuštěni v platformě jsou udržovány v Gitu ve vlastní branch. Důvodem je přítomnost nutných modifikací pro běh v platformě. Konkrétně se jedná o Eureku zajišťující registraci do služby service discovery.

6.1 Modularita *FPM*

Jelikož od začátku vývoje počítáme s možností uvolnění systému jako open-source, tak jsou důležité části naprogramovány obecně. V zásadě to znamená, že je programováno vůči rozhraní. Konkrétně se jedná o napojení na ticketovací systém, build systém a systém pro poskytování VPS v cloudu. Dále je pak možné přidat různé alerty či definovat bloky kódu (hooky), které se mají spustit před a po vytvoření samotného VPS.

Konkrétní implementace vznikly pouze pro potřebu spuštění *FPM* se systémem *Factorify*. Změna je velice jednoduchá. Buď se jedná o automatické přidání třídy implementující konkrétní rozhraní za pomocí anotace `@Component`, což za nás zařídí framework SpringBoot, nebo musíme třídu manuálně zaregistrovat v konfigurační třídě – anotace `@Configuration`.

6.2 Nasazení

V době dokončování DP, konec května 2017, máme v platformě 4 servery. Z toho 2 prostředí jsou produkční a 2 prostředí byla vytvořena v OVH cloudu.

6.3 Budoucnost

Jak již bylo zmíněno, do platformy chceme během několika následujících měsíců přenést všechny klienty. Ti aktuálně běží v Jailu na FreeBSD. Před samotnou migrací budeme na méně kritických klientech testovat funkčnost. Počítáme s tím, že se během zátěže mohou objevit chyby, které během vývoje nebylo možné odchytit.

V platformě budou evidováni klienti běžící v cloudu i ti nasazení on-premise.

6.4 Open-source

V plánu je uvolnění celé platformy pod open-source licencí. Nejdříve ale potřebujeme přejít na *FPM* kompletně uvnitř *Factorify*. Do té doby je možné, že celá aplikace projde nějakou zásadnější úpravou.

Po uvolnění zdrojových kódů musíme mít také vyhrazený čas k reakci na uživatelské připomínky, nahlášené chyby a pull requesty.

Kapitola 7
Závěr

Factorify Platform je založená na moderních technologiích – cloud, Kotlin, SpringBoot, Docker, Angular 2. Platforma do správy jednotlivých klientů přináší strukturu, jednoduchost a automatizaci. Při implementaci jsme se snažili maximum funkcionality pokrýt pomocí existujících řešení, což bylo důvodem integrace na řadu služeb.

Realizaci systému hodnotíme kladně. Hlavním ukazatelem je fakt, že systém úspěšně používáme v produkčním nasazení u několika menších klientů. Nutno říci, že ne všechny části *Factorify Platform* jsou aktuálně plně využity. Je to z důvodu postupné migrace do platformy. V částech, kde platformu využíváme naplno, sledujeme úsporu nutné administrativy a času. Úsporu očekáváme také v místech doposud neotestovaných produkčním nasazením.

Během vývoje jsme vyzkoušeli nové technologie. Zejména rozhodnutí zvolit pro implementaci programovací jazyk Kotlin místo Javy byl obohacující. Tato zkušenost potvrdila naši myšlenku začít Kotlin využívat i v dalších projektech.

Použití kontejnerů pro distribuci aplikace hodnotíme kladně. Naše zkušenost se vztahuje zejména k Dockeru, který jsme pro tento účel využili. Během vývoje a nasazení jsme se setkali s několika problémy, ale to provází pravděpodobně každou technologii. Z pohledu výkonu si aplikace v Dockeru vedla téměř stejně jako mimo něj. Správa a distribuce kódu je za použití privátního docker repositáře mnohem snazší.

Literatura

[1] ČVUT FEL. *Směrnice děkana pro magisterské státní závěrečné zkoušky na ČVUT FEL*. 2012.
http://www.fel.cvut.cz/rozvoj/smerniceMSZZ.html.

[2] Přispěvatelé Wikipedia. *Java classpath - referenční příručka*. 2017.
https://en.wikipedia.org/wiki/Classpath_(Java).

[3] Parallel Universe. *Capsule*. 2017.
http://www.capsule.io/.

[4] Ron Fabio. *Capsule - možnosti nasazení*. 2015.
http://blog.paralleluniverse.co/2015/09/10/capsule-1-0/.

[5] Github. *Capsule - aktivita projektu*. 2017.
https://github.com/puniverse/capsule/graphs/contributors.

[6] Přispěvatelé Wikipedia. *Operating-system-level virtualization*. 2017.
https://en.wikipedia.org/wiki/Operating-system-level_virtualization.

[7] Docker Inc. *Docker diagram*. 2017.
https://docs.docker.com/get-started/#container-diagram.

[8] Docker Inc. *Dockerfile - referenční příručka*. 2017.
https://docs.docker.com/engine/reference/builder/.

[9] Phusion. *Baseimage-docker*. 2017.
http://phusion.github.io/baseimage-docker/.

[10] Docker Inc. *Docker pro Windows - instalace*. 2017.
https://docs.docker.com/docker-for-windows/install/#what-to-know-before-you-install.

[11] Přispěvatelé Wikipedia. *Orchestrace*. 2017.
https://cs.wikipedia.org/wiki/Orchestrace_(informatika).

[12] RightScale. *Využití veřejného cloudu*. 2017.
http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2017-state-cloud-survey.

[13] Anthony T. Velte, Toby J. Velte a Robert C. Elsenpeter. *Cloud computing: praktický průvodce*. In: Brno: Computer Press, 2011. 90-101. ISBN 9788025133330;.

[14] OVH. *Síť OVH v Evropě*. 2017.
https://www.ovh.cz/discover/.

[15] Nick Antonopoulos a Lee Gillam. *Cloud computing: principles, systems and applications*. In: London: Springer, 2010. 29-30. ISBN 9781849962407;.

[16] PostgreSQL Global Development Group. *Vývojáři PostgreSQL*. 2017.
https://www.postgresql.org/community/contributors/.

[17] Oracle Corporation and/orits affiliates. *Referenční příručka MySQL 8.0*. 2017.
https://dev.mysql.com/doc/refman/8.0/en/.

[18] Ebean a kolektiv. *Ebean ORM v porovnání s JPA*. 2017.
http://ebean-orm.github.io/architecture/compare-jpa.

[19] Docker Inc. *Docker Compose - referenční příručka*. 2017.
https://docs.docker.com/compose/compose-file/.

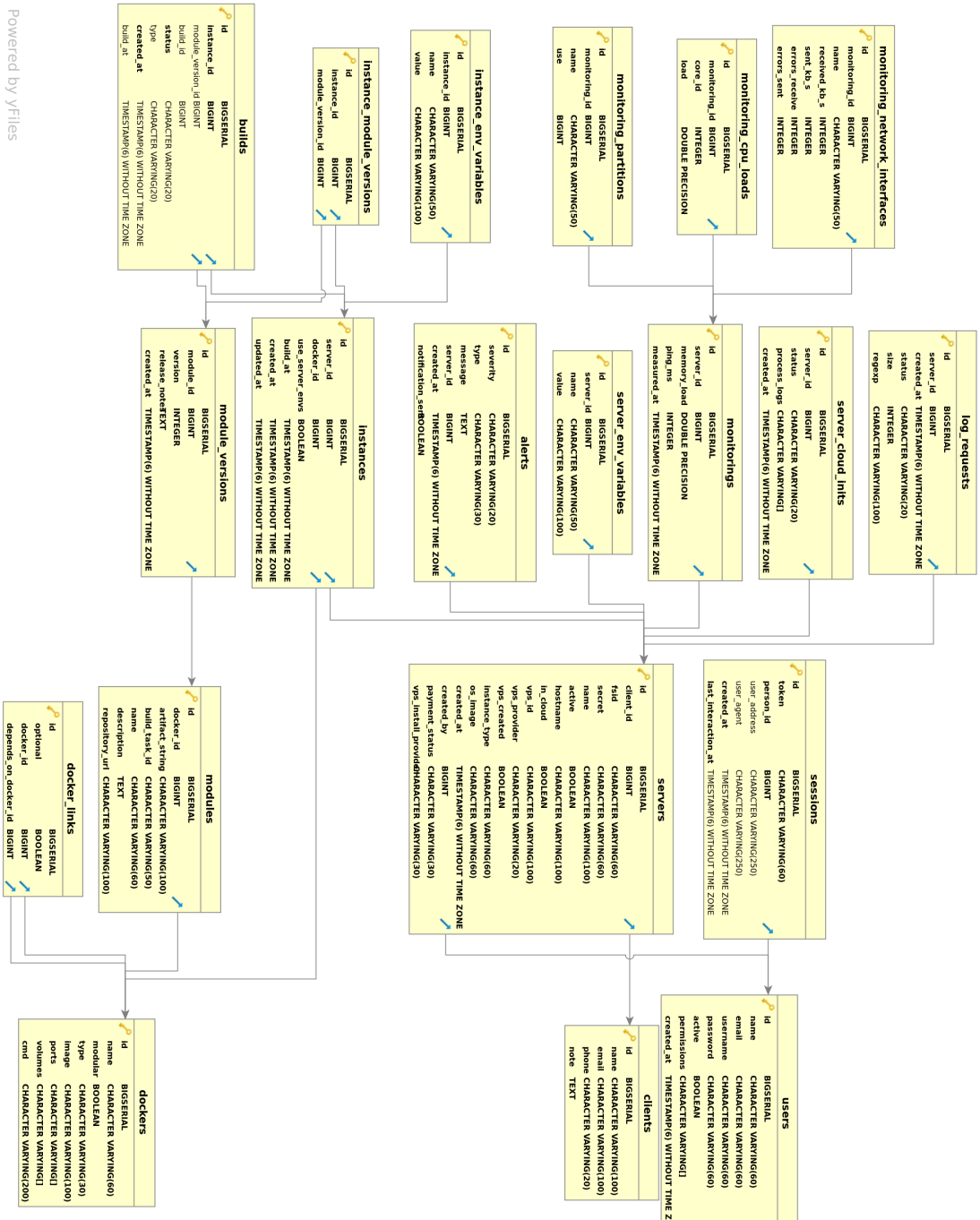
[20] iDataLabs. *iDataLabs - statistika použití vývojových nástrojů*. 2017.
https://idatalabs.com/tech/software-development-tools.

[21] JetBrains. *TeamCity ceník*. 2017.
https://www.jetbrains.com/teamcity/buy/#license-typeñew-license.

[22] Angular 2. *Angular 2 - architektura*. 2017.
https://angular.io/docs/ts/latest/guide/architecture.html.

[23] Petr Skalička. *Implementace frontendu FPM, veřejně nepublikováno*. 2017.

Příloha A
Databázové schema



Příloha B
Popis REST API

URL	Metoda	Typ obsahu
/server/{id}	GET	application/json
/servers	GET	application/json
/server	POST	application/json
/server/{id}	POST	application/json
/server/{id}/info	GET	application/json
/server/{id}/monitoring	GET	application/json
/server/{id}/run-cloud	POST	application/json
/server/providers	GET	application/json
/server/install-providers	GET	application/json
/server/{id}/installed	GET	application/json
/server/{fsid}/email	GET	text/plain
/server/changeLog	GET	application/json
/server/{fsid}/gcr-key	GET	text/plain
/server/{fsid}/install-script/{type}	GET	text/plain
/server/{fsid}/compose	GET	text/plain
/login	POST	application/json
/logout	POST	application/json
/session/{securityToken}	GET	application/json
/build/instance/{id}	POST	application/json
/build/instance/scheduled/{id}	POST	application/json
/build/module/{id}	POST	application/json
/build/status/{id}	GET	application/json
/dockers	GET	application/json
/docker/{id}	GET	application/json
/docker	POST	application/json
/docker/{id}	POST	application/json
/docker/options	GET	application/json
/docker/types	GET	application/json
/monitoring/stats	GET	application/json
/instance/update	POST	application/json
/users	GET	application/json
/user/{id}	GET	application/json
/user	POST	application/json
/user/{id}	POST	application/json
/client/{id}	GET	application/json
/clients	GET	application/json
/client	POST	application/json
/client/{id}	POST	application/json
/server/{id}/request-logs	POST	application/json
/server/{id}/logs	GET	application/json
/log/{id}	GET	application/zip
/module/{id}/versions	GET	application/json
/alerts	GET	application/json

Příloha C
Kalkulace cen cloudu

	USD_IN_CZK	25												
			VS Hosting	AWS			Google cloud platform		OVH	Azure				
Cloud	Cena / měsíc – cluster [USD]	1354.4		Docker cluster (On demand)	1261	Docker cluster (prepaid 3Y)	469	VPS (prepaid 3Y)	Cloud	VPS	VPS	VPS		
	Přenost dat OUT / měsíc / klienta [GB]				70		70		70					
	Přenos dat celkem	0			2100		2100		2100		0		0	
	Cena / GB				0.09		0.09		0.115					
	Přenos – Cena / měsíc	***0***			***189***		***189***		***241.5***		***0***		***0***	
	Storage / klienta [GB]				20		20		20					
	Storage celkem	0			600		600		600		0		0	
	Cena / GB				0.12		0.12		0.04					
	Storage – Cena/ měsíc	***0***			***72***		***72***		***24***		***0***		***0***	
	Klientů	30			30		30		30		30		30	
	Technické detaily		2x (2x Intel Xeon 8-core) • 64GB RAM • 2x 120GB SSD v RAID1 - 500GB SAS	- 2x Instance m4.4xlarge - 16vCPU - 64GB RAM - SSD (160MB/s)		- 2x Instance m4.4xlarge - 16vCPU - 64GB RAM - SSD (160MB/s)								
	Cena na klienta [USD]	**45.1466666666667**		**50.7333333333333**		**24.3333333333333**		**6.3**	**38.8166666666667**		**0**		**0**	
Load balancer	Cena / měsíc / klient				1.5		1.5							
	Technické detaily		1x Intel Xeon E3-1230v3 • 16GB RAM • 2x 120GB SSD v RAID1 • HW RAID radič • 4x ethernet											
VPS	Cena / měsíc [USD]							18.97	49.78	21.6	55.8			
	Cena za GB							0.12	0.17					
	Disk [GB]							50	50					
	Cena za disk [USD]							6	0	0	0	Instance A2 - 2 vCPU - 3.5GB RAM - 60GB HDD		
	Technické detaily						- Instance t2.medium - 2 vCPU - 4GB RAM		Custom 2-4 - 2 vCPU - 4GB RAM					
	Cena / měsíc / klient	**0**		**0**		**0**	**24.97**		**0**	**58.28**	**21.6**		**55.8**	
Databáze	Cena / měsíc [USD]				1229		459	459	1460	1460	345.92		882	
	Diskový prostor / klient [GB]				4		4	4	4	4				
	Diskový prostor [GB]	0			120		120	120	120	120	0		0	
	Cena / GB				0.12		0.12	0.12	0.17	0.17				
	Cena za diskový prostor [USD]				14.4		14.4	14.4	20.4	20.4	0		0	
	Přenos dat / měsíc [GB]													
	Cena / GB													
	Cena přenosu [USD]				0									
	Technické detaily		(cena db již) 2x (2x Intel Xeon 8-core) • 64GB RAM • 2x 400GB SSD v RAID1 • 2x 2000GB SATA v RAID1	- Instalaci DB řešíme samy - 2x Instance c3.4xlarge - 16vCPU - 30GB RAM - 2x160GB SSD (primo v PC)		- Instalaci DB si řešíme samy - 2x Instance c3.4xlarge - 16vCPU - 30GB RAM - 2x160GB SSD (primo v PC)	- Instalaci DB si řešíme sam - 2x Instance c3.4xlarge - 16vCPU - 30GB RAM - 2x160GB SSD (primo v PC	2x n1-standard-16 - 16vCPU - 60GB RAM	2x n1-standard-16 - 16vCPU - 60GB RAM	2x EG-60 - 16vCore 2.3GHz - 60GB RAM - 1600GB HA - 500Mbit	Instance D13 v2 - 8vCPU - 56GB RAM - 400GB HDD	
	Cena / měsíc / klient [USD]	**0**		**41.4666666666667**		**15.78**	**15.78**	**49.3466666666667**	**49.3466666666667**	**11.5306666666667**		**29.4**		
Poznámka								https://cloud.googl	https://cloud.google.com/products/calculator#id=fcf703fb-0					
	Cena / klienta [USD]	45.1		93.7		41.6	47.1	88.2	107.6	33.1	85.2			
	Cena / klienta [CZK]	**1127.5**		**2342.5**		**1040**	**1177.5**	**2205**	**2690**	**827.5**	**2130**			